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

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY PUBLISHING COMPANY

Volume 1 / **Fundamental Algorithms**

THE ART OF COMPUTER PROGRAMMING

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

This book is in the

ADDISON-WESLEY SERIES IN

COMPUTER SCIENCE AND INFORMATION PROCESSING

RICHARD S. VARGA and MICHAEL A. HARRISON, Editors

Second Printing 1969

COPYRIGHT © 1968 BY ADDISON-WESLEY PUBLISHING COMPANY, INC. ALL RIGHTS RESERVED. NO PART OF THIS PUBLICATION MAY BE REPRODUCED, STORED IN A RETRIEVAL SYSTEM, OR TRANSMITTED, IN ANY FORM OR BY ANY MEANS, ELECTRONIC, MECHANICAL, PHOTOCOPYING, RECORDING, OR OTHERWISE, WITHOUT THE PRIOR WRITTEN PERMISSION OF THE PUBLISHER. PRINTED IN THE UNITED STATES OF AMERICA. PUBLISHED SIMULTANEOUSLY IN CANADA. LIBRARY OF CONGRESS CATALOG CARD NO. 67-26020.

PREFACE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Volume 1. Fundamental Algorithms

Chapter 1. Basic Concepts
Chapter 2. Information Structures

Volume 2. Seminumerical Algorithms

Chapter 3. Random Numbers
Chapter 4. Arithmetic

Volume 3. Sorting and Searching

Chapter 5. Sorting Techniques
Chapter 6. Searching Techniques

Volume 4. Combinatorial Algorithms

Chapter 7. Combinatorial Searching
Chapter 8. Recursion

Volume 5. Syntactical Algorithms

Chapter 9. Lexical Scanning
Chapter 10. Parsing Techniques

Volume 6. Theory of Languages

Chapter 11. Mathematical Linguistics

Volume 7. Compilers

Chapter 12. Programming Language Translation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CACM = Communications of the Association for Computing Machinery

JACM = Journal of the Association for Computing Machinery

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

Math. Comp. = Mathematics of Computation

AMM = American Mathematical Monthly

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

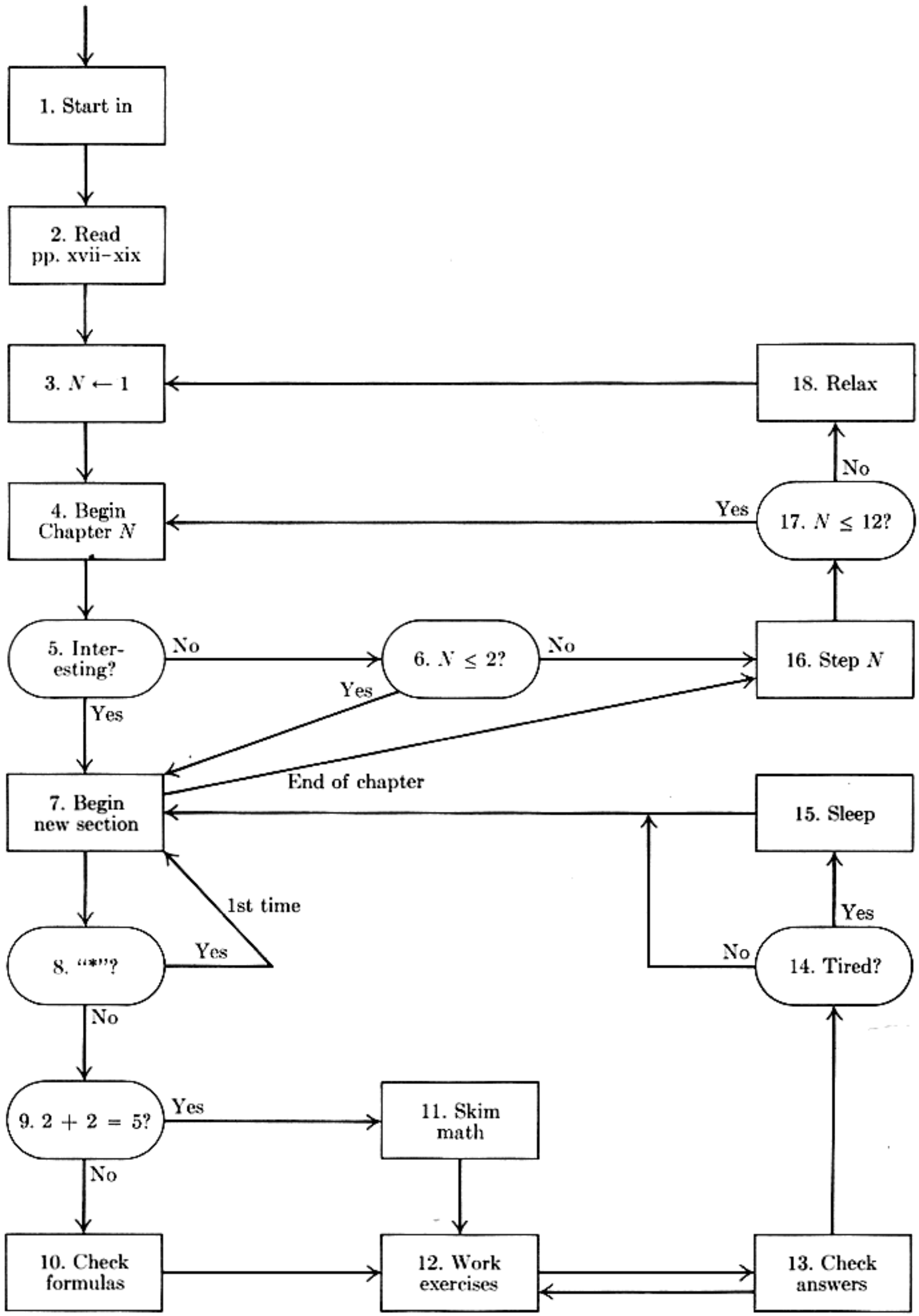
I have, of course, received assistance from a great many people during the five years I have been preparing these books, and for this I am extremely thankful. Acknowledgments are due, first, to my wife, Jill, for her infinite patience, for being the first guinea pig in reading the manuscript, and for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

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

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

Pasadena, California D. E. K.
October 1967

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



Flow chart for reading this set of books.

Procedure for Reading This Set of Books

1. Begin reading this procedure, unless you have already begun to read it. *Continue to follow the steps faithfully.* (The general form of this procedure and its accompanying flowchart will be used throughout this book.)
2. Read the Notes on the Exercises, pp. xvii–xix.
3. Set N equal to 1.
4. Begin reading chapter N. Do *not* read the quotations which appear at the beginning of the chapter.
5. Is the subject of the chapter interesting to you? If so, go to step 7; if not, go to step 6.
6. Is $N \leq 2$? If not, go to step 16; if so, scan through the chapter anyway. (Chapters 1 and 2 contain important introductory material and also a review of basic programming techniques. You should at least skim over the sections on notation and about `MIX`.)
7. Begin reading the next section of the chapter; if you have reached the end of the chapter, go to step 16.
8. Is section number marked with "*"? If so, you may omit this section on first reading (it covers a rather specialized topic which is interesting but not essential); go back to step 7.
9. Are you mathematically inclined? If math is all Greek to you, go to step 11; otherwise go to step 10.
10. Check the mathematical derivations made in this section (and report errors to the author). Go to step 12.
11. If the current section is full of mathematical computations, you had better omit reading the derivations. However, you should become familiar with the basic results of the section; these are usually stated near the beginning or in italics right at the very end of the hard parts.
12. Work the recommended exercises in this section in accordance with the hints given in the Notes on the Exercises (which you read in step 2).

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

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

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

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

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

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

Notes on the Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

Summary of codes:

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

EXERCISES

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

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

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

CONTENTS

Chapter 1—Basic Concepts

1.1. Algorithms 1
1.2. Mathematical Preliminaries 10
1.2.1. Mathematical Induction 11
1.2.2. Numbers, Powers, and Logarithms 21
1.2.3. Sums and Products 26
1.2.4. Integer Functions and Elementary Number Theory 37
1.2.5. Permutations and Factorials 44
1.2.6. Binomial Coefficients 51
1.2.7. Harmonic Numbers 73
1.2.8. Fibonacci Numbers 78
1.2.9. Generating Functions 86
1.2.10. Analysis of an Algorithm 94
*1.2.11. Asymptotic Representations 104
1.2.11.1. The O-notation 104
1.2.11.2. Euler's summation formula 108
1.2.11.3. Some applications 112
1.3. MIX 120
1.3.1. Description of MIX 120
1.3.2. The MIX Assembly Language 141
1.3.3. Applications to Permutations 160
1.4. Some Fundamental Programming Techniques 182
1.4.1. Subroutines 182
1.4.2. Coroutines 190
1.4.3. Interpretive Routines 197
1.4.3.1. A MIX simulator 198
*1.4.3.2. Trace routines 208
1.4.4. Input and Output 211
1.4.5. History and Bibliography 225

Chapter 2—Information Structures

2.1. Introduction 228
2.2 Linear Lists 234
2.2.1. Stacks, Queues, and Deques 234
2.2.2. Sequential Allocation 240
2.2.3. Linked Allocation 251
2.2.4. Circular Lists 270

2.2.5. Doubly Linked Lists 278
2.2.6. Arrays and Orthogonal Lists 295
2.3. Trees 305
2.3.1. Traversing Binary Trees 315
2.3.2. Binary Tree Representation of Trees 332
2.3.3. Other Representations of Trees 347
*2.3.4. Basic Mathematical Properties of Trees 362
2.3.4.1. Free trees 362
2.3.4.2. Oriented trees 371
2.3.4.3. The "infinity lemma" 381
2.3.4.4. Enumeration of trees 385
2.3.4.5. Path length 399
2.3.4.6. History and bibliography 405
2.3.5. Lists and Garbage Collection 406
2.4. Multilinked Structures 423
2.5. Dynamic Storage Allocation 435
2.6. History and Bibliography 456

Answers to Exercises 465

Appendix A—Index to Notations 607

Appendix B—Tables of Numerical Quantities

1. Fundamental Constants (decimal) 613
2. Fundamental Constants (octal) 614
3. Harmonic Numbers, Bernoulli Numbers, Fibonacci Numbers . . . 615

Index and Glossary 617

CHAPTER ONE

BASIC CONCEPTS

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

— ADA AUGUSTA, Countess of Lovelace (1844)

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

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

1.1. ALGORITHMS

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

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

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

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

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

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

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

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

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

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

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

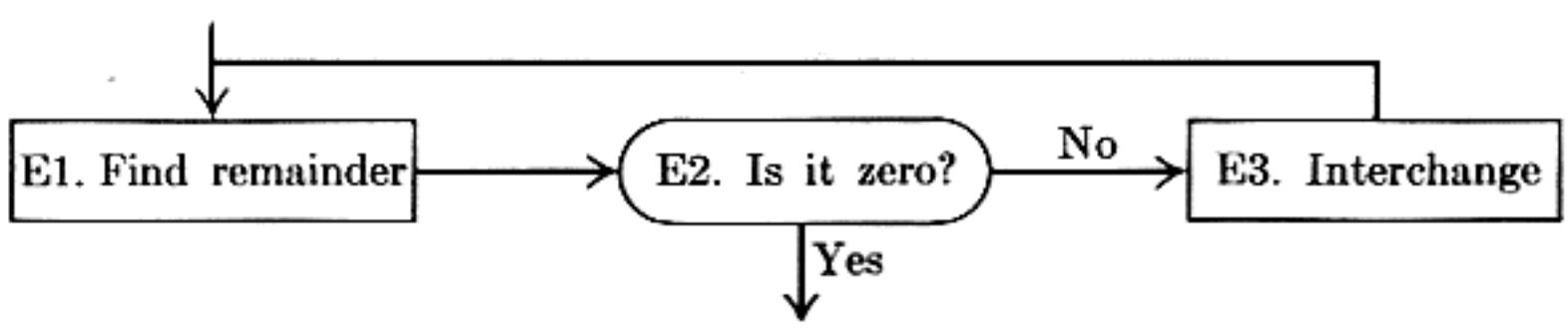


Fig. 1. Flow chart for Algorithm E.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$m = qn + r,$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The correspondence between this notation and Algorithm E is evident.

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

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

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

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

EXERCISES

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

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

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

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

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

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

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

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

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

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

1.2. MATHEMATICAL PRELIMINARIES

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

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

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

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

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

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

1.2.1. Mathematical Induction

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

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

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

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

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

We can formulate the general property as follows:

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

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

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

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

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

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

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

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

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

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

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

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

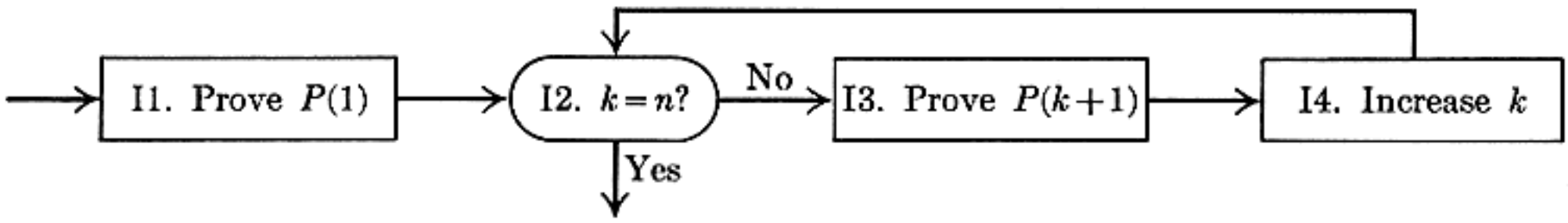


Fig. 2. Algorithm I: Mathematical induction.

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

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

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

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

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

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

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

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

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

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

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

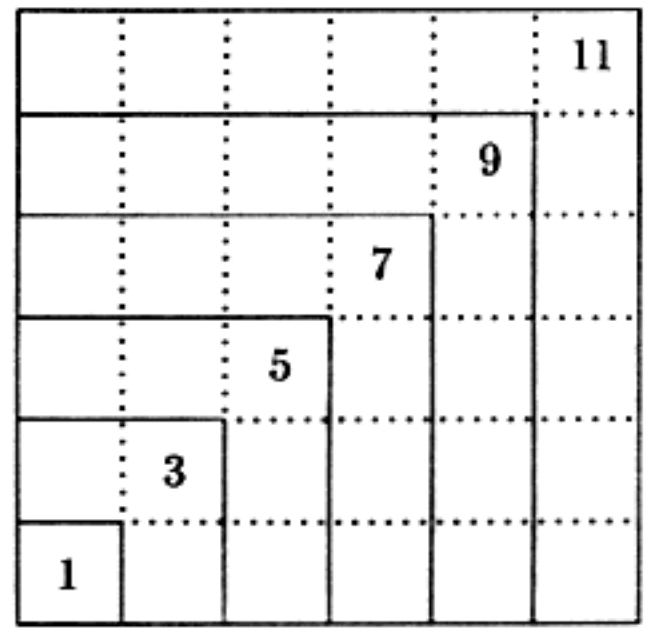


Fig. 3. The sum of odd numbers is a square.

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

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

for all positive integers n.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

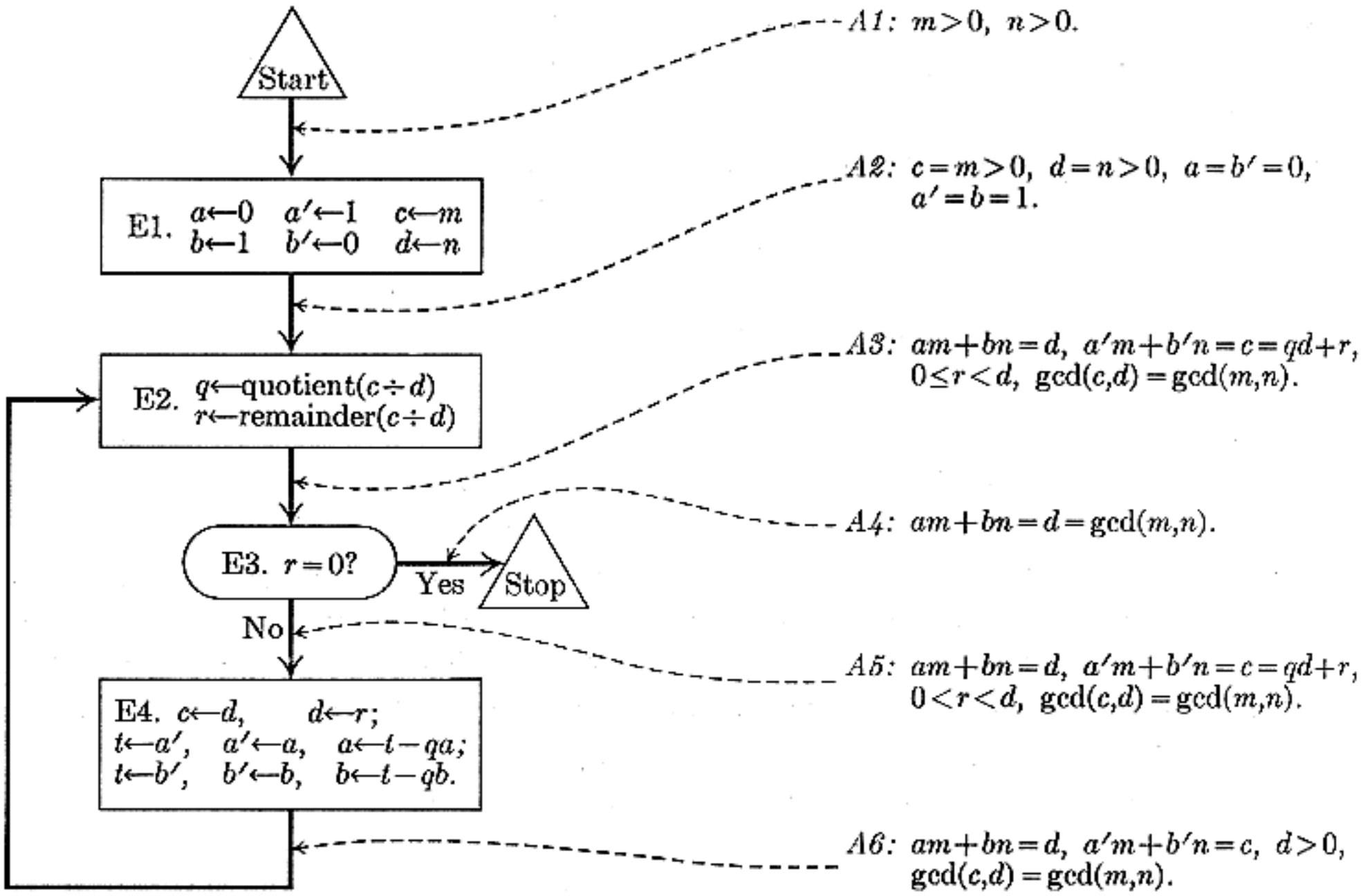


Fig. 4. Flow chart for Algorithm E, labeled with assertions which prove the validity of the algorithm.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXERCISES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Side $= 5+5+5+5+5+5 = 5\cdot(5+1)$

Side $= 5+4+3+2+1+1+2+3+4+5$
$= 2(1+2+\cdots+5)$

Area $= 4\cdot 1^2 + 4\cdot 2\cdot 2^2 + 4\cdot 3\cdot 3^2 + 4\cdot 4\cdot 4^2 + 4\cdot 5\cdot 5^2$
$= 4(1^3 + 2^3 + \cdots + 5^3)$

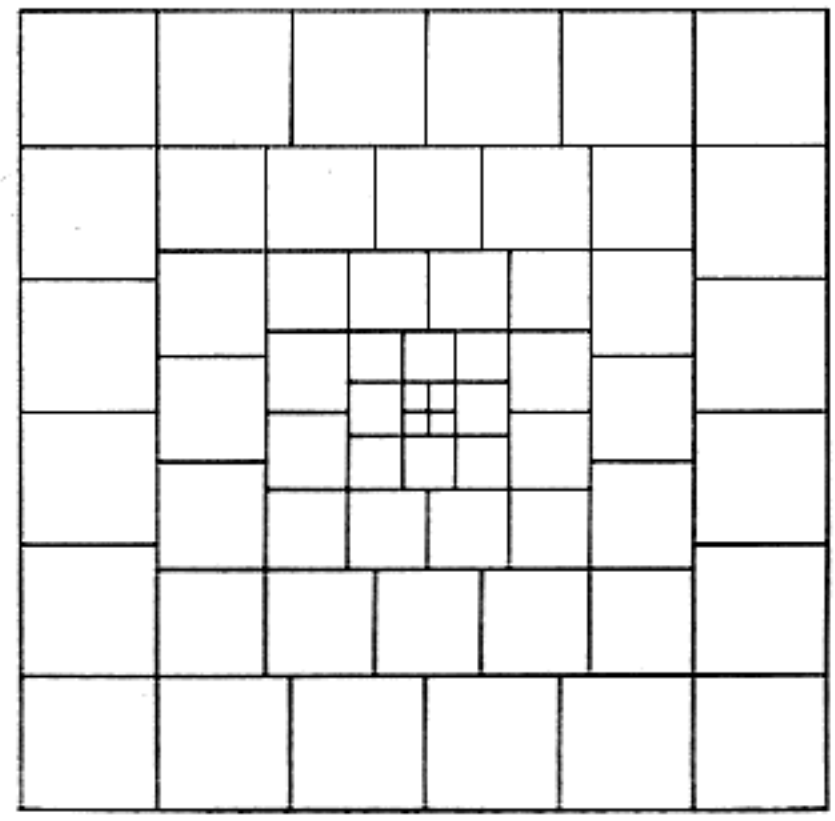


Fig. 5. Geometric version of exercise 8, with $n = 5$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.2. Numbers, Powers, and Logarithms

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

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

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

$$x = n + 0.d_1 d_2 d_3 \ldots, \tag{1}$$

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

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

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

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

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

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

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

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

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

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

whenever x and y are integers.

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

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

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

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

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

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

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

For example, we find that

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

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

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

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

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

As an example, Eqs. (7) show that

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

From the laws of exponents it follows that

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

and

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

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

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

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

Hence

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

and in general we find that

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

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

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

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

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

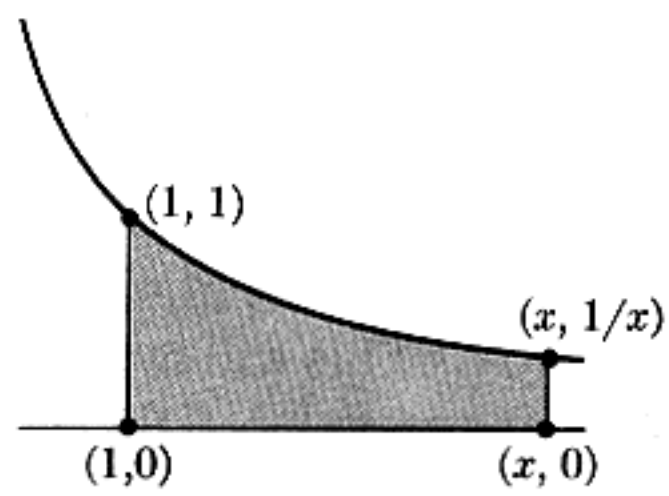


Fig. 6. Natural logarithm.

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

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

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

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

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

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

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

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

The validity of this procedure follows from the fact that

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

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

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

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

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

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

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

EXERCISES

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

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

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

▸ 4. [*05*] What is $(0.125)^{-2/3}$?

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

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

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

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

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

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

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

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

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

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

15. [*10*] Prove or disprove:

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

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

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

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

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

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

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

▸ 22. [*20*] Prove that

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

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

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

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

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

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

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

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

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

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

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

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

▸ **27.** [*M25*] Consider the method for calculating $\log_{10} x$ discussed in the text. Let x'_k denote the computed approximation to x_k, determined as follows: $x(1 - \eta) \le 10^n x'_0 \le x(1 + \epsilon)$; and in the determination of x'_k by Eqs. (15), the quantity y_k is used in place of $(x'_{k-1})^2$, where $(x'_{k-1})^2(1 - \eta) \le y_k \le (x'_{k-1})^2(1 + \epsilon)$. Here η and ϵ are small constants which reflect the upper and lower errors due to rounding or truncation. If $\log' x$ denotes the result of the calculations, show that after k steps we have

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

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

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

1.2.3. Sums and Products

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, consider the special case

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

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

b) *Change of variable:*

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

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

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

The reader should study this example carefully.

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

c) *Interchanging order of summation:*

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

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

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

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

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

where we let

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

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

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

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

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

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

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

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

For example,

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

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

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

Example 1.

$$\begin{aligned}
\sum_{0\le j\le n} a_j &= \sum_{\substack{0\le j\le n,\\ j \text{ even}}} a_j + \sum_{\substack{0\le j\le n,\\ j \text{ odd}}} a_j && \text{by rule (d)}\\
&= \sum_{\substack{0\le 2j\le n,\\ 2j \text{ even}}} a_{2j} + \sum_{\substack{0\le 2j+1\le n,\\ 2j+1 \text{ odd}}} a_{2j+1} && \text{by rule (b)}\\
&= \sum_{0\le j\le n/2} a_{2j} + \sum_{0\le j<n/2} a_{2j+1}.
\end{aligned}$$

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

Example 2. Let

$$\begin{aligned}
S_1 = \sum_{0\le i\le n}\ \sum_{0\le j\le i} a_i a_j &= \sum_{0\le j\le n}\ \sum_{j\le i\le n} a_i a_j && \text{by rule (c)} \quad \text{[cf. Eq. (10)]}\\
&= \sum_{0\le i\le n}\ \sum_{i\le j\le n} a_i a_j && \text{by rule (b),}
\end{aligned}$$

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

$$\begin{aligned}
2S_1 = S_1 + S_2 &= \sum_{0\le i\le n}\left(\sum_{0\le j\le i} a_i a_j + \sum_{i\le j\le n} a_i a_j\right) && \text{by Eq. (8)}\\
&= \sum_{0\le i\le n}\left(\left(\sum_{0\le j\le n} a_i a_j\right) + a_i a_i\right) && \text{by rule (d)}\\
& && \quad \text{[cf. Eq. (12)]}\\
&= \sum_{0\le i\le n}\ \sum_{0\le j\le n} a_i a_j + \sum_{0\le i\le n} a_i a_i && \text{by Eq. (8)}\\
&= \left(\sum_{0\le i\le n} a_i\right)\left(\sum_{0\le j\le n} a_j\right) + \left(\sum_{0\le i\le n} a_i^2\right) && \text{by rule (a)}\\
&= \left(\sum_{0\le i\le n} a_i\right)^2 + \left(\sum_{0\le i\le n} a_i^2\right) && \text{by rule (b).}
\end{aligned}$$

Thus we have derived the important identity

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

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

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

Comparing the first relation with the fifth, we have

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

and so we obtain the basic formula

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

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

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

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

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

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

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

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

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

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

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

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

A further example which demonstrates the usefulness of this notation is

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

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

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

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

EXERCISES—First Set

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

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

and also the equivalent of

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16. [*M22*] Prove that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

assuming that $0 < a_j < 1$.

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

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

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

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

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

32. [*M20*] Prove that

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

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

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

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

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

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

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

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

34. [*M25*] Prove that

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

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

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

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

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

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

EXERCISES—Second Set

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

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

The following types of matrices are of special importance:

Vandermonde's matrix,

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

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

Combinatorial matrix,

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

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

Cauchy's matrix,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.4. Integer Functions and Elementary Number Theory

If x is any real number, we write

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

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

The following formulas and examples are easily verified:

$$\lfloor \sqrt{2} \rfloor = 1, \qquad \lceil \sqrt{2} \rceil = 2;$$
$$\lfloor +\tfrac{1}{2} \rfloor = 0, \qquad \lceil -\tfrac{1}{2} \rceil = 0, \qquad \lfloor -\tfrac{1}{2} \rfloor = -1 \quad (\textit{not} \text{ zero!});$$
$$\lceil x \rceil = \lfloor x \rfloor \quad \text{if and only if } x \text{ is an integer},$$
$$\lceil x \rceil = \lfloor x \rfloor + 1 \quad \text{if and only if } x \text{ is not an integer};$$
$$\lfloor -x \rfloor = -\lceil x \rceil; \qquad x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1.$$

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

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

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

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

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

therefore

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Multiplying each side of this congruence by a, we obtain

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

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

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

EXERCISES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In particular, prove that

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

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

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

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

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

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

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

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

c) Given that q is odd, show that $\sum_{0\le k<p/2} \lfloor 2kq/p \rfloor \equiv \sum_{0\le k<p/2} \lfloor kq/p \rfloor$ (modulo 2). [*Hint:* Consider $\lfloor (p-1-2k)q/p \rfloor$.]

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

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

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

1.2.5. Permutations and Factorials

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and, renaming, we get

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

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

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

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

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

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

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

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

and with this convention the basic identity

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

is valid for all positive integers n.

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

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

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

It is helpful to keep the value

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Stirling tried again, by noticing that

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

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

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

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

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

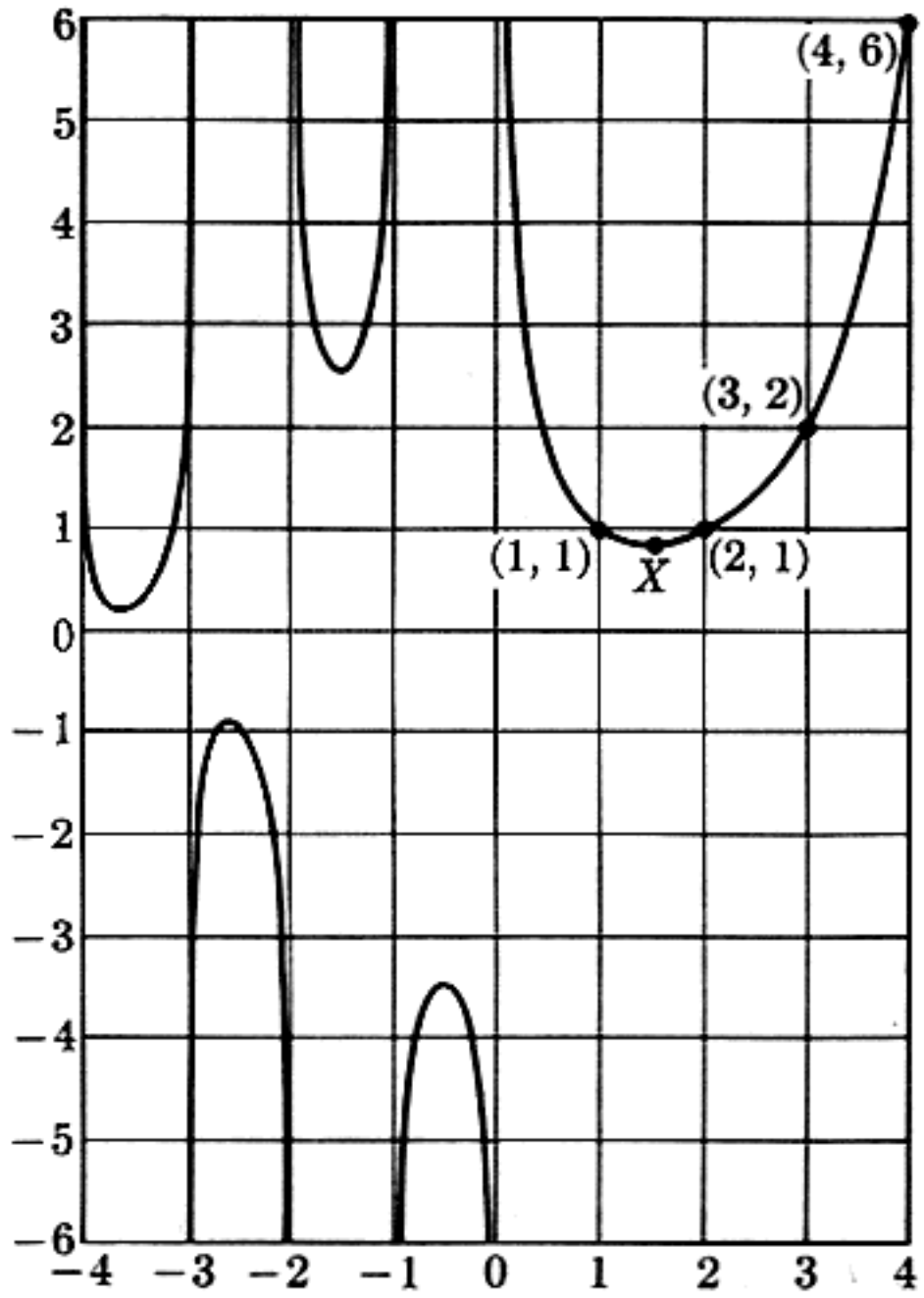


Fig. 7. The function $\Gamma(x) = (x - 1)!$. Point X has the coordinates (1.4616321450, 0.8856031944).

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

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

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

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

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

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

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

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

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

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

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

EXERCISES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.6. Binomial Coefficients

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

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

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

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

For example,

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

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

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

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

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

For particular cases we have

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

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

Table 1

TABLE OF BINOMIAL COEFFICIENTS (PASCAL'S TRIANGLE)

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

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

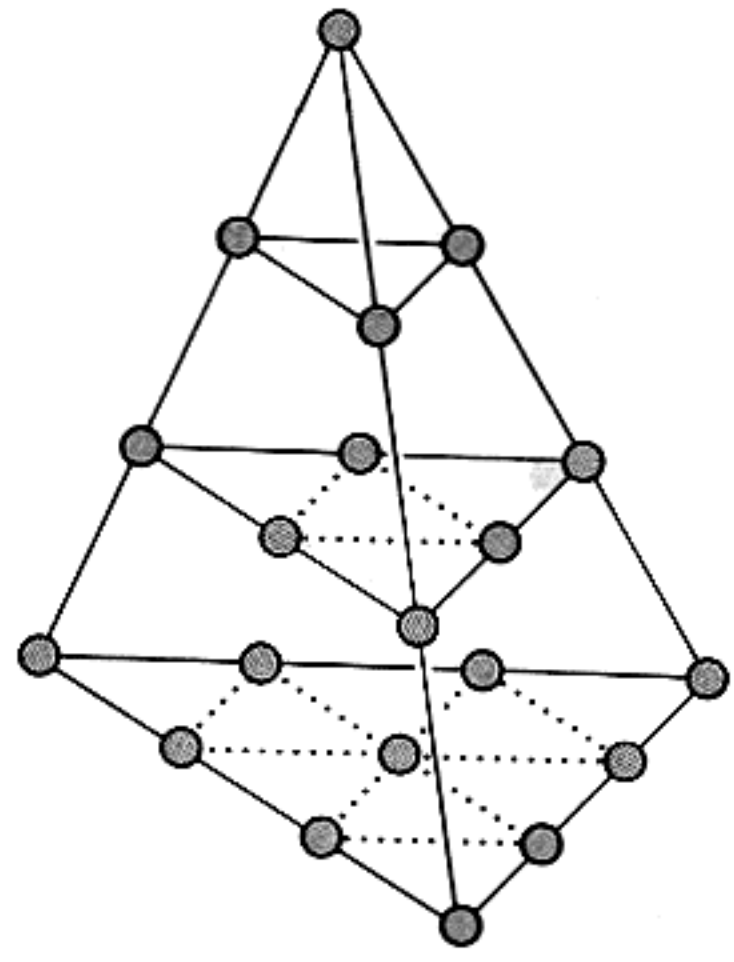


Fig. 8. Geometric interpretation of $\binom{n+2}{3}$, $n = 4$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D. Addition formula. The basic relation

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

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

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

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

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

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

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

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

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

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

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

$$\sum_{0\le k\le n}\binom{k}{m}=\sum_{-m\le k\le n-m}\binom{m+k}{m}=\sum_{-m\le k<0}\binom{m+k}{m}+\sum_{0\le k\le n-m}\binom{m+k}{k}$$
$$=0+\binom{m+(n-m)+1}{n-m}=\binom{n+1}{m+1},$$

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

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

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

our old friend, the sum of an arithmetic progression.

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

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

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

$$1^2+2^2+\cdots+n^2=2\frac{(n+1)n(n-1)}{6}+\frac{(n+1)n}{2}$$
$$=\tfrac{1}{3}n(n+\tfrac{1}{2})(n+1). \tag{12}$$

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

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

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

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

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

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

It should be noted that formula (13) gives

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

and we will use this convention consistently.

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

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

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

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

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

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

G. Negating the upper index. The basic identity

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

running variable k might appear:

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

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

$$\sum_k \binom{r}{k}\binom{s+k}{n}(-1)^k = (-1)^r \binom{s}{n-r},$$
$$\text{integer } n, \quad \text{integer } r \geq 0. \tag{23}$$

$$\sum_{0\leq k\leq r} \binom{r-k}{m}\binom{s}{k-t}(-1)^k = (-1)^t \binom{r-t-s}{r-t-m},$$
$$\text{integer } t \geq 0, \quad \text{integer } r \geq 0,$$
$$\text{integer } m \geq 0. \tag{24}$$

$$\sum_{0\leq k\leq r} \binom{r-k}{m}\binom{s+k}{n} = \binom{r+s+1}{m+n+1},$$
$$\text{integer } n \geq \text{integer } s \geq 0,$$
$$\text{integer } m \geq 0, \quad \text{integer } r \geq 0. \tag{25}$$

$$\sum_{k\geq 0} \binom{r-tk}{k}\binom{s-t(n-k)}{n-k}\frac{r}{r-tk} = \binom{r+s-tn}{n},$$
$$\text{integer } n,$$
$$r \neq tk \quad \text{for} \quad 0 \leq k \leq n. \tag{26}$$

Of these identities, Eq. (21) is by far the most important, and it should be memorized. Obviously Eq. (21) is a special case of Eq. (26) with $t = 0$. These formulas are the principal tools we have for working with difficult sums.

We should point out a nonobvious use of Eqs. (23) and (25); it is often helpful to replace the simple binomial coefficient on the right-hand side by the more complicated expression on the left, interchange the order of summation, and simplify. We may regard the left-hand sides as expansions of

$$\binom{s}{n+a} \quad \text{in terms of} \quad \binom{s+k}{n}.$$

Formula (23) is used for negative a, formula (25) for positive a.

This completes our study of "binomial-coefficientology." The reader is advised to learn especially Eqs. (5), (6), (7), (9), (13), (17), (20), and (21)—frame them in black!

With all these methods at our disposal, we should be able to solve "almost any" problem that comes along, in at least three different ways. The following examples illustrate the techniques.

Problem 1. When r is a positive integer, what is the value of

$$\sum_k \binom{r}{k}\binom{s}{k} k?$$

Solution. Formula (7) is useful for disposing of the outside k:

$$\sum_k \binom{r}{k}\binom{s}{k} k = \sum_k \binom{r}{k}\binom{s-1}{k-1} s$$

$$= s \sum_k \binom{r}{k}\binom{s-1}{k-1},$$

and now formula (22) applies, with $n = -1$. The answer is therefore

$$\sum_k \binom{r}{k}\binom{s}{k} k = \binom{r+s-1}{r-1} s, \qquad \text{integer } r \geq 0.$$

Problem 2. What is the value of

$$\sum_{k\geq 0} \binom{n+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1},$$

if $n \geq 0$?

Solution. Now the problem is tougher; the summation index k appears in six places! First we apply Eq. (20), and we obtain

$$\sum_{k\geq 0} \binom{n+k}{k}\binom{n}{k}\frac{(-1)^k}{k+1}.$$

We can now breathe more easily, since several of the menacing characteristics of the original formula have now disappeared. The next step should be obvious; we apply Eq. (7) in a manner similar to the technique used in Problem 1:

$$\sum_{k\geq 0} \binom{n+k}{k}\binom{n+1}{k+1}\frac{(-1)^k}{n+1}, \tag{27}$$

and another k has disappeared. There are now two equally promising lines of attack. We can replace

$$\binom{n+k}{k} \quad \text{by} \quad \binom{n+k}{n},$$

and use Eq. (23):

$$\sum_{k\geq 0}\binom{n+k}{n}\binom{n+1}{k+1}\frac{(-1)^k}{n+1}$$

$$= -\frac{1}{n+1}\sum_{k\geq 1}\binom{n-1+k}{n}\binom{n+1}{k}(-1)^k$$

$$= -\frac{1}{n+1}\sum_{k\geq 0}\binom{n-1+k}{n}\binom{n+1}{k}(-1)^k + \frac{1}{n+1}\binom{n-1}{n}$$

$$= -\frac{1}{n+1}(-1)^{n+1}\binom{n-1}{-1} + \frac{1}{n+1}\binom{n-1}{n} = \frac{1}{n+1}\binom{n-1}{n}.$$

Now

$$\binom{n-1}{n}$$

equals zero except when $n = 0$, in which case it equals one.

It is convenient to represent the answer to our problem by using the "Kronecker delta" notation:

$$\delta_{ij} = \begin{cases} 0, & \text{if} \quad i \neq j \\ 1, & \text{if} \quad i = j \end{cases}. \tag{28}$$

Using the δ-symbol, we have found that the answer is δ_{n0}.

Another way to proceed from Eq. (27) is to use Eq. (17), obtaining

$$\sum_k \binom{-(n+1)}{k}\binom{n+1}{k+1}\frac{1}{n+1}.$$

At this point Eq. (22) does not apply (since it requires that $r \geq 0$), but we can use Eq. (6) so that Eq. (21) applies:

$$\sum_k \binom{-(n+1)}{k}\binom{n+1}{n-k}\frac{1}{n+1} = \binom{0}{n}\frac{1}{n+1},$$

and once again we have derived the answer:

$$\sum_{k\geq 0}\binom{n+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1} = \delta_{n0}, \qquad \text{integer } n \geq 0. \tag{29}$$

Problem 3. What is the value of

$$\sum_{k\geq 0}\binom{n+k}{m+2k}\binom{2k}{k}\frac{(-1)^k}{k+1}, \qquad \text{if} \qquad m, n > 0?$$

Solution. If m were zero, we would have the same formula to work with that we had in Problem 2. However, now the presence of m means that we cannot even begin to use the method of the previous solution, since the first step there was to use Eq. (20), which no longer applies. In this situation it pays to introduce *still further* complication by replacing

$$\binom{n+k}{m+2k}$$

by a sum of terms of the form

$$\binom{x+k}{2k},$$

since our problem then becomes a sum of problems we know how to solve! Accordingly, we use Eq. (25) with

$$r = n + k - 1, \qquad m = 2k, \qquad s = 0, \qquad n = m - 1,$$

and we have

$$\sum_{k\geq 0} \sum_{0\leq j\leq n+k-1} \binom{n+k-1-j}{2k}\binom{2k}{k}\binom{j}{m-1}\frac{(-1)^k}{k+1}. \tag{30}$$

We wish to perform the summation on k first; interchanging the order of summation demands that we sum on the values of k which are ≥ 0 *and* $\geq j - n + 1$. The latter condition raises problems, because if $j \geq n$, we do *not* know the desired sum. Let us save the situation, however, by observing that the terms of (30) are zero when $n \leq j \leq n + k - 1$. This condition implies that $k \geq 1$; thus $0 \leq n + k - 1 - j \leq k - 1 < 2k$, and the first binomial coefficient in (30) vanishes. We may therefore replace the condition on the second sum by "$0 \leq j < n$," and the interchange of summation is done easily. Summing on k by Eq. (29) now gives

$$\sum_{0\leq j<n} \binom{j}{m-1} \delta_{(n-1-j)0},$$

and all terms vanish except $j = n - 1$; hence our final answer is

$$\binom{n-1}{m-1}.$$

The solution to this problem was fairly complicated, but not really mysterious; there was a good reason for each step. The derivation should be studied closely because it illustrates some delicate maneuvering with the conditions in our equations. There is actually a better way to attack this problem, however; it is left for the reader to figure out a way to transform the given problem so that Eq. (26) applies (see exercise 30).

Problem 4. Prove that

$$\sum_k A_k(r, t)A_{n-k}(s, t) = A_n(r + s, t), \qquad \text{integer } n \geq 0, \tag{31}$$

where

$$A_n(x, t) = \binom{x - nt}{n} \frac{x}{x - nt}, \qquad (x \neq nt).$$

Solution. We must assume that the given sum is defined, that is, that $r \neq kt \neq s$ for $0 \leq k \leq n$. Our problem is to evaluate

$$\sum_k \binom{r - kt}{k}\binom{s - (n - k)t}{n - k} \frac{r}{r - kt} \frac{s}{s - (n - k)t},$$

which, if anything, looks much worse than our previous horrible problems! Note the strong similarity to Eq. (26), however, and also note the case $t = 0$.

We are tempted to change

$$\binom{r - kt}{k} \frac{r}{r - kt} \quad \text{to} \quad \binom{r - kt - 1}{k - 1} \frac{r}{k},$$

except that the latter tends to lose the analogy with Eq. (26) and it fails when $k = 0$. The best way to proceed is to use the technique of "partial fractions," i.e., a complicated denominator can often be replaced by a sum of simpler denominators. Indeed, we have

$$\frac{1}{r - kt} \frac{1}{s - (n - k)t} = \frac{1}{r + s - nt}\left(\frac{1}{r - kt} + \frac{1}{s - (n - k)t}\right).$$

Putting this into our sum we get

$$\frac{s}{r + s - nt} \sum_k \binom{r - kt}{k}\binom{s - (n - k)t}{n - k} \frac{r}{r - kt}$$
$$+ \frac{r}{r + s - nt} \sum_k \binom{r - kt}{k}\binom{s - (n - k)t}{n - k} \frac{s}{s - (n - k)t},$$

and Eq. (26) evaluates both of these if we change k to $(n - k)$ in the second formula; the desired result follows immediately. Identities (26) and (31) are due to H. A. Rothe, *Formulae de serierum reversione* (Leipzig, 1793); special cases of these formulas are still being "discovered" frequently. For the interesting history of these identities and some generalizations, see H. W. Gould and J. Kaucký, *Journal of Combinatorial Theory* **1** (1966), 233–248.

Problem 5. Determine the values of $a_0, a_1, a_2, \ldots$ such that

$$n! = a_0 + a_1 n + a_2 n(n - 1) + a_3 n(n - 1)(n - 2) + \cdots \tag{32}$$

for all nonnegative integers n.

Solution. This question came up in the previous section (cf. Eq. 1.2.5–11) and we stated the answer without proof. Let us pretend we do not know the answer. It is clear that the problem *has* a solution, since we can set $n = 0$ and determine a_0, then set $n = 1$ and determine a_1, etc.

First we would like to write Eq. (32) in terms of binomial coefficients:

$$n! = \sum_k \binom{n}{k} k! a_k. \tag{33}$$

The problem of solving implicit equations like this for a_k is called the *inversion problem,* and the technique to be used applies to similar problems as well.

The idea is based on the following special case of Eq. (23) ($s = 0$):

$$\sum_k \binom{r}{k}\binom{k}{n} (-1)^k = (-1)^r \binom{0}{n-r} = (-1)^r \delta_{nr}, \qquad \text{integer } n, \quad \text{integer } r \geq 0. \tag{34}$$

The importance of this formula is that when $n \neq r$, the sum is zero; this enables us to solve our problem (much like we did in Problem 3) as follows:

$$\begin{aligned}\sum_n n! \binom{m}{n} (-1)^n &= \sum_n \sum_k \binom{n}{k} k! a_k \binom{m}{n} (-1)^n \\ &= \sum_k k! a_k \sum_n \binom{n}{k}\binom{m}{n} (-1)^n \\ &= \sum_k k! a_k (-1)^m \delta_{km} = (-1)^m m! a_m.\end{aligned}$$

Note how we were able to get an equation in which only one value a_m appears—by adding together suitable multiples of Eq. (33) for $n = 0, 1, 2, \ldots$. We have now

$$a_m = \sum_{n \geq 0} (-1)^{m+n} \frac{n!}{m!} \binom{m}{n} = \sum_{0 \leq n \leq m} \frac{(-1)^{m+n}}{(m-n)!} = \sum_{0 \leq n \leq m} \frac{(-1)^n}{n!}.$$

This completes the solution to Problem 5. Let us now take a closer look at the implications of Eq. (34): we have

$$\sum_k \binom{r}{k} (-1)^k \left(c_0 \binom{k}{0} + c_1 \binom{k}{1} + \cdots + c_r \binom{k}{r} \right) = (-1)^r c_r,$$

since the first terms vanish after summation. By properly choosing the coefficients c_i, we can represent *any* polynomial in k as a sum of binomial coefficients with upper index k. We therefore find that

$$\sum_k \binom{r}{k} (-1)^k (b_0 + b_1 k + \cdots + b_r k^r) = (-1)^r r! b_r, \qquad \text{integer } r \geq 0, \tag{35}$$

where $b_0 + \cdots + b_r k^r$ represents any polynomial whatever of degree r or less. [This formula will be of no great surprise to students of numerical analysis, since $\sum_k \binom{r}{k}(-1)^{r+k} f(x+k)$ is the "rth difference" of the function $f(x)$.]

Using Eq. (35), we can immediately obtain many other relations which appear complicated at first and which are often given very lengthy proofs, e.g.,

$$\sum_k \binom{r}{k}\binom{s-kt}{r}(-1)^k = t^r, \qquad \text{integer } r \geq 0. \tag{36}$$

It is customary in textbooks such as this to give a lot of impressive examples of neat tricks, etc., but to never mention simple-looking problems where the techniques fail. The above examples may have given the impression that all things are possible with binomial coefficients; it should be mentioned, however, that in spite of Eqs. (10), (11), and (18), there seems to be no simple formula for the analogous sum

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

when $n < m$. (For $n = m$ the answer is simple; what is it? See exercise 36.)

There are several generalizations of the concept of binomial coefficients, which we will discuss briefly. First, we can consider arbitrary real values of the lower index k in $\binom{r}{k}$; see exercises 40 through 45. We also have the generalization

$$\binom{r}{k}_q = \frac{(1-q^r)(1-q^{r-1})\ldots(1-q^{r-k+1})}{(1-q^k)(1-q^{k-1})\ldots(1-q^1)}, \tag{37}$$

which, as q approaches the limiting value one, becomes the ordinary binomial coefficient $\binom{r}{k}_1 = \binom{r}{k}$. [This can be seen by dividing each term in numerator and denominator by $(1-q)$.]

However, for our purposes the most important generalization is the *multinomial coefficient*

$$\binom{k_1 + k_2 + \cdots + k_m}{k_1, k_2, \ldots, k_m} = \frac{(k_1 + k_2 + \cdots + k_m)!}{k_1! k_2! \cdots k_m!}, \qquad \text{integer } k_i \geq 0. \tag{38}$$

The principal property of multinomial coefficients is the generalization of Eq. (13):

$$(x_1 + x_2 + \cdots + x_m)^n = \sum_{k_1+k_2+\cdots+k_m=n} \binom{n}{k_1, k_2, \ldots, k_m} x_1^{k_1} x_2^{k_2} \cdots x_m^{k_m}. \tag{39}$$

It is important to observe that any multinomial coefficient can be expressed in terms of binomial coefficients:

$$\binom{k_1+k_2+\cdots+k_m}{k_1, k_2, \ldots, k_m} = \binom{k_1+k_2}{k_1}\binom{k_1+k_2+k_3}{k_1+k_2}\cdots\binom{k_1+k_2+\cdots+k_m}{k_1+\cdots+k_{m-1}},$$

so we may apply the techniques we already know for manipulating binomial coefficients.

We conclude this section with a brief analysis of the transformation from a polynomial 'expressed in powers of k to a polynomial expressed in binomial coefficients. The coefficients involved in this transformation are called *Stirling numbers*, and these numbers will arise several times in later sections of this book.

Stirling numbers come in two flavors: we denote Stirling numbers of the first kind by $\left[{n \atop k}\right]$, and those of the second kind by $\left\{{n \atop k}\right\}$. Table 2 displays "Stirling's triangles," which are in some ways analogous to Pascal's triangle.

There is absolutely no agreement today on notation for Stirling's numbers. Some authors define half of the Stirling numbers to be the negatives of the values given here. However, the notation used here, in which all Stirling numbers are nonnegative, makes it much easier to remember the analogies with binomial coefficients.

Stirling numbers of the first kind are used to convert from binomial coefficients to powers:

$$\begin{aligned} n!\binom{x}{n} &= x(x-1)\cdots(x-n+1) \\ &= \begin{bmatrix} n \\ n \end{bmatrix} x^n - \begin{bmatrix} n \\ n-1 \end{bmatrix} x^{n-1} + \cdots + (-1)^n \begin{bmatrix} n \\ 0 \end{bmatrix} \\ &= \sum_k (-1)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k. \end{aligned} \tag{40}$$

For example, from Table 2,

$$\binom{x}{5} = \tfrac{1}{120}(x^5 - 10x^4 + 35x^3 - 50x^2 + 24x).$$

Stirling numbers of the second kind are used to convert from powers to binomial coefficients:

$$x^n = \left\{{n \atop n}\right\}\binom{x}{n} n! + \cdots + \left\{{n \atop 1}\right\}\binom{x}{1} 1! + \left\{{n \atop 0}\right\}\binom{x}{0} 0! = \sum_k \left\{{n \atop k}\right\}\binom{x}{k} k!. \tag{41}$$

For example, from Table 2,

$$\begin{aligned} x^5 &= \binom{x}{5} 5! + 10\binom{x}{4} 4! + 25\binom{x}{3} 3! + 15\binom{x}{2} 2! + \binom{x}{1} 1! \\ &= 120\binom{x}{5} + 240\binom{x}{4} + 150\binom{x}{3} + 30\binom{x}{2} + \binom{x}{1}. \end{aligned}$$

We shall now list the most important identities involving Stirling numbers. (In these equations, the variables m and n always denote nonnegative integers.)

Table 2

STIRLING NUMBERS OF THE FIRST AND SECOND KINDS*

$\left[{n \atop 0}\right]$	$\left[{n \atop 1}\right]$	$\left[{n \atop 2}\right]$	$\left[{n \atop 3}\right]$	$\left[{n \atop 4}\right]$	$\left[{n \atop 5}\right]$	$\left[{n \atop 6}\right]$	$\left[{n \atop 7}\right]$	$\left[{n \atop 8}\right]$	n	$\left\{{n \atop 0}\right\}$	$\left\{{n \atop 1}\right\}$	$\left\{{n \atop 2}\right\}$	$\left\{{n \atop 3}\right\}$	$\left\{{n \atop 4}\right\}$	$\left\{{n \atop 5}\right\}$	$\left\{{n \atop 6}\right\}$	$\left\{{n \atop 7}\right\}$	$\left\{{n \atop 8}\right\}$
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	2	0	1	1	0	0	0	0	0	0
0	2	3	1	0	0	0	0	0	3	0	1	3	1	0	0	0	0	0
0	6	11	6	1	0	0	0	0	4	0	1	7	6	1	0	0	0	0
0	24	50	35	10	1	0	0	0	5	0	1	15	25	10	1	0	0	0
0	120	274	225	85	15	1	0	0	6	0	1	31	90	65	15	1	0	0
0	720	1764	1624	735	175	21	1	0	7	0	1	63	301	350	140	21	1	0
0	5040	13068	13132	6769	1960	322	28	1	8	0	1	127	966	1701	1050	266	28	1

* For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun, U.S. Government Printing Office, 1964, Tables 24.3 and 24.4, where $\left[{n \atop m}\right]$ is denoted by $(-1)^{n+m}S_n^{(m)}$ and $\left\{{n \atop m}\right\}$ is denoted by $\mathfrak{S}_n^{(m)}$.

Table 2

STIRLING NUMBERS OF THE FIRST AND SECOND KINDS*

$\left[{n \atop 0}\right]$	$\left[{n \atop 1}\right]$	$\left[{n \atop 2}\right]$	$\left[{n \atop 3}\right]$	$\left[{n \atop 4}\right]$	$\left[{n \atop 5}\right]$	$\left[{n \atop 6}\right]$	$\left[{n \atop 7}\right]$	$\left[{n \atop 8}\right]$	n	$\left\{{n \atop 0}\right\}$	$\left\{{n \atop 1}\right\}$	$\left\{{n \atop 2}\right\}$	$\left\{{n \atop 3}\right\}$	$\left\{{n \atop 4}\right\}$	$\left\{{n \atop 5}\right\}$	$\left\{{n \atop 6}\right\}$	$\left\{{n \atop 7}\right\}$	$\left\{{n \atop 8}\right\}$
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	2	0	1	1	0	0	0	0	0	0
0	2	3	1	0	0	0	0	0	3	0	1	3	1	0	0	0	0	0
0	6	11	6	1	0	0	0	0	4	0	1	7	6	1	0	0	0	0
0	24	50	35	10	1	0	0	0	5	0	1	15	25	10	1	0	0	0
0	120	274	225	85	15	1	0	0	6	0	1	31	90	65	15	1	0	0
0	720	1764	1624	735	175	21	1	0	7	0	1	63	301	350	140	21	1	0
0	5040	13068	13132	6769	1960	322	28	1	8	0	1	127	966	1701	1050	266	28	1

* For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun, U.S. Government Printing Office, 1964, Tables 24.3 and 24.4, where $\left[{n \atop m}\right]$ is denoted by $(-1)^{n+m}S_n^{(m)}$ and $\left\{{n \atop m}\right\}$ is denoted by $\mathfrak{S}_n^{(m)}$.

Addition formulas:

$$\begin{aligned} \begin{bmatrix} n \\ m \end{bmatrix} &= (n-1)\begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}, \\ \begin{Bmatrix} n \\ m \end{Bmatrix} &= m\begin{Bmatrix} n-1 \\ m \end{Bmatrix} + \begin{Bmatrix} n-1 \\ m-1 \end{Bmatrix}, \end{aligned} \qquad \text{if} \quad n > 0. \tag{42}$$

Inversion formulas (compare with Eq. 34):

$$\sum_k \begin{bmatrix} n \\ k \end{bmatrix}\begin{Bmatrix} k \\ m \end{Bmatrix}(-1)^k = (-1)^n\delta_{mn}, \qquad \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix}\begin{bmatrix} k \\ m \end{bmatrix}(-1)^k = (-1)^n\delta_{mn}. \tag{43}$$

Special values:

$$\binom{0}{n} = \begin{bmatrix} 0 \\ n \end{bmatrix} = \begin{Bmatrix} 0 \\ n \end{Bmatrix}, \qquad \binom{n}{n} = \begin{bmatrix} n \\ n \end{bmatrix} = \begin{Bmatrix} n \\ n \end{Bmatrix} = 1; \tag{44}$$

$$\begin{bmatrix} n \\ n-1 \end{bmatrix} = \begin{Bmatrix} n \\ n-1 \end{Bmatrix} = \binom{n}{2}; \tag{45}$$

$$\begin{aligned} &\begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{Bmatrix} n \\ 0 \end{Bmatrix} = 0, \qquad \begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)!, \qquad \begin{Bmatrix} n \\ 1 \end{Bmatrix} = 1, \\ &\begin{Bmatrix} n \\ 2 \end{Bmatrix} = 2^{n-1} - 1, \end{aligned} \qquad \text{if} \quad n > 0. \tag{46}$$

Expansion formulas:

$$\sum_k \begin{bmatrix} n \\ k \end{bmatrix}\binom{k}{m} = \begin{bmatrix} n+1 \\ m+1 \end{bmatrix}, \qquad \sum_k \begin{bmatrix} n+1 \\ k+1 \end{bmatrix}\binom{k}{m}(-1)^k = \begin{bmatrix} n \\ m \end{bmatrix}(-1)^m; \tag{47}$$

$$\sum_k \begin{Bmatrix} k \\ m \end{Bmatrix}\binom{n}{k} = \begin{Bmatrix} n+1 \\ m+1 \end{Bmatrix}, \qquad \sum_k \begin{Bmatrix} k+1 \\ m+1 \end{Bmatrix}\binom{n}{k}(-1)^k = \begin{Bmatrix} n \\ m \end{Bmatrix}(-1)^n; \tag{48}$$

$$\sum_k \binom{n}{k} k^m(-1)^k = (-1)^n n! \begin{Bmatrix} m \\ n \end{Bmatrix}; \tag{49}$$

$$\begin{aligned} &\sum_k \binom{m-n}{m+k}\binom{m+n}{n+k}\begin{Bmatrix} m+k \\ k \end{Bmatrix} = \begin{bmatrix} n \\ n-m \end{bmatrix}, \\ &\sum_k \binom{m-n}{m+k}\binom{m+n}{n+k}\begin{bmatrix} m+k \\ k \end{bmatrix} = \begin{Bmatrix} n \\ n-m \end{Bmatrix}, \end{aligned} \qquad \text{if} \quad n \geq m; \tag{50}$$

$$\sum_k \begin{Bmatrix} n+1 \\ k+1 \end{Bmatrix}\begin{bmatrix} k \\ m \end{bmatrix}(-1)^k = (-1)^m\binom{n}{m}; \tag{51}$$

$$\sum_{k \leq n} \begin{bmatrix} k \\ m \end{bmatrix}\frac{n!}{k!} = \begin{bmatrix} n+1 \\ m+1 \end{bmatrix}, \qquad \sum_{k \leq n} \begin{Bmatrix} k \\ m \end{Bmatrix}(m+1)^{n-k} = \begin{Bmatrix} n+1 \\ m+1 \end{Bmatrix}. \tag{52}$$

Some other fundamental Stirling number identities appear in exercises 1.2.6–61, 1.2.7–6, and in Eqs. (23), (26), (27), and (28) of Section 1.2.9. For further information on Stirling numbers, see Karoly (Charles) Jordan, *Calculus of Finite Differences* (New York: Chelsea, 1947), Chapter 4.

EXERCISES

1. [*00*] How many combinations of n things taken $n - 1$ at a time are possible?

2. [*00*] What is $\binom{0}{0}$?

3. [*00*] How many bridge hands are possible (i.e., 13 cards out of a 52-card deck)?

4. [*10*] Give the answer to Problem 3 as a product of prime numbers.

▶ 5. [*05*] Explain the fact that $11^4 = 14641$ in terms of Pascal's triangle.

▶ 6. [*10*] Pascal's triangle (Table 1) can be extended in all directions by use of the addition formula, Eq. (9). Find the three rows which go on *top* of Table 1 (i.e., for $n = -1, -2$, and -3).

7. [*12*] If n is a fixed positive integer, what value of k makes $\binom{n}{k}$ a maximum?

8. [*00*] What property of Pascal's triangle is reflected in the "symmetry condition," Eq. (6)?

9. [*01*] What is the value of $\binom{n}{n}$? (Consider all integers n.)

▶ 10. [*M25*] If p is prime, show that:

a) $\binom{n}{p} \equiv \left\lfloor \frac{n}{p} \right\rfloor$ (modulo p).

b) $\binom{p}{k} \equiv 0$ (modulo p), for $1 \le k \le p - 1$.

c) $\binom{p-1}{k} \equiv (-1)^k$ (modulo p), for $0 \le k \le p - 1$.

d) $\binom{p+1}{k} \equiv 0$ (modulo p), for $2 \le k \le p - 1$.

e) (E. Lucas, 1877)

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}\binom{n \bmod p}{k \bmod p} \qquad \text{(modulo } p\text{)}.$$

f) If the p-ary number system representations of n, k are

$$\begin{aligned} n &= a_r p^r + \cdots + a_1 p + a_0, \\ k &= b_r p^r + \cdots + b_1 p + b_0, \end{aligned} \qquad \text{then} \qquad \binom{n}{k} \equiv \binom{a_r}{b_r} \cdots \binom{a_1}{b_1}\binom{a_0}{b_0} \quad \text{(modulo } p\text{)}.$$

▶ 11. [*M20*] (E. Kummer, 1852.) Show that the highest power to which a prime p divides

$$\binom{a+b}{a}$$

is equal to the number of *carries* which occur when a is added to b in the p-ary number system. (Cf. exercise 1.2.5–12.)

12. [*M22*] Are there any positive integers n for which all the nonzero entries in the nth row of Pascal's triangle are *odd*? If so, find all such n.

13. [*M13*] Prove the summation formula Eq. (10).

14. [*M21*] Evaluate $\sum_{0\le k\le n} k^4$.

15. [*M15*] Prove the binomial formula, Eq. (13).

16. [*M15*] Given that n, k are positive integers, show that

$$(-1)^n\binom{-n}{k-1} = (-1)^k\binom{-k}{n-1}.$$

▸ **17.** [*M18*] Prove the basic identity, Eq. (21), from Eq. (13), using the idea that $(1+x)^{r+s} = (1+x)^r(1+x)^s$.

18. [*M15*] Prove Eq. (22) using Eqs. (21) and (6).

19. [*M18*] Prove Eq. (23) by induction.

20. [*M20*] Prove Eq. (24) by using Eqs. (21) and (19), then show that another use of Eq. (19) yields Eq. (25).

▸ **21.** [*M05*] Both sides of Eq. (25) are polynomials in s; why isn't that equation an identity in s?

22. [*M20*] Prove Eq. (26) for the special case $s = n - 1 - r + nt$.

23. [*M13*] Assuming that Eq. (26) holds for (r, s, t, n) and $(r, s-t, t, n-1)$, prove it for $(r, s+1, t, n)$.

24. [*M15*] Explain why the results of the previous two exercises combine to give a proof of Eq. (26).

25. [*HM30*] Let the notation $A_n(x, t)$ have the significance of Eq. (31). Let $z = x^{t+1} - x^t$, and assume that $r \neq kt$ for integer $k \geq 0$. Prove that $\sum_k A_k(r, t)z^k = x^r$, provided z is small enough. [*Note:* If $t = 0$, this result is essentially the binomial theorem, and this equation is an important generalization of the binomial theorem. The binomial theorem (Eq. 15) may be assumed in the proof.] *Hint:* Start with the identity

$$\sum_j (-1)^j\binom{k}{j}\binom{r-jt}{k}\frac{r}{r-jt} = \delta_{k0}.$$

26. [*HM25*] Using the assumptions of the previous exercise, prove that

$$\sum_k \binom{r-tk}{k} z^k = \frac{x^{r+1}}{(t+1)x - t}.$$

27. [*HM20*] Solve Problem 4 in the text by using the result of exercise 25; and prove Eq. (26) from the preceding two exercises.

28. [*M25*] Prove that

$$\sum_k \binom{r+tk}{k}\binom{s-tk}{n-k} = \sum_{k\geq 0}\binom{r+s-k}{n-k} t^k,$$

if n is a nonnegative integer.

29. [*M20*] Show that Eq. (35) is just a very special case of the general identity proved in exercise 1.2.3–33.

▶ **30.** [*M24*] Show that there is a better way to solve Problem 3 than the way used in the text, by manipulating the sum so that Eq. (26) applies.

▶ **31.** [*M20*] Evaluate

$$\sum_k \binom{m-r+s}{k}\binom{n+r-s}{n-k}\binom{r+k}{m+n}$$

in terms of r, s, m, and n, given that m and n are nonnegative integers. Begin by replacing

$$\binom{r+k}{m+n} \quad \text{by} \quad \sum_j \binom{r}{m+n-j}\binom{k}{j}.$$

32. [*M20*] Let the notation $x^{\overline{n}}$ stand for $x(x+1)\cdots(x+n-1)$. Show that $\sum_k \left[{n \atop k}\right] x^k = x^{\overline{n}}$.

33. [*M20*] Using the notation of the previous exercise, show that the binomial formula is valid when it involves the modified "powers" of x as well as ordinary powers; i.e., show that $(x+y)^{\overline{n}} = \sum_k \binom{n}{k} x^{\overline{k}} y^{\overline{n-k}}$.

34. [*M23*] (Torelli's sum.) In the light of the previous exercise show that Abel's generalization of the binomial formula is also true for modified "powers":

$$(x+y)^{\overline{n}} = \sum_k \binom{n}{k} x(x-kz+1)^{\overline{k-1}}(y+kz)^{\overline{n-k}}. \qquad \text{(Cf. Eq. 16.)}$$

35. [*M23*] Prove the addition formulas, Eq. (42), for Stirling numbers directly from the definitions, Eqs. (40) and (41).

36. [*M10*] What is the sum $\sum_k \binom{n}{k}$ of the numbers in each row of Pascal's triangle? What is the sum of these numbers with alternating signs, $\sum_k \binom{n}{k}(-1)^k$?

37. [*M10*] From the answers to the preceding exercise, deduce the value of the sum of every other entry in a row, $\binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \cdots$.

38. [*HM30*] (C. Ramus, 1834.) Generalizing the result of the preceding exercise, show that we have the following formula, given that $0 \le k < m$:

$$\binom{n}{k} + \binom{n}{m+k} + \binom{n}{2m+k} + \cdots = \frac{1}{m}\sum_{0\le j<m}\left(2\cos\frac{j\pi}{m}\right)^n \cos\frac{j(n-2k)\pi}{m}.$$

For example,

$$\binom{n}{1} + \binom{n}{4} + \binom{n}{7} + \cdots = \frac{1}{3}\left(2^n + 2\cos\frac{(n-2)\pi}{3}\right).$$

[*Hint:* Find the right combinations of these coefficients multiplied by mth roots of unity.]

39. [*M10*] What is the sum $\sum_k \left[{n \atop k}\right]$ of the numbers in each row of Stirling's first triangle? What is the sum of these numbers with alternating signs? (Cf. exercise 36.)

40. [*HM17*] The *Beta function* $B(x, y)$ is defined for positive real numbers x, y by the formula $B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$.

a) Show that $B(x, 1) = B(1, x) = 1/x$.
b) Show that $B(x+1, y) + B(x, y+1) = B(x, y)$.
c) Show that $B(x, y) = \big((x+y)\big)/yB(x, y+1)$.

41. [*HM22*] We showed a relation between the Gamma function and the Beta function in exercise 1.2.5–19, by showing that $\Gamma_m(x) = m^x B(x, m+1)$, if m is a positive integer.

a) Prove that

$$B(x, y) = \frac{\Gamma_m(y)m^x}{\Gamma_m(x+y)} B(x, y+m+1).$$

b) Show that

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}.$$

42. [*HM10*] Express the binomial coefficient $\binom{r}{k}$ in terms of the Beta function defined above. (This gives us a way to extend the definition to all real values of k.)

43. [*HM20*] Show that $B(\frac{1}{2}, \frac{1}{2}) = \pi$. (From exercise 41 we may now conclude that $\Gamma(\frac{1}{2}) = \sqrt{\pi}$.)

44. [*HM2*] Using the generalized binomial coefficient suggested in exercise 42, show that

$$\binom{r}{1/2} = 2^{2r+1} \Big/ \binom{2r}{r} \pi.$$

45. [*HM21*] Using the generalized binomial coefficient suggested in exercise 42, find $\lim_{r\to\infty} \binom{r}{k}/r^k$.

▸ **46.** [*M21*] Using Stirling's approximation (Eq. 1.2.5–7), find an approximate value of

$$\binom{x+y}{y},$$

assuming that both x and y are large. In particular, find the approximate size of $\binom{2n}{n}$ when n is large.

47. [*M21*] Given that k is an integer, show that

$$\binom{n}{k}\binom{n+\frac{1}{2}}{k} = \binom{2n+1}{k}\binom{2n+1-k}{k} \Big/ 4^k.$$

Give a simpler formula for the special case $n = -1$.

▸ **48.** [*M25*] Show that

$$\sum_{k\geq 0} \binom{n}{k} \frac{(-1)^k}{k+x} = \frac{n!}{x(x+1)\cdots(x+n)} = \frac{1}{x\binom{n+x}{n}},$$

if the denominators are not zero. [Note that this formula gives us the reciprocal of a binomial coefficient, as well as the partial fraction expansion of $1/x(x+1)\cdots(x+n)$.]

49. [*M20*] Show that the identity $(1+x)^r = (1-x^2)^r(1-x)^{-r}$ implies a relation on binomial coefficients.

50. [*M20*] Prove Abel's formula, Eq. (16), in the special case $x + y = 0$.

51. [*M21*] Prove Abel's formula, Eq. (16), by writing $y = (x+y) - x$, expanding the right-hand side in powers of $(x+y)$, and applying the result of the previous exercise.

52. [*HM50*] For what complex values of x, y, z, and r is Abel's generalized binomial formula, Eq. (16), valid?

53. [*M25*] (a) Prove the following identity by induction on m:

$$\sum_{0\le k\le m} \binom{r}{k}\binom{s}{n-k}\Big(nr - (r+s)k\Big) = (m+1)(n-m)\binom{r}{m+1}\binom{s}{n-m},$$

integer m, n.

(b) Making use of the important relations

$$\binom{-1/2}{n} = \frac{(-1)^n}{2^{2n}}\binom{2n}{n},\quad \binom{1/2}{n} = \frac{(-1)^{n-1}}{2^{2n}(2n-1)}\binom{2n}{n} = \frac{(-1)^{n-1}}{2^{2n-1}(2n-1)}\binom{2n-1}{n}$$

show that the following formula can be obtained as a special case of the identity in part (a):

$$\sum_{0\le k\le m}\binom{2k-1}{k}\binom{2n-2k}{n-k}\frac{-1}{2k-1} = \frac{n-m}{2n}\binom{2m}{m}\binom{2n-2m}{n-m} + \frac{1}{2}\binom{2n}{n}.$$

(This result is considerably more general than Eq. (26) in the case $r = -1$, $s = 0$, $t = -2$.)

54. [*HM21*] Consider Pascal's triangle (as shown in Table 1) as a matrix. What is the *inverse* of that matrix?

55. [*M21*] Considering each of Stirling's triangles (Table 2) as matrices, determine their inverses.

▸ **56.** [*20*] (The "binomial number system.") For each integer $n = 0, 1, 2, \ldots, 20$, find three integers a, b, c for which $n = \binom{a}{1} + \binom{b}{2} + \binom{c}{3}$ and $0 \le a < b < c$. Can you see how this can be continued for higher values of n?

▸ **57.** [*M22*] Show that the coefficient a_m in Stirling's attempt at generalizing the factorial function (Eq. 1.2.5–12) is

$$\frac{(-1)^m}{m!}\sum_{k\ge 1}(-1)^k\binom{m-1}{k-1}\ln k.$$

58. [*M21*] In the notation of Eq. (37), prove the "q-binomial theorem":

$$(1+x)(1+qx)\cdots(1+q^{n-1}x) = \sum_k \binom{n}{k}_q q^{k(k-1)/2}x^k.$$

59. [*M25*] A sequence of numbers A_{nk}, $n \ge 0$, $k \ge 0$, satisfies the relations $A_{n0} = 1$, $A_{0k} = \delta_{0k}$, $A_{nk} = A_{(n-1)k} + A_{n(k-1)} + \binom{n}{k}$. Find A_{nk}.

▸ **60.** [*24*] We have seen that $\binom{n}{k}$ is the number of combinations of n things, k at a time, i.e., the number of ways to choose k different things out of a set of n. The *combinations with repetitions* are similar to ordinary combinations, except we may choose each object

any number of times. Thus, the list (1) would be extended to include also *aaa*, *aab*, *aac*, *aad*, *aae*, *abb*, etc., if we were considering combinations with repetition. How many k-combinations of n objects are there, if repetition is allowed?

61. [*M25*] Evaluate the sum

$$\sum_k \begin{bmatrix} n+1 \\ k+1 \end{bmatrix} \begin{Bmatrix} k \\ m \end{Bmatrix} (-1)^k,$$

thereby obtaining a companion formula for Eq. (51).

62. [*M38*] The text gives formulas for sums involving a product of two binomial coefficients. Of the sums involving a product of three binomial coefficients, the following seems to be most useful:

$$\sum_k (-1)^k \binom{2l}{l+k}\binom{2m}{m+k}\binom{2n}{n+k} = \frac{(l+m+n)!(2l)!(2m)!(2n)!}{(l+m)!(m+n)!(n+l)!l!m!n!},$$

integer $l, m, n \geq 0$.

(Note that the sum includes positive and negative values of k.) Prove this identity.

63. [*50*] Develop computer programs for simplifying sums that involve binomial coefficients.

▸ **64.** [*M22*] Show that $\left\{ {n \atop m} \right\}$ is the number of ways to partition a set of n elements into m nonempty disjoint subsets. For example, the set $\{1, 2, 3, 4\}$ can be partitioned into two subsets in $\left\{ {4 \atop 2} \right\} = 7$ ways: $\{1, 2, 3\}\{4\}$; $\{1, 2, 4\}\{3\}$; $\{1, 3, 4\}\{2\}$; $\{2, 3, 4\}\{1\}$; $\{1, 2\}\{3, 4\}$; $\{1, 3\}\{2, 4\}$; $\{1, 4\}\{2, 3\}$. *Hint:* Use the fact that

$$\begin{Bmatrix} n \\ m \end{Bmatrix} = m \begin{Bmatrix} n-1 \\ m \end{Bmatrix} + \begin{Bmatrix} n-1 \\ m-1 \end{Bmatrix}.$$

Note that the result of this exercise provides us with a mnemonic device for remembering the difference between the "[ ]" and "{ }" notations for Stirling numbers, since "{ }" is commonly used also for sets. The other Stirling numbers $\left[{n \atop k} \right]$ also have a combinatorial interpretation: $\left[{n \atop k} \right]$ is the number of permutations on n letters having k "cycles"; see Section 1.3.3.

1.2.7. Harmonic Numbers

The following sum will be of great importance in our later work:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{1 \le k \le n} \frac{1}{k}, \qquad n \ge 0. \tag{1}$$

This sum does not occur very frequently in classical mathematics, and there is no standard notation for it; but in the analysis of algorithms it pops up nearly every time we turn around, and we will consistently use the symbol H_n to represent the above quantity. (Besides H_n, the notations h_n and S_n are occasionally used in mathematical literature. The letter H stands for "harmonic," and we call H_n a *harmonic number* because (1) is customarily called the harmonic series.)

It may seem at first that H_n does not get too large when n has a large value, since we are always adding smaller and smaller numbers. But actually it is not hard to see that H_n will get as large as we please if we take n to be big enough, because of the following rule:

$$H_{2^m} \geq 1 + \frac{m}{2}. \tag{2}$$

This rule may be proved by observing that, for $m > 0$,

$$\begin{aligned} H_{2^{m+1}} &= H_{2^m} + \frac{1}{2^m+1} + \frac{1}{2^m+2} + \cdots + \frac{1}{2^{m+1}} \\ &> H_{2^m} + \frac{1}{2^{m+1}} + \frac{1}{2^{m+1}} + \cdots + \frac{1}{2^{m+1}} \\ &= H_{2^m} + \tfrac{1}{2}. \end{aligned}$$

So as m increases by 1, the left-hand side of Eq. (2) increases by at least $\frac{1}{2}$.

It is important to have more detailed information about the value of H_n than is given in Eq. (2). The approximate size of H_n is a well-known quantity (at least in mathematical circles) which may be expressed as follows:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon, \qquad 0 < \epsilon < \frac{1}{252n^6}. \tag{3}$$

Here $\gamma = 0.57721\ 56649\ldots$ is *Euler's constant.* Exact values of H_n for small n, and a 40-place value for γ, are given in the tables in Appendix B. We shall prove Eq. (3) in Section 1.2.11.2.

Thus H_n is reasonably close to the natural logarithm of n. Exercise 7 shows that H_n has a somewhat logarithmic behavior.

In a sense, H_n "just barely" goes to infinity as n gets large, because it can be proved that the sum

$$1 + \frac{1}{2^r} + \frac{1}{3^r} + \cdots + \frac{1}{n^r} \tag{4}$$

stays bounded for all n, when r is any real-valued exponent *greater* than unity. (See exercise 3.) We denote the sum in Eq. (4) by $H_n^{(r)}$.

When r in Eq. (4) is two or more, the value of $H_n^{(r)}$ is fairly close to its maximum value $H_\infty^{(r)}$, except for very small n. The quantity $H_\infty^{(r)}$ is very well known in mathematics as Riemann's "zeta function":

$$H_\infty^{(r)} = \zeta(r). \tag{5}$$

When r is an *even integer*, the value of $\zeta(r)$ is known to be equal to

$$H_\infty^{(r)} = \frac{1}{2}|B_r| \frac{(2\pi)^r}{r!}, \tag{6}$$

where B_r is a Bernoulli number (see Section 1.2.11.2 and Appendix B). In

particular,

$$H_\infty^{(2)} = \frac{\pi^2}{6}, \qquad H_\infty^{(4)} = \frac{\pi^4}{90}, \qquad H_\infty^{(6)} = \frac{\pi^6}{945}, \qquad H_\infty^{(8)} = \frac{\pi^8}{9450}. \tag{7}$$

For discussion and proof, see K. Knopp, *Theory and Application of Infinite Series*, tr. by R. C. H. Young (Glasgow: Blackie, 1951), Section 32.4.

Now we will consider a few important properties involving summations. First,

$$\sum_{1\le k\le n} H_k = (n+1)H_n - n. \tag{8}$$

This follows from simple transformation of sums:

$$\sum_{1\le k\le n} \sum_{1\le j\le k} \frac{1}{j} = \sum_{1\le j\le n} \sum_{j\le k\le n} \frac{1}{j} = \sum_{1\le j\le n} \frac{n+1-j}{j}.$$

Formula (8) is a special case of the sum $\sum_{1\le k\le n} \binom{k}{m} H_k$, which we will now determine. The "trick" to be used here is called summation by parts, and it is a useful technique for determining $\sum a_k b_k$ when the quantities $\sum a_k$ and $(b_{k+1} - b_k)$ have simple forms (see exercise 10). We observe in this case that

$$\binom{k}{m} = \binom{k+1}{m+1} - \binom{k}{m+1},$$

and therefore

$$\binom{k}{m} H_k = \binom{k+1}{m+1}\left(H_{k+1} - \frac{1}{k+1}\right) - \binom{k}{m+1} H_k;$$

hence

$$\begin{aligned}
&\sum_{1\le k\le n} \binom{k}{m} H_k \\
&\quad = \left(\binom{2}{m+1} H_2 - \binom{1}{m+1} H_1\right) + \cdots \\
&\qquad + \left(\binom{n+1}{m+1} H_{n+1} - \binom{n}{m+1} H_n\right) - \sum_{1\le k\le n} \binom{k+1}{m+1} \frac{1}{k+1} \\
&\quad = \binom{n+1}{m+1} H_{n+1} - \binom{1}{m+1} H_1 - \frac{1}{m+1} \sum_{0\le k\le n} \binom{k}{m} + \frac{1}{m+1}\binom{0}{m}.
\end{aligned}$$

Applying Eq. 1.2.6–11 yields the desired formula:

$$\sum_{1\le k\le n} \binom{k}{m} H_k = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right). \tag{9}$$

(The above derivation and final result are somewhat analogous to the determination of $\int_1^n x^m \ln x\, dx$ in integral calculus.)

We conclude this section by considering a different kind of sum, $\sum_k \binom{n}{k} x^k H_k$, which we will temporarily denote by S_n for brevity. We find that

$$S_{n+1} = \sum_k \left(\binom{n}{k} + \binom{n}{k-1}\right) x^k H_k = S_n + x \sum_k \binom{n}{k-1} x^{k-1} \left(H_{k-1} + \frac{1}{k}\right)$$

$$= S_n + x\, S_n + \frac{1}{n+1} \sum_{k \geq 1} \binom{n+1}{k} x^k.$$

Hence $S_{n+1} = (x+1)S_n + [(x+1)^{n+1} - 1]/(n+1)$, and we have

$$\frac{S_{n+1}}{(x+1)^{n+1}} = \frac{S_n}{(x+1)^n} + \frac{1}{n+1} - \frac{1}{(n+1)(x+1)^{n+1}}.$$

This equation, together with the fact that $S_1 = x$, shows us that

$$\frac{S_n}{(x+1)^n} = H_n - \sum_{1 \leq k \leq n} \frac{1}{k(x+1)^k}. \tag{10}$$

The remaining sum is part of the infinite series for $\ln\left(1/(1 - 1/(x+1))\right) = \ln(1 + 1/x)$, and when $x > 0$, the series is convergent; the difference is

$$\sum_{k>n} \frac{1}{k(x+1)^k} < \frac{1}{(n+1)(x+1)^{n+1}} \sum_{k \geq 0} \frac{1}{(x+1)^k} = \frac{1}{(n+1)(x+1)^n x}.$$

This proves the following theorem:

Theorem A. *If $x > 0$, then*

$$\sum_{1 \leq k \leq n} \binom{n}{k} x^k H_k = (x+1)^n \left(H_n - \ln\left(1 + \frac{1}{x}\right)\right) + \epsilon,$$

where $0 < \epsilon < 1/x(n+1)$. ∎

EXERCISES

1. [*01*] What are H_0, H_1, and H_2?

2. [*13*] Show that the simple argument used in the text to prove that $H_{2^m} \geq 1 + m/2$ can be slightly modified to prove that $H_{2^m} \leq 1 + m$.

3. [*M21*] Generalize the argument used in the previous exercise to show that $H_n^{(r)}$ remains bounded for all n, and find an upper bound, assuming that $r > 1$.

▶ 4. [*10*] Which of the following statements are true for all positive integers n? (a) $H_n < \ln n$. (b) $H_n > \ln n$. (c) $H_n > \ln n + \gamma$.

5. [*15*] Give the value of H_{10000} to 15 decimal places, using the tables in Appendix B.

6. [*M15*] Prove that the harmonic numbers are directly related to Stirling's numbers, which were introduced in the previous section; in fact,

$$H_n = \left[{n+1 \atop 2}\right] \Big/ n!.$$

7. [*M21*] Let $T(m, n) = H_m + H_n - H_{mn}$. (a) Show that if m or n increases, $T(m, n)$ decreases (assuming that m and n are positive). (b) Compute the minimum and maximum values of $T(m, n)$ for $m, n > 0$.

8. [*M18*] Compare Eq. (8) with $\sum_{1\le k\le n} \ln k$; estimate the difference as a function of n.

▶ **9.** [*M18*] Theorem A applies only when $x > 0$; what is the value of the sum considered when $x = -1$?

10. [*M20*] (Summation by parts.) We have used special cases of the general method of summation by parts in exercise 1.2.4–42 and in the derivation of Eq. (9). Prove the general formula

$$\sum_{1\le k<n} (a_{k+1} - a_k)b_k = a_n b_n - a_1 b_1 - \sum_{1\le k<n} a_{k+1}(b_{k+1} - b_k).$$

▶ **11.** [*M21*] Using summation by parts, evaluate

$$\sum_{1<k\le n} \frac{1}{k(k-1)} H_k.$$

▶ **12.** [*M10*] Evaluate $H_\infty^{(1000)}$ correct to at least 100 decimal places.

13. [*M22*] Prove the identity

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

(Note in particular the special case $x = 0$, which gives us an identity related to exercise 1.2.6–48.)

14. [*M22*] Show that

$$\sum_{1\le k\le n} \frac{H_k}{k} = \tfrac{1}{2}(H_n^2 + H_n^{(2)}),$$

and evaluate $\sum_{1\le k\le n} H_k/(k+1)$.

▶ **15.** [*M23*] Express $\sum_{1\le k\le n} H_k^2$ in terms of n and H_n.

16. [*18*] Express the sum $1 + \frac{1}{3} + \cdots + 1/(2n+1)$ in terms of harmonic numbers.

17. [*M24*] (E. Waring, 1782.) Let p be an odd prime. Show that the numerator of H_{p-1} is divisible by p.

18. [*M33*] (J. Selfridge.) What is the highest power of 2 which divides the numerator of $1 + \frac{1}{3} + \cdots + 1/(2n-1)$?

▶ **19.** [*M30*] List all nonnegative integers n for which H_n is an integer. [*Hint:* If H_n = odd/even, it cannot be an integer.]

20. [*HM22*] There is an analytic way to approach summation problems such as the one leading to Theorem A in this section: If $f(x) = \sum_{k\geq 0} a_k x^k$, and this series converges for $x = x_0$, then show that

$$\sum_{k\geq 0} a_k x_0^k H_k = \int_0^1 \frac{f(x_0) - f(x_0 y)}{1 - y}\, dy.$$

21. [*M24*] Evaluate $\sum_{1\leq k\leq n} H_k/(n + 1 - k)$.

22. [*M28*] Evaluate $\sum_{1\leq k\leq n} H_k H_{n+1-k}$.

▶ **23.** [*HM20*] By considering the function $\Gamma'(x)/\Gamma(x)$, show how we can get a natural generalization of H_n to noninteger values of n. You may use the fact that $\Gamma'(1) = -\gamma$, anticipating the next exercise.

24. [*HM21*] Show that

$$xe^{\gamma x} \prod_{k\geq 1} \left(\left(1 + \frac{x}{k}\right) e^{-x/k} \right) = \frac{1}{\Gamma(x)}.$$

(Consider the partial products of this infinite product.)

1.2.8. Fibonacci Numbers

The sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots, \tag{1}$$

in which each number is the sum of the preceding two, plays an important role in at least a dozen seemingly unrelated algorithms which we will study later. The numbers of the sequence are denoted by F_n, and we formally define this as

$$F_0 = 0, \qquad F_1 = 1, \qquad F_{n+2} = F_{n+1} + F_n, \quad n \geq 0. \tag{2}$$

This famous sequence was originated in 1202 by Leonardo Pisano (Leonardo of Pisa), who is sometimes called Leonardo Fibonacci (*Filius Bonaccii*, son of Bonaccio). His *Liber Abbaci* (Book of the Abacus) contains the following exercise: "How many pairs of rabbits can be produced from a single pair in a year's time?" To solve this problem, we are told to assume that each pair produces a new pair of offspring every month, each new pair becomes fertile at the age of one month, and furthermore, the rabbits never die. After one month there will be 2 pairs of rabbits; after two months, there will be 3; the following month the original pair and the pair born during the first month will both usher in a new pair and there will be 5 in all; and so on.

Fibonacci was by far the greatest European mathematician before the Renaissance. He studied the work of al-Khowârizmî (after whom "algorithm" is named, see Section 1.1) and he added numerous original contributions to

arithmetic and geometry. The writings of Fibonacci were reprinted in 1857 [B. Boncompagni, *Scritti di Leonardo Pisano* (Rome, 1857), 2 vols.; F_n appears in Vol. 1, pp. 283–285]. The rabbit problem was, of course, not posed as a practical application to biology and the population explosion; it was an exercise in addition. In fact, it still makes a rather good computer exercise in addition (cf. exercise 3); Fibonacci wrote: "It is possible to do [the addition] in this order for an infinite number of months."

The same sequence also appears in the work of Kepler, 1611, in connection with "phyllotaxis," the study of the arrangement of leaves and flowers in plant life. Kepler was presumably unaware of Fibonacci's brief mention of the sequence. Fibonacci numbers have often been observed in nature, probably for reasons similar to the original assumptions of the rabbit problem.

A first indication of the intimate connections between F_n and algorithms came to light in 1844, when G. Lamé used Fibonacci's sequence to study the efficiency of Euclid's algorithm. He proved that if the numbers m, n in algorithm 1.1E are not greater than F_k, step E2 will be executed at most $k + 1$ times. This was the first practical application of Fibonacci's sequence. During the next 50 years the mathematician E. Lucas obtained very profound results about the Fibonacci numbers, and in particular he used them to prove that the 39-digit number $2^{127} - 1$ is prime. Lucas gave the name "Fibonacci numbers" to the sequence F_n, and that name has been used ever since.

We already have examined the Fibonacci sequence briefly in Section 1.2.1 (Eq. (3) and exercise 4), where we found that $\phi^{n-2} \le F_n \le \phi^{n-1}$, if n is a positive integer and if

$$\phi = \tfrac{1}{2}(1 + \sqrt{5}). \tag{3}$$

We will see shortly that this quantity, ϕ, is intimately connected with the Fibonacci numbers.

The number ϕ itself has a very interesting history. Euclid called it the "extreme and mean ratio"; the ratio of A to B is the ratio of $(A + B)$ to A, if the ratio of A to B is ϕ. Renaissance writers called it the "divine proportion"; and in the last century it has commonly been called the "golden ratio." In the art world, the ratio of ϕ to 1 is said to be the most pleasing proportion aesthetically, and this opinion is confirmed from the standpoint of computer programming aesthetics as well. For the story of ϕ, see the excellent article "The Golden Section, Phyllotaxis, and Wythoff's Game," by H. S. M. Coxeter, *Scripta Math.* **19** (1953), 135–143, and see also Chapter 8 of *The 2nd Scientific American book of Mathematical Puzzles and Diversions*, by Martin Gardner (New York: Simon and Schuster, 1961).

The notations we are using in this section are a little undignified. In most of the sophisticated mathematical literature, F_n is called u_n instead, and ϕ is called τ. Our notations are almost universally used in recreational mathematics (and some crank literature!) and they are rapidly coming into wider use. The designation ϕ comes from the first letter of the Greek artist Phidias who is said to have

used the golden ratio frequently in his sculpture. The notation F_n is in accordance with that used in the *Fibonacci Quarterly* journal (published 1963–) where the reader may find numerous facts about the Fibonacci sequence. A good reference to the classical literature about Fibonacci's sequence is Chapter 17 of L. E. Dickson's *History of the Theory of Numbers*, Vol. 1 (New York: Chelsea, 1952).

The Fibonacci numbers satisfy many interesting identities, some of which appear in the exercises at the end of this section. One of the most commonly quoted relations is

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

which is easily proved by induction. A more esoteric method of proving the same formula starts with a simple inductive proof of the matrix identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}. \tag{5}$$

We then take the determinant of both sides of this equation.

Relation (4) shows that F_n and F_{n+1} are relatively prime, since any common divisor would have to be a divisor of $(-1)^n$.

From the definition (2) we find immediately that

$$F_{n+3} = F_{n+2} + F_{n+1} = 2F_{n+1} + F_n; \qquad F_{n+4} = 3F_{n+1} + 2F_n;$$

and, in general, by induction that

$$F_{n+m} = F_m F_{n+1} + F_{m-1} F_n \tag{6}$$

for any positive integer m.

If we take m to be a multiple of n in Eq. (6), we find inductively that

$$F_{nk} \quad \textit{is a multiple of} \quad F_k.$$

Thus every third number is even, every fourth number is a multiple of 3, every fifth is a multiple of 5, and so on.

In fact, much more than this is true. If we write gcd (m, n) to stand for the greatest common divisor of m and n, we have the rather surprising theorem:

Theorem A. (E. Lucas, 1876). *A number divides both F_m and F_n if and only if it is a divisor of F_d, where $d = \gcd(m, n)$; in particular,*

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}. \tag{7}$$

Proof: This result is proved by using Euclid's algorithm. We observe that because of Eq. (6) any common divisor of F_m and F_n is also a divisor of F_{n+m}; and, conversely, any common divisor of F_{n+m} and F_n is a divisor of $F_m F_{n+1}$. Since F_{n+1} is relatively prime to F_n, a common divisor of F_{n+m} and F_n also

divides F_m. Thus we have proved that for any number d

$$d \text{ divides } F_m \text{ and } F_n \text{ if and only if } d \text{ divides } F_{m+n} \text{ and } F_n. \tag{8}$$

We will now show that *any* sequence, F_n, for which statement (8)holds and for which $F_0 = 0$, satisfies Theorem A.

First it is clear that statement (8) may be extended by induction on k to the rule

$$d \text{ divides } F_m \text{ and } F_n \text{ if and only if } d \text{ divides } F_{m+kn} \text{ and } F_n,$$

where k is any nonnegative integer. This result may be stated more succinctly:

$$d \text{ divides } F_{(m \bmod n)} \text{ and } F_n \text{ if and only if } d \text{ divides } F_m \text{ and } F_n. \tag{9}$$

Now if r is the remainder after division of m by n, that is, if $r = m \bmod n$, then the common divisors of F_m, F_n are the common divisors of F_n, F_r. It follows that throughout the manipulations of Algorithm 1.1E the set of common divisors of F_m, F_n remains unchanged as m and n change; finally, when $r = 0$, the common divisors are simply the divisors of $F_0 = 0$ and $F_{\gcd(m,n)}$. ∎

Most of the important results involving Fibonacci numbers can be deduced from the representation of F_n in terms of ϕ, which we now proceed to derive. The method we shall use in the following derivation is extremely important, and the mathematically oriented reader should study it carefully; we will study the same method in detail in the next section.

We start by setting up the infinite series

$$\begin{aligned} G(z) &= F_0 + F_1 z + F_2 z^2 + F_3 z^3 + F_4 z^4 + \cdots \\ &= z + z^2 + 2z^3 + 3z^4 + \cdots \end{aligned} \tag{10}$$

We have no *a priori* reason to expect that this infinite sum exists or that the function $G(z)$ is at all interesting—but let us be optimistic and see what we can conclude about the function $G(z)$ if it does exist. The advantage of such a procedure is that $G(z)$ is a single quantity which represents the *entire* Fibonacci sequence at once; and if we find out that $G(z)$ is a "known" function, its coefficients can be determined. $G(z)$ is called the *generating function* for the sequence $\langle F_n \rangle$.

We can now proceed to investigate $G(z)$ as follows:

$$\begin{aligned} zG(z) &= F_0 z + F_1 z^2 + F_2 z^3 + F_3 z^4 + \cdots \\ z^2 G(z) &= \qquad\quad F_0 z^2 + F_1 z^3 + F_2 z^4 + \cdots ; \end{aligned}$$

by subtraction,

$$\begin{aligned} (1 - z - z^2)G(z) &= F_0 + (F_1 - F_0)z + (F_2 - F_1 - F_0)z^2 \\ &\quad + (F_3 - F_2 - F_1)z^3 + (F_4 - F_3 - F_2)z^4 + \cdots \\ &= z. \end{aligned}$$

All further terms are zero because of the definition of F_n; and so we see that, if $G(z)$ exists,

$$G(z) = z/(1 - z - z^2). \tag{11}$$

In fact, this function *can* be expanded in an infinite series in z (a Taylor series); working backwards we find the coefficients of the power series expansion of Eq. (11) must be the Fibonacci numbers.

We can now manipulate $G(z)$ and find out more about the Fibonacci sequence. The denominator $1 - z - z^2$ is a quadratic equation with the two roots $\frac{1}{2}(-1 \pm \sqrt{5})$; after a little calculation we find that $G(z)$ can be expanded by the method of partial fractions into the form

$$G(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right), \tag{12}$$

where

$$\hat{\phi} = 1 - \phi = \tfrac{1}{2}(1 - \sqrt{5}). \tag{13}$$

The quantity $1/(1 - \phi z)$ is the sum of the infinite geometric series $1 + \phi z + \phi^2 z^2 + \cdots$, so we have

$$G(z) = \frac{1}{\sqrt{5}}(1 + \phi z + \phi^2 z^2 + \cdots - 1 - \hat{\phi} z - \hat{\phi}^2 z^2 - \cdots).$$

We now look at the coefficient of z^n, *which must be equal to* F_n, and we find that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n). \tag{14}$$

This is an important "closed form" expression for the Fibonacci numbers, first discovered by A. de Moivre in 1718. The method we have used to derive Eq. (14) is due to Nicolas Bernoulli (1728).

We could have merely stated Eq. (14) and proved it by induction; the point of the rather long derivation above was to show how it would be possible to *discover* the equation in the first place, using the important method of generating functions, which is a valuable technique for solving so many problems.

Many things can be proved from Eq. (14). First we observe that $\hat{\phi}$ is a *negative* number ($-0.61803\ldots$) whose magnitude is less than unity, so $\hat{\phi}^n$ gets very small as n gets large. In fact, $\hat{\phi}^n/\sqrt{5}$ is always small enough so that we have

$$F_n = \phi^n/\sqrt{5} \qquad \textit{rounded to the nearest integer.} \tag{15}$$

Other results can be obtained directly from $G(z)$; for example,

$$G(z)^2 = \frac{1}{5}\left(\frac{1}{(1 - \phi z)^2} + \frac{1}{(1 - \hat{\phi} z)^2} - \frac{2}{1 - z - z^2}\right), \tag{16}$$

and the coefficient of z^n in $G(z)^2$ is $\sum_{0\le k\le n} F_k F_{n-k}$. We therefore deduce that

$$\begin{aligned}\sum_{0\le k\le n} F_k F_{n-k} &= \tfrac{1}{5}\big((n+1)(\phi^n+\hat{\phi}^n) - 2F_{n+1}\big)\\ &= \tfrac{1}{5}\big((n+1)(F_n+2F_{n-1}) - 2F_{n+1}\big)\\ &= \frac{n-1}{5}F_n + \frac{2n}{5}F_{n-1}.\end{aligned} \tag{17}$$

(The second step in this derivation follows from the result of exercise 11.)

EXERCISES

1. [*10*] In Leonardo Fibonacci's problem, how many pairs of rabbits are present after k months? What is the answer to his question, i.e., how many pairs are present after a year?

▶ **2.** [*20*] In lieu of Eq. (15), what is the approximate value of F_{1000}? (Use logarithms found in the table in Appendix B.)

3. [*34*] Write a program for some computer which calculates and prints F_1 through F_{1000}. (Cf. the previous exercise for the size of the numbers which must be handled.)

▶ **4.** [*14*] Find all n for which $F_n = n$.

5. [*20*] Find all n for which $F_n = n^2$.

6. [*HM10*] Prove Eq. (5).

▶ **7.** [*15*] If n is not a prime number, F_n is not a prime number (with one exception). Prove this and find the exception.

8. [*15*] In many cases it is convenient to define F_n for *negative* n, by assuming that $F_{n+2} = F_{n+1} + F_n$ for all integers n. Explore this possibility; what is F_{-1}? What is F_{-2}? Can F_{-n} be expressed in a simple way in terms of F_n?

9. [*M20*] Using the conventions of the preceding exercise, determine whether Eqs. (4), (6), (14), and (15) still hold when the subscripts are allowed to be *any* integer.

10. [*15*] Is $\phi^n/\sqrt{5}$ greater than F_n or less than F_n?

11. [*M20*] Show that $\phi^n = F_n\phi + F_{n-1}$, $\hat{\phi}^n = F_n\hat{\phi} + F_{n-1}$, for *all* integers n.

▶ **12.** [*M26*] The "second order" Fibonacci sequence is defined by the rule

$$\mathfrak{F}_0 = 0, \qquad \mathfrak{F}_1 = 1, \qquad \mathfrak{F}_{n+2} = \mathfrak{F}_{n+1} + \mathfrak{F}_n + F_n.$$

Express $\mathfrak{F}_n$ in terms of F_n. [*Hint:* Use generating functions.]

▶ **13.** [*M22*] Express the following sequences in terms of the Fibonacci numbers:

a) $a_0 = r, \quad a_1 = s, \quad a_{n+2} = a_{n+1} + a_n, \qquad n \geq 0.$
b) $b_0 = 0, \quad b_1 = 1, \quad b_{n+2} = b_{n+1} + b_n + c, \qquad n \geq 0.$

14. [*M28*] Let m be a fixed positive integer. Find a_n given that

$$a_0 = 0, a_1 = 1, a_{n+2} = a_{n+1} + a_n + \binom{n}{m}.$$

15. [*M22*] Let $f(n)$, $g(n)$ be arbitrary functions. Let

$$\begin{aligned} a_0 &= 0, \quad a_1 = 1, \quad a_{n+2} = a_{n+1} + a_n + f(n); \\ b_0 &= 0, \quad b_1 = 1, \quad b_{n+2} = b_{n+1} + b_n + g(n); \\ c_0 &= 0, \quad c_1 = 1, \quad c_{n+2} = c_{n+1} + c_n + xf(n) + yg(n). \end{aligned}$$

Express c_n in terms of x, y, a_n, b_n, and F_n.

▶ **16.** [*M20*] Fibonacci numbers appear implicitly in Pascal's triangle if it is viewed from the right angle. Show that the following sum of binomial coefficients is a Fibonacci number:

$$\sum_{0\le k\le n} \binom{n-k}{k}.$$

17. [*M24*] Using the conventions of exercise 8, prove the following generalization of Eq. (4): $F_{n+k}F_{m-k} - F_nF_m = (-1)^nF_{m-n-k}F_k$.

18. [*20*] Is $F_n^2 + F_{n+1}^2$ always a Fibonacci number?

19. [*M27*] What is cos 36°?

20. [*M16*] Express $\sum_{0\le k\le n} F_k$ in terms of Fibonacci numbers.

21. [*M25*] What is $\sum_{0\le k\le n} F_kx^k$?

▶ **22.** [*M20*] Show that $\sum_k \binom{n}{k}F_{m+k}$ is a Fibonacci number.

23. [*M23*] Generalizing the preceding exercise, show that $\sum_k\binom{n}{k}F_t^kF_{t-1}^{n-k}F_{m+k}$ is always a Fibonacci number.

24. [*HM20*] Evaluate the $n \times n$ determinant

$$\begin{pmatrix} 1 & -1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 & \ldots & 0 & 0 & 0 \\ \vdots & & & & & & & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 & 1 \end{pmatrix}.$$

25. [*M21*] Show that

$$2^nF_n = 2\sum_{k \text{ odd}} \binom{n}{k} 5^{(k-1)/2}.$$

▶ **26.** [*M20*] Using the previous exercise, show that $F_p \equiv 5^{(p-1)/2}$ (modulo p) if p is an odd prime.

27. [*M20*] Using the previous exercise, show that if p is a prime different from 5, then either F_{p-1} or F_{p+1} (not both) is a multiple of p.

28. [*M21*] What is $F_{n+1} - \phi F_n$?

▶ **29.** [*M23*] (The Fibonacci triangle.) Define

$$\left(\!\binom{n}{k}\!\right) = \frac{F_nF_{n-1}\cdots F_{n-k+1}}{F_kF_{k-1}\cdots F_1} = \prod_{1\le j\le k}\left(\frac{F_{n-k+j}}{F_j}\right)$$

in a manner analogous to binomial coefficients. (a) Make a table of $\left(\!\binom{n}{k}\!\right)$ for $0 \le n \le 6$. (b) Show that

$$\left(\!\binom{n}{k}\!\right) = F_{k-1}\left(\!\binom{n-1}{k}\!\right) + F_{n-k+1}\left(\!\binom{n-1}{k-1}\!\right).$$

▸ **30.** [*M38*] (D. Jarden.) The sequence of mth powers of Fibonacci numbers satisfies a recurrence relation in which each term depends on the preceding $m+1$ terms. Show that

$$\sum_k \left(\!\binom{m}{k}\!\right)(-1)^{\lfloor (m-k)/2 \rfloor} F_{n+k}^{m-1} = 0, \qquad \text{if} \qquad m > 0.$$

For example, when $m = 3$, we get the identity $F_n^2 - 2F_{n+1}^2 - 2F_{n+2}^2 + F_{n+3}^2 = 0$.

31. [*M20*] Let $\psi = \phi - 1 = 1/\phi$. Show that $(F_{2n}\psi) \bmod 1 = 1 - \psi^{2n}$ and $(F_{2n+1}\psi) \bmod 1 = \psi^{2n+1}$.

32. [*M24*] The remainder of one Fibonacci number divided by another is $\pm$ a Fibonacci number: Show that

$$F_{mn+r} \equiv F_r, \quad (-1)^{r+1}F_{n-r}, \quad (-1)^n F_r, \qquad \text{or} \qquad (-1)^{r+1+n}F_{n-r} \qquad (\text{modulo } F_n),$$

depending on whether $m \bmod 4 = 0, 1, 2$, or 3, respectively.

33. [*HM24*] Given that $z = \pi/2 + i \ln \phi$, show that $\sin(nz)/\sin z = i^{1-n}F_n$.

▸ **34.** [*M24*] (The Fibonacci number system.) Let the notation $k \gg m$ mean that $k \ge m + 2$. Show that every positive integer n has a *unique* representation $n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}$, where $k_1 \gg k_2 \gg \cdots \gg k_r \gg 0$.

35. [*M24*] (A phi number system.) Consider real numbers written with the digits 0 and 1 using base ϕ. (Thus $100.1 = \phi^2 + \phi^{-1}$.) Show that there are infinitely many ways to represent the number 1 (for example, $1 = .11 = .011111\ldots$); but if we require that no two adjacent 1's occur and that no infinite sequence "01010101 . . ." appears, every number has a unique representation.

▸ **36.** [*M32*] ("Fibonacci-string sequence.") Let $S_1 =$ "a", $S_2 =$ "b", and $S_{n+2} = S_{n+1}S_n$, $n > 0$; in other words, S_{n+2} is formed by placing S_n at the right of S_{n+1}. We have $S_3 =$ "ba", $S_4 =$ "bab", $S_5 =$ "$babba$", etc. Clearly S_n has F_n letters. Explore the properties of S_n. (Where do double letters occur? Can you predict the value of the kth letter of S_n? What is the density of the b's? And so on.)

▸ **37.** [*M35*] (R. E. Gaskell, M. J. Whinihan.) Two players compete in the following game: There is a pile containing n chips; the first player removes any number of chips except that he cannot take the whole pile. From then on, the players alternate moves, each person removing one or more chips but *not more than twice as many chips as the preceding player has taken.* The player who removes the last chip wins. (For example, suppose that $n = 11$; player A removes 3 chips; player B may remove up to 6 chips, and he takes 1. There remain 7 chips; player A may take 1 or 2 chips, and he takes 2; player B may remove up to 4, and he picks up 1. There remain 4 chips; player A now takes 1; player B must take at least one chip and player A wins in the following turn.)

What is the best move for the first player to make if there are initially 1000 chips?

38. [*35*] Write a computer program which plays the game described in the previous exercise and which plays optimally.

39. [*M24*] Find a closed form expression for a_n, given that $a_0 = 0$, $a_1 = 1$, $a_{n+2} = a_{n+1} + 6a_n$.

1.2.9. Generating Functions

Whenever we want to obtain information about a sequence of numbers $\langle a_n \rangle = a_0, a_1, a_2, \ldots$, we can set up an infinite sum in terms of a "parameter" z,

$$G(z) = a_0 + a_1 z + a_2 z^2 + \cdots = \sum_{n \geq 0} a_n z^n. \tag{1}$$

We can then try to obtain information about the function G. This function G is a single quantity which represents the whole sequence $\langle a_n \rangle$; if the sequence $\langle a_n \rangle$ has been defined inductively (that is, if a_n has been defined in terms of $a_0, a_1, \ldots, a_{n-1}$), this is an important advantage. Furthermore, we can recover the values of $a_0, a_1, \ldots$ from the function $G(z)$, assuming that the infinite sum in Eq. (1) exists for some values of z, by using techniques of differential calculus.

$G(z)$ is called the *generating function* for the sequence $a_0, a_1, a_2, \ldots$ The use of generating functions opens up a whole new range of techniques, and it broadly increases our capacity for problem solving. Starting in 1741, L. Euler used generating functions in several number-theoretic investigations. Pierre S. Laplace developed the techniques further in his classic work *Théorie Analytique des Probabilities* (1812). A similar concept, which has become known as the "Laplace transform," was actually introduced by another mathematician N. Abel [*Collected Works* (Oslo, 1881), Vol. 2, pp. 67–81]. (See exercise 13 for a connection between generating functions and Laplace transforms.) Part of the credit for discovering generating functions belongs also to N. Bernoulli (1728) and J. Stirling (1730); see C. Tweedie, *James Stirling*, (London: Oxford University Press, 1922), pp. 37–38, 95.

The question of convergence of the infinite sum, Eq. (1), is of some importance. Any textbook about the theory of infinite series will prove that:

a) If Eq. (1) exists ("converges") for a particular value of $z = z_0$, then it converges for all values of z with $|z| < z_0$.
b) The sequence converges for some $z \neq 0$ if and only if the sequence $\sqrt[n]{|a_n|}$ is bounded. (If this condition is not satisfied, it may be possible to get a convergent series for a related sequence, e.g., for the sequence $\langle a_n/n! \rangle$.)

On the other hand, it often does not pay to worry about convergence of the series when we work with generating functions, since we are only exploring possible approaches to the solution of some problem. When we discover the solution by *any* means, however sloppy they might be, it may be possible to justify the solution independently. For example, in the previous section we used a generating function to deduce Eq. (14); yet once this equation has been found,

it is a simple matter to prove it by induction, and we need not even mention that we used generating functions to discover that relation.

Let us now study the principal techniques used with generating functions.

A. Addition. If $G_1(z)$ is the generating function for $a_0, a_1, \ldots$ and $G_2(z)$ is the generating function for $b_0, b_1, \ldots$, then $\alpha G_1(z) + \beta G_2(z)$ is the generating function for $\alpha a_0 + \beta b_0, \alpha a_1 + \beta b_1, \ldots$:

$$\alpha \sum_{k\geq 0} a_k z^k + \beta \sum_{k\geq 0} b_k z^k = \sum_{k\geq 0} (\alpha a_k + \beta b_k) z^k. \tag{2}$$

B. Shifting. If $G(z)$ is the generating function for $a_0, a_1, \ldots$ then $z^n G(z)$ is the generating function for $0, \ldots, 0, a_0, a_1, \ldots$:

$$z^n \sum_{k\geq 0} a_k z^k = \sum_{k\geq n} a_{k-n} z^k. \tag{3}$$

The last summation may be extended over all $k \geq 0$ if we regard $a_k = 0$ for any negative value of k.

Similarly, $(G(z) - a_0 - a_1 z - \cdots - a_{n-1} z^{n-1})/z^n$ is the generating function for $a_n, a_{n+1}, \ldots$:

$$z^{-n} \sum_{k\geq n} a_k z^k = \sum_{k\geq 0} a_{k+n} z^k. \tag{4}$$

We combined operations A and B to solve the Fibonacci problem in the previous section; $G(z)$ was the generating function for $\langle F_n \rangle$, $zG(z)$ for $\langle F_{n-1} \rangle$, $z^2 G(z)$ for $\langle F_{n-2} \rangle$, and $(1 - z - z^2)G(z)$ for $\langle F_n - F_{n-1} - F_{n-2} \rangle$. The latter sequence is zero when $n \geq 2$, so $(1 - z - z^2)G(z)$ is a polynomial. The same technique applies to any "linearly recurrent" sequence where $a_n = c_1 a_{n-1} + \cdots + c_m a_{n-m}$; the generating function is a polynomial divided by $(1 - c_1 z - \cdots - c_m z^m)$.

Let us consider the simplest example of all: If $G(z)$ is the generating function for the *constant* sequence $1, 1, 1, \ldots$, then $zG(z)$ generates $0, 1, 1, \ldots$, so $(1 - z)G(z) = 1$. This gives us the very special case

$$\frac{1}{1 - z} = 1 + z + z^2 + \cdots. \tag{5}$$

C. Multiplication. If $G_1(z)$ is the generating function for $a_0, a_1, \ldots$ and $G_2(z)$ is the generating function for $b_0, b_1, \ldots$, then

$$\begin{aligned} G_1(z)G_2(z) &= (a_0 + a_1 z + a_2 z^2 + \cdots)(b_0 + b_1 z + b_2 z^2 + \cdots) \\ &= (a_0 b_0) + (a_0 b_1 + a_1 b_0)z + (a_0 b_2 + a_1 b_1 + a_2 b_0)z^2 + \cdots; \end{aligned}$$

thus $G_1(z)G_2(z)$ is the generating function for the sequence $S_0, S_1, \ldots$, where

$$S_n = \sum_{0\leq k\leq n} a_k b_{n-k}. \tag{6}$$

Equation (3) is a very special case of this. Another important special case occurs when each b_n is equal to unity:

$$\frac{1}{1-z}G(z) = a_0 + (a_0 + a_1)z + (a_0 + a_1 + a_2)z^2 + \cdots. \tag{7}$$

Here we have the generating function for the sums of the original sequence.

The rule for a product of *three* functions follows from Eq. (6); $G_1(z)G_2(z)G_3(z)$ generates $S_0, S_1, \ldots,$ where

$$S_n = \sum_{\substack{i,j,k\geq 0\\ i+j+k=n}} a_i b_j c_k. \tag{8}$$

The general rule for products of *any number* of functions (whenever this is meaningful) is

$$\prod_{j\geq 0}\left(\sum_{k\geq 0} a_{jk}z^k\right) = \sum_{n\geq 0} z^n \left(\sum_{\substack{k_0,k_1,\ldots\geq 0\\ k_0+k_1+\ldots=n}} a_{0k_0}a_{1k_1}\cdots\right). \tag{9}$$

When the recurrence relation for some sequence involves binomial coefficients, we often want to get a generating function for a sequence $c_0, c_1, \ldots$ defined by

$$c_n = \sum_k \binom{n}{k} a_k b_{n-k}. \tag{10}$$

In this case it is usually better to use generating functions for the sequences $\langle a_n/n!\rangle$, $\langle b_n/n!\rangle$, $\langle c_n/n!\rangle$, since we have

$$\left(\frac{a_0}{0!} + \frac{a_1}{1!}z + \frac{a_2}{2!}z^2 + \cdots\right)\left(\frac{b_0}{0!} + \frac{b_1}{1!}z + \frac{b_2}{2!}z^2 + \cdots\right) = \left(\frac{c_0}{0!} + \frac{c_1}{1!}z + \frac{c_2}{2!}z^2 + \cdots\right), \tag{11}$$

where c_n is given by Eq. (10).

D. Change of z. Clearly $G(cz)$ is the generating function for the sequence $a_0, ca_1, c^2a_2, \ldots$ In particular, the generating function for the sequence $1, c, c^2, c^3, \ldots$ is $1/(1 - cz)$.

There is a familiar trick for extracting alternate terms of a series:

$$\begin{aligned} \tfrac{1}{2}[G(z) + G(-z)] &= a_0 + a_2z^2 + a_4z^4 + \cdots \\ \tfrac{1}{2}[G(z) - G(-z)] &= a_1 + a_3z^3 + a_5z^5 + \cdots. \end{aligned} \tag{12}$$

Generally, using complex numbers, we can extract every mth term: Let $\omega = e^{2\pi i/m}$; we have

$$\sum_{k \bmod m = r} a_k z^k = \frac{1}{m}\sum_{1\leq j\leq m} \omega^{-jr}G(\omega^j z), \qquad 0 \leq r < m. \tag{13}$$

For example, if $m = 3$ and $r = 1$, we have $\omega = \cos 120° + i \sin 120°$ (a complex cube root of unity); it follows that

$$a_1 z + a_4 z^4 + a_7 z^7 + \cdots = \tfrac{1}{3}[G(z) + \omega^{-1} G(\omega z) + \omega^{-2} G(\omega^2 z)].$$

Proof is left to the reader (exercise 14).

E. Differentiation and integration. The techniques of calculus give us further operations. If $G(z)$ is given by Eq. (1), the derivative is

$$G'(z) = a_1 + 2a_2 z + 3a_3 z^2 + \cdots = \sum_{k \geq 0} (k+1) a_{k+1} z^k. \tag{14}$$

The generating function for the sequence $\langle n a_n \rangle$ is $zG'(z)$. Hence we can combine the nth term of a sequence with polynomials in n by manipulating the generating function.

Reversing the process, integration gives another useful operation:

$$\int_0^z G(t)\, dt = a_0 z + \tfrac{1}{2} a_1 z^2 + \tfrac{1}{3} a_2 z^3 + \cdots = \sum_{k \geq 1} \frac{1}{k} a_{k-1} z^k. \tag{15}$$

As special cases, we have the derivative and integral of (5):

$$\frac{1}{(1-z)^2} = 1 + 2z + 3z^2 + \cdots = \sum_{k \geq 0} (k+1) z^k. \tag{16}$$

$$\ln \frac{1}{1-z} = z + \tfrac{1}{2} z^2 + \tfrac{1}{3} z^3 + \cdots = \sum_{k \geq 1} \frac{1}{k} z^k. \tag{17}$$

We can combine the second formula with Eq. (7) to get the generating function for the harmonic numbers:

$$\frac{1}{1-z} \ln \frac{1}{1-z} = z + \tfrac{3}{2} z^2 + \tfrac{11}{6} z^3 + \cdots = \sum_{k \geq 0} H_k z^k. \tag{18}$$

F. Known generating functions. Whenever it is possible to determine the power series expansion of a function, we have implicitly found the generating function for a particular sequence. These special functions can be quite useful in conjunction with the operations described above.

The most important power series expansions are given in the following list.

i) *Binomial theorem*

$$(1+z)^r = 1 + rz + \frac{r(r-1)}{2} z^2 + \cdots = \sum_{k \geq 0} \binom{r}{k} z^k. \tag{19}$$

When r is a negative integer, we get a special case already reflected in Eqs. (5) and (16):

$$\frac{1}{(1-z)^{n+1}} = \sum_{k\geq 0} \binom{-n-1}{k} (-z)^k = \sum_{k\geq 0} \binom{n+k}{n} z^k. \tag{20}$$

There is also a generalization, which was proved in exercise 1.2.6–25:

$$x^r = 1 + rz + \frac{r(r-2t-1)}{2} z^2 + \cdots = \sum_{k\geq 0} \binom{r-kt}{k} \frac{r}{r-kt} z^k, \tag{21}$$

if $r \neq kt$ for positive integers k, and if x is the continuous function of z which solves the equation $x^{t+1} = x^t + z$, $x = 1$ when $z = 0$.

ii) *Exponential series*

$$e^z = 1 + z + \frac{1}{2!} z^2 + \cdots = \sum_{k\geq 0} \frac{1}{k!} z^k. \tag{22}$$

In general, we have the following formula involving Stirling numbers:

$$(e^z - 1)^n = z^n + \frac{1}{n+1} \left\{ {n+1 \atop n} \right\} z^{n+1} + \cdots = n! \sum_k \left\{ {k \atop n} \right\} z^k/k!. \tag{23}$$

iii) *Logarithm series*

$$\ln(1+z) = z - \tfrac{1}{2}z^2 + \tfrac{1}{3}z^3 - \cdots = \sum_{k\geq 1} \frac{(-1)^{k+1}}{k} z^k, \tag{24}$$

$$\ln\left(\frac{1}{1-z}\right) = z + \tfrac{1}{2}z^2 + \tfrac{1}{3}z^3 + \cdots = \sum_{k\geq 1} \frac{1}{k} z^k. \tag{25}$$

Using Stirling numbers (cf. Eq. 23), we have a more general equation:

$$\left(\ln\left(\frac{1}{1-z}\right)\right)^n = z^n + \frac{1}{n+1} \left[{n+1 \atop n} \right] z^{n+1} + \cdots = n! \sum_k \left[{k \atop n} \right] z^k/k!. \tag{26}$$

iv) *Miscellaneous*

$$z(z+1)\ldots(z+n-1) = \sum_k \left[{n \atop k} \right] z^k, \tag{27}$$

$$\frac{z^n}{(1-z)(1-2z)\ldots(1-nz)} = \sum_k \left\{ {k \atop n} \right\} z^k, \tag{28}$$

$$\frac{z}{e^z - 1} = 1 - \tfrac{1}{2}z + \tfrac{1}{12}z^2 + \cdots = \sum_{k\geq 0} \frac{B_k z^k}{k!}. \tag{29}$$

The coefficients B_k which appear in the last formula are the *Bernoulli numbers;* they will be examined further in Section 1.2.11.2, and a table of Bernoulli numbers appears in Appendix B.

Another identity, analogous to Eq. (21), is the following (see exercise 2.3.4.4–29):

$$x^r = 1 + rz + \frac{r(r+2t)}{2} z^2 + \cdots = \sum_{k \geq 0} \frac{r(r+kt)^{k-1}}{k!} z^k, \tag{30}$$

if x is the continuous function of z which solves the equation $x = e^{zx^t}$, where $x = 1$ when $z = 0$.

We conclude this section by returning to a problem that was only partially solved in Section 1.2.3. We saw (Eq. 1.2.3–13 and exercise 1.2.3–29) that

$$\sum_{1 \leq i \leq j \leq n} x_i x_j = \frac{1}{2}\left(\left(\sum_{1 \leq k \leq n} x_k\right)^2 + \left(\sum_{1 \leq k \leq n} x_k^2\right)\right);$$

$$\sum_{1 \leq i \leq j \leq k \leq n} x_i x_k x_j = \frac{1}{6}\left(\left(\sum_{1 \leq k \leq n} x_k\right)^3 + 3\left(\sum_{1 \leq k \leq n} x_k\right)\left(\sum_{1 \leq k \leq n} x_k^2\right) + 2\left(\sum_{1 \leq k \leq n} x_k^3\right)\right).$$

In general, suppose that we have n numbers $x_1, x_2, \ldots, x_n$ and we want the sum

$$a_m = \sum_{1 \leq j_1 \leq \ldots \leq j_m \leq n} x_{j_1} \ldots x_{j_m}.$$

If possible, this sum should be expressed in terms of $S_1, S_2, \ldots, S_m$, where

$$S_j = \sum_{1 \leq k \leq n} x_k^j, \tag{31}$$

the sum of jth powers. Using this more compact notation, the above formulas become $a_2 = \frac{1}{2}S_1^2 + \frac{1}{2}S_2$; $a_3 = \frac{1}{6}S_1^3 + \frac{1}{2}S_1S_2 + \frac{1}{3}S_3$.

We can attack this problem by setting up the generating function

$$G(z) = 1 + a_1 z + a_2 z^2 + \cdots = \sum_{k \geq 0} a_k z^k. \tag{32}$$

By our rules for multiplying series, we find that

$$\begin{aligned} G(z) &= (1 + x_1 z + x_1^2 z^2 + \cdots) \cdots (1 + x_n z + x_n^2 z^2 + \cdots) \\ &= \frac{1}{(1 - x_1 z) \cdots (1 - x_n z)}. \end{aligned} \tag{33}$$

So $G(z)$ is the reciprocal of a polynomial. It often helps to take the logarithm

of a product, and we find that

$$\ln G(z) = \ln\left(\frac{1}{1-x_1z}\right) + \cdots + \ln\left(\frac{1}{1-x_nz}\right)$$
$$= \left(\sum_{k\ge1}\frac{x_1^kz^k}{k}\right) + \cdots + \left(\sum_{k\ge1}\frac{x_n^kz^k}{k}\right) = \sum_{k\ge1}\frac{S_kz^k}{k}.$$

Now $\ln G(z)$ has been expressed in terms of the S's, so all we must do to obtain the answer to our problem is to compute the power series expansion of $G(z)$ again:

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k\ge1}\frac{S_kz^k}{k}\right) = \prod_{k\ge1} e^{S_kz^k/k}$$
$$= \left(1 + S_1z + \frac{S_1^2z^2}{2!} + \cdots\right)\left(1 + \frac{S_2z^2}{2} + \frac{S_2^2z^4}{2^2\cdot 2!} + \cdots\right)\cdots$$
$$= \sum_{m\ge0}\left(\sum_{\substack{k_1,k_2,\ldots,k_m\ge0\\ k_1+2k_2+\cdots+mk_m=m}} \frac{S_1^{k_1}}{1^{k_1}k_1!}\,\frac{S_2^{k_2}}{2^{k_2}k_2!}\cdots\frac{S_m^{k_m}}{m^{k_m}k_m!}\right)z^m. \tag{34}$$

The parenthesized quantity is a_m. This rather imposing sum is really not complicated when it is examined carefully. The number of terms for a particular value of m is $p(m)$, the number of partitions of m (cf. Section 1.2.1). For example, one partition of 12 is

$$12 = 1 + 2 + 2 + 2 + 5;$$

this corresponds to a solution of the equation $k_1 + 2k_2 + \cdots + 12k_{12} = 12$, where k_j is the number of j's in the partition. In our example $k_1 = 1$, $k_2 = 3$, $k_5 = 1$, and the other k's are zero; so we get the term

$$\frac{S_1}{1^11!}\,\frac{S_2^3}{2^33!}\,\frac{S_5}{5^11!} = \tfrac{1}{240}S_1S_2^3S_5$$

as part of the expression for a_{12}.

For a table of the coefficients appearing in Eq. (34), as well as a table of the similar coefficients in Faa di Bruno's formula (exercise 1.2.5–21), see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun, (U.S. Gov't Printing Office, 1964), Table 24.2.

EXERCISES

1. [*M12*] What is the generating function for the sequence 2, 5, 13, 35, . . . $= \langle 2^n + 3^n \rangle$?

▶ 2. [*M13*] Prove Eq. (11).

3. [*HM21*] Differentiate the generating function (Eq. 18) for $\langle H_n \rangle$, and compare this with the generating function for $\langle \sum_{1 \le k \le n} H_k \rangle$. What relation can you deduce?

4. [*M01*] Explain why Eq. (19) is a special case of Eq. (21).

5. [*M20*] Prove Eq. (23) by induction on n.

▶ 6. [*HM15*] Find the generating function for

$$\sum_{1\le k<n} \frac{1}{k(n-k)};$$

differentiate it and express the coefficients in terms of harmonic numbers.

7. [*M20*] Verify all the steps leading to Eq. (34).

8. [*M23*] Find the generating function for $p(n)$, the number of partitions of n.

9. [*M11*] In the notation of Eqs. (31) and (32), what is a_4 in terms of S_1, S_2, S_3, and S_4?

▶ 10. [*M25*] An *elementary symmetric function* is defined by the formula

$$b_m = \sum_{1\le j_1<\cdots<j_m\le n} x_{j_1}\ldots x_{j_m}.$$

(This is the same as a_m of Eq. (32), except that equal subscripts are not allowed.) Find the generating function for b_m and then express b_m in terms of the S_j in Eq. (31). Write out the formulas for b_1, b_2, b_3, and b_4.

11. [*HM30*] Set up the generating function for the sequence $\langle n!\rangle$ and study properties of this function.

▶ 12. [*M20*] Suppose that we have a doubly subscripted sequence a_{mn} for $m, n = 0, 1, \ldots$; show how this double sequence can be represented by a *single* generating function of two variables, and determine the generating function for the sequence $a_{mn} = \binom{n}{m}$.

13. [*HM22*] The "Laplace transform" of a function $f(x)$ is the function $\mathbf{L}f(s) = \int_0^\infty e^{-st}f(t)\,dt$. Given that $a_0, a_1, a_2, \ldots$ is an infinite sequence having a convergent generating function, let $f(x)$ be the step function $\sum_{0\le k\le x} a_k$. Express the Laplace transform of $f(x)$ in terms of the generating function G for this sequence.

14. [*HM21*] Prove Eq. (13).

15. [*M28*] By considering $H(w) = \sum_{n\ge 0} G_n(z)w^n$, find a "closed form" for the generating function

$$G_n(z) = \sum_{0\le k\le n} \binom{n-k}{k} z^k.$$

16. [*M22*] Give a simple formula for the generating function $G_{nr}(z) = \sum_k a_{nkr}z^k$, where a_{nkr} is the number of ways to choose k things out of n objects, subject to the condition that each object may be chosen at most r times. (If $r = 1$, we have $\binom{n}{k}$ ways, and if $r \ge k$, we have the number of combinations with repetitions (cf. exercise 1.2.6–60).)

17. [*M25*] What are the coefficients of $1/(1-z)^w$ if this function is expanded into a *double* power series in terms of both z and w?

▶ **18.** [*M25*] Given positive integers n and r, find a simple formula for the value of the following sums:

$$\text{(a)} \quad \sum_{1\le k_1<k_2<\cdots<k_r\le n} k_1k_2\ldots k_r; \qquad \text{(b)} \quad \sum_{1\le k_1\le k_2\le\cdots\le k_r\le n} k_1k_2\ldots k_r.$$

(For example, when $n = 3$, $r = 2$, the sums are $1\cdot 2+1\cdot 3+2\cdot 3$ and $1\cdot 1+1\cdot 2+1\cdot 3+2\cdot 2+2\cdot 3+3\cdot 3$, respectively.)

19. [*HM32*] (K. F. Gauss.) The sums of the following infinite series are well known:

$$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots = \ln 2; \qquad 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots = \frac{\pi}{4};$$

$$1-\frac{1}{4}+\frac{1}{7}-\frac{1}{10}+\cdots = \frac{\pi}{3\sqrt{3}}+\frac{1}{3}\ln 2.$$

These series may be written respectively as

$$\frac{1}{2}\sum_{n\ge 0}\left(\frac{1}{n+\frac{1}{2}}-\frac{1}{n+1}\right); \quad \frac{1}{4}\sum_{n\ge 0}\left(\frac{1}{n+\frac{1}{4}}-\frac{1}{n+1}\right)-\frac{1}{4}\sum_{n\ge 0}\left(\frac{1}{n+\frac{3}{4}}-\frac{1}{n+1}\right);$$

and

$$\frac{1}{6}\sum_{n\ge 0}\left(\frac{1}{n+\frac{1}{6}}-\frac{1}{n+1}\right)-\frac{1}{6}\sum_{n\ge 0}\left(\frac{1}{n+\frac{2}{3}}-\frac{1}{n+1}\right).$$

Prove that, in general, the series

$$\sum_{n\ge 0}\left(\frac{1}{n+p/q}-\frac{1}{n+1}\right)$$

has the value

$$\frac{\pi}{2}\cot\frac{p}{q}\pi+\ln 2q-2\sum_{0<k<q/2}\cos\frac{2pk}{q}\pi\cdot\ln\sin\frac{k}{q}\pi,$$

when p and q are integers with $0 < p < q$. [*Hint:* By Abel's limit theorem the sum is

$$\lim_{x\to 1-}\sum_{n\ge 0}\left(\frac{1}{n+p/q}-\frac{1}{n+1}\right)x^{p+nq}.$$

Use Eq. (13) to express this power series in such a way that the limit can be evaluated readily.]

1.2.10. Analysis of an Algorithm

Let us now apply some of the techniques of the preceding sections to the study of a typical algorithm.

Algorithm M (*Find the maximum*). Given n elements $X[1], X[2], \ldots, X[n]$, we will find m and j such that $m = X[j] = \max_{1 \le k \le n} X[k]$, and for which j is as large as possible.

M1. [Initialize.] Set $j \leftarrow n$, $k \leftarrow n - 1$, $m \leftarrow X[n]$.

M2. [All tested?] If $k = 0$, the algorithm terminates.

M3. [Compare.] If $X[k] \le m$, go to M5.

M4. [Change m.] Set $j \leftarrow k$, $m \leftarrow X[k]$. (Now m is the current maximum.)

M5. [Decrease k.] Decrease k by one, return to M2. ∎

This rather obvious algorithm may seem so trivial we should not bother to analyze it in detail; but it actually makes a good demonstration of the manner in which more complicated algorithms may be studied. Analysis of algorithms is quite important in computer programming, because there are usually several algorithms available for a particular application and we would like to know which is best.

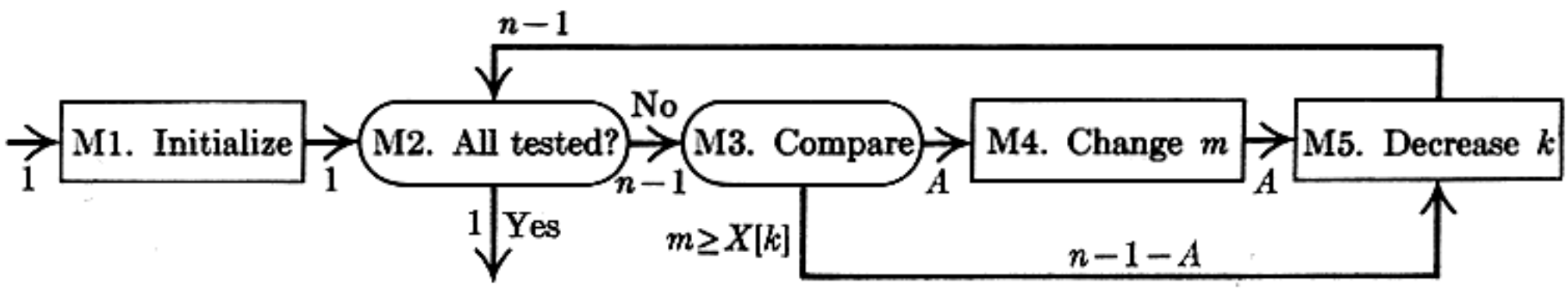


Fig. 9. Algorithm M. Labels on the arrows indicate the number of times each path is taken. Note that "Kirchhoff's law" must be satisfied, i.e., the amount of flow into each node must equal the amount of flow going out.

Algorithm M requires a fixed amount of storage, so we will analyze only the time required to perform it. To do this, we will *count the number of times each step is executed* (cf. Fig. 9):

Step number	Number of times
M1	1
M2	n
M3	$n - 1$
M4	A
M5	$n - 1$

Knowing the number of times each step is executed gives us the information necessary to determine the running time on a particular computer.

In the above table we know everything except the quantity A, which is the number of times we must change the value of the current maximum. To complete the analysis, we will study this interesting quantity A.

The analysis usually consists of finding the *minimum* value of A (for optimistic people), the *maximum* value of A (for pessimistic people), the *average* value of A (for probabilistic people), and the *standard deviation* of A (a quantitative indication of how close to the average we may expect the value to be).

The *minimum* value of A is zero; this happens if $X[n] = \max_{1 \le k \le n} X[k]$. The *maximum* value is $n - 1$; this happens in case $X[1] > X[2] > \cdots > X[n]$.

Thus the average value lies between 0 and $n - 1$. Is it $\frac{1}{2}n$? Is it $\frac{1}{3}n$? To answer this question we need to define what we mean by the average; and to properly define the average, we must make some assumptions about the expected characteristics of the input data $X[1], X[2], \ldots, X[n]$. *We will assume that the $X[k]$ are distinct values, and that each of the $n!$ permutations of these values is equally likely.* (This is a reasonable assumption to make in most situations, but the analysis can be carried out under other assumptions, as shown in the exercises at the end of this section.)

The performance of Algorithm M does not depend on what the precise values of the $X[k]$ are; only the relative order is involved. For example, suppose that $n = 3$. We will say that each of the following six possibilities is equally probable:

Situation	Value of A	Situation	Value of A
$X[1] < X[2] < X[3]$	0	$X[2] < X[3] < X[1]$	1
$X[1] < X[3] < X[2]$	1	$X[3] < X[1] < X[2]$	1
$X[2] < X[1] < X[3]$	0	$X[3] < X[2] < X[1]$	2

The average value of A when $n = 3$ comes to $(0 + 1 + 0 + 1 + 1 + 2)/6 = 5/6$.

It is clear that we may take $X[1], X[2], \ldots, X[n]$ to be the numbers $1, 2, \ldots, n$ in some order; under our assumption we regard each of the $n!$ permutations as equally likely. The *probability* that A has the value k will be

$$p_{nk} = (\text{number of permutations of } n \text{ objects for which } A = k)/n! \qquad (1)$$

For example, from our table above, $p_{30} = \frac{1}{3}$, $p_{31} = \frac{1}{2}$, $p_{32} = \frac{1}{6}$.

The *average* ("mean") value is defined, as usual, to be

$$A_n = \sum_k k p_{nk}. \qquad (2)$$

The *variance* V_n is defined to be the average value of $(A - A_n)^2$; we have therefore

$$V_n = \sum_k (k - A_n)^2 p_{nk} = \sum_k k^2 p_{nk} - 2A_n \sum_k k p_{nk} + A_n^2 \sum_k p_{nk}$$
$$= \sum_k k^2 p_{nk} - 2A_n A_n + A_n^2 = \sum_k k^2 p_{nk} - A_n^2. \qquad (3)$$

Finally, the *standard deviation* σ_n is defined to be $\sqrt{V_n}$.

We can determine the behavior of A by determining the probabilities p_{nk}. It is not hard to do this inductively: by Eq. (1) we want to count the numbers of permutations on n elements that have $A = k$.

Consider the permutations $x_1x_2 \ldots x_n$ on $\{1, 2, \ldots, n\}$ (cf. Section 1.2.5). If $x_1 = n$, the value of A is *one higher* than the value obtained on $x_2 \ldots x_n$; if $x_1 \neq n$, the value of A is *exactly the same* as its value on $x_2 \ldots x_n$. Therefore we find that

$$p_{nk} = \frac{1}{n} p_{(n-1)(k-1)} + \frac{n-1}{n} p_{(n-1)k}. \tag{4}$$

This equation will determine p_{nk} if we provide the initial conditions

$$p_{1k} = \delta_{0k}; \qquad \text{and} \qquad p_{nk} = 0 \quad \text{if} \quad k < 0. \tag{5}$$

We can now get information about the quantities p_{nk} by using generating functions. Let

$$G_n(z) = p_{n0} + p_{n1}z + \cdots = \sum_k p_{nk}z^k. \tag{6}$$

(We know that $A \leq n - 1$, so $p_{nk} = 0$ for large values of k; thus $G_n(z)$ is actually a polynomial, even though an infinite sum has been specified for convenience.)

From Eq. (5) we have $G_1(z) = 1$; and from Eq. (4) we have

$$G_n(z) = \frac{z}{n} G_{n-1}(z) + \frac{n-1}{n} G_{n-1}(z) = \frac{z+n-1}{n} G_{n-1}(z). \tag{7}$$

(The reader should study the relation between Eqs. (4) and (7) carefully.) We can now see that

$$\begin{aligned} G_n(z) = \frac{z+n-1}{n} G_{n-1}(z) &= \frac{z+n-1}{n} \frac{z+n-2}{n-1} G_{n-2}(z) = \cdots \\ &= \frac{1}{n!}(z+n-1)(z+n-2)\ldots(z+1) \\ &= \frac{1}{z+n}\binom{z+n}{n}. \end{aligned} \tag{8}$$

So $G_n(z)$ is essentially a binomial coefficient!

This function appears in the previous section (Eq. 1.2.9–27), where we have

$$G_n(z) = \frac{1}{n!} \sum_k \begin{bmatrix} n \\ k \end{bmatrix} z^{k-1}.$$

Therefore p_{nk} can be expressed in terms of Stirling numbers:

$$p_{nk} = \begin{bmatrix} n \\ k+1 \end{bmatrix} \Big/ n! \tag{9}$$

Figure 10 shows the approximate sizes of p_{nk} when $n = 12$.

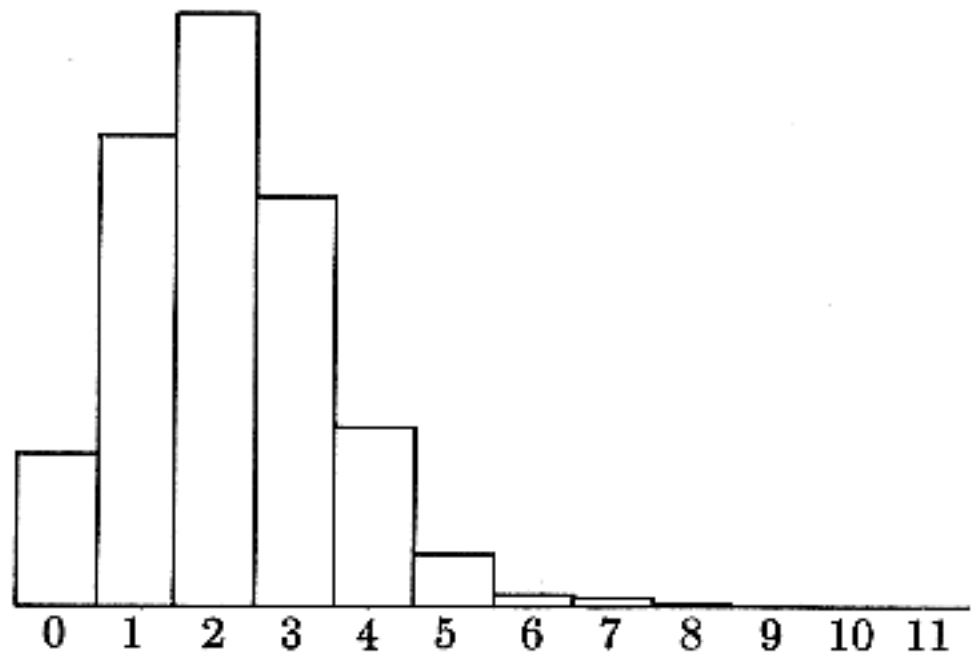


Fig. 10. Probability distribution for step M4, when $n = 12$. The average is $58301/22720 \approx 2.11$.

Now all we must do is plug this value of p_{nk} into Eqs. (2) and (3) and we have the desired average value. But this is easier said than done. It is, in fact, unusual to be able to determine the probabilities p_{nk} explicitly; in most problems we will know the generating function $G_n(z)$, but we will not have any special knowledge about the actual probabilities. The important fact is that *we can determine the mean and variance easily from the generating function itself.*

To see this, let us suppose that we have a generating function whose coefficients represent probabilities:

$$G(z) = p_0 + p_1 z + p_2 z^2 + \cdots .$$

Here p_k is the probability that some event has a value k. We wish to calculate the quantities

$$\text{mean}\,(G) = \sum_k k p_k, \qquad \text{var}\,(G) = \sum_k k^2 p_k - (\text{mean}\,(G))^2. \tag{10}$$

Using differentiation, it is not hard to discover how to do this. Note that

$$G(1) = 1, \tag{11}$$

since $G(1) = p_0 + p_1 + p_2 + \cdots$ is the sum of all possible probabilities. Similarly, since $G'(z) = \sum k p_k z^{k-1}$, we have

$$\text{mean}\,(G) = \sum_k k p_k = G'(1). \tag{12}$$

Finally, we apply differentiation again and we obtain (see exercise 2)

$$\text{var}\,(G) = G''(1) + G'(1) - G'(1)^2. \tag{13}$$

Equations (12) and (13) give the desired expressions of the mean and variance in terms of the generating function.

In our case, we wish to calculate $G_n'(1) = A_n$. From Eq. (7) we have

$$G_n'(z) = \frac{1}{n} G_{n-1}(z) + \frac{z+n-1}{n} G_{n-1}'(z);$$

$$G_n'(1) = \qquad \frac{1}{n} \qquad + G_{n-1}'(1).$$

From the initial condition $G_1'(1) = 0$, we find therefore

$$A_n = G_n'(1) = H_n - 1. \tag{14}$$

This is the desired average number of times step M4 is executed; it is approximately $\ln n$ when n is large. [*Note:* The rth moment is the coefficient of z^n in $1/(1-z) \sum_k \left\{ {r \atop k} \right\} \ln\left(1/(1-z)^k\right)$, and it has the approximate value $(\ln n)^r$; see *CACM* **9** (1966), 342.]

We can proceed similarly to calculate the variance V_n. Before doing this, let us state an important simplification:

Theorem A. *Let G, H be two generating functions with $G(1) = H(1) = 1$. If the quantities* mean (G), var (G) *are defined by Eqs.* (12), (13), *we have*

$$\text{mean}\,(GH) = \text{mean}\,(G) + \text{mean}\,(H); \quad \text{var}\,(GH) = \text{var}\,(G) + \text{var}\,(H). \tag{15}$$

We will prove this theorem later. It tells us that the mean and variance of a product of generating functions may be reduced to a sum. ▮

Letting $Q_n(z) = (z+n-1)/n$, we have $Q_n'(1) = 1/n$, $Q_n''(1) = 0$; hence

$$\text{mean}\,(Q_n) = \frac{1}{n}, \qquad \text{var}\,(Q_n) = \frac{1}{n} - \frac{1}{n^2}.$$

Finally, since $G_n(z) = \prod_{2 \le k \le n} Q_k(z)$, it follows that

$$\text{mean}\,(G_n) = \sum_{2 \le k \le n} \text{mean}\,(Q_k) = \sum_{2 \le k \le n} \frac{1}{k} = H_n - 1$$

$$\text{var}\,(G_n) = \sum_{2 \le k \le n} \text{var}\,(Q_k) = \sum_{1 \le k \le n} \left(\frac{1}{k} - \frac{1}{k^2}\right) = H_n - H_n^{(2)}.$$

Summing up, we have found the desired statistics related to quantity A:

$$A = (\min 0, \qquad \text{ave}\ H_n - 1, \qquad \max n - 1, \qquad \text{dev}\ \sqrt{H_n - H_n^{(2)}}). \tag{16}$$

The notation used in Eq. (16) will be used to describe the statistical characteristics of other probabilistic quantities throughout this book.

We have completed the analysis of Algorithm M; the new feature that has appeared in this analysis is the introduction of probability theory. Not much probability theory is required for most of the applications in this book: the

simple counting techniques and the definitions of mean, variance, and standard deviation which have already been given will suffice.

Let us consider some simple probability problems, to get a little more practice using these methods. In all probability the first problem that comes to mind is a coin-tossing problem. Suppose we flip a coin n times and there is a probability p that "heads" turns up at each toss; what is the average number of heads which will occur? What is the standard deviation?

We will consider our coin to be biased, i.e., we will not assume that $p = \frac{1}{2}$. This makes the problem more interesting, and, furthermore, every real coin is biased (else we could not tell one side from the other!).

Proceeding as before, we let p_{nk} be the probability that k heads will occur, and let $G_n(z)$ be the corresponding generating function. We have clearly

$$p_{nk} = p \cdot p_{(n-1)(k-1)} + q \cdot p_{(n-1)k}. \tag{17}$$

Here, $q = 1 - p$ is the probability that "tails" turns up at each toss. As before, we argue from Eq. (17) that $G_n(z) = (q + pz)G_{n-1}(z)$; and from the obvious initial condition that $G_1(z) = q + pz$ we have

$$G_n(z) = (q + pz)^n. \tag{18}$$

Hence

$$\text{mean } (G_n) = n \text{ mean } (G_1) = pn;$$
$$\text{var } (G_n) = n \text{ var } (G_1) = (p - p^2)n = pqn.$$

For the number of heads, we have therefore

$$(\text{min } 0, \qquad \text{ave } pn, \qquad \text{max } n, \qquad \text{dev } \sqrt{pqn}). \tag{19}$$

Figure 11 shows the values of p_{nk} when $p = \frac{3}{5}$, $n = 12$. When the standard deviation is proportional to $\sqrt{n}$ and the difference between maximum and minimum is proportional to n, we may consider the situation "stable" about the average.

Let us work one more simple problem. Suppose that in some process there is *equal* probability of obtaining the values $1, 2, \ldots, n$. The generating function for this situation is

$$G(z) = \frac{1}{n}z + \frac{1}{n}z^2 + \cdots + \frac{1}{n}z^n = \frac{1}{n}\frac{z^{n+1} - z}{z - 1}. \tag{20}$$

We find after some rather laborious calculation that

$$G'(z) = \frac{nz^{n+1} - (n+1)z^n + 1}{n(z-1)^2},$$
$$G''(z) = \frac{n(n-1)z^{n+1} - 2(n+1)(n-1)z^n + n(n+1)z^{n-1} - 2}{n(z-1)^3}. \tag{21}$$

Now to calculate the mean and variance, we need to know $G'(1)$ and $G''(1)$; but the form in which we have expressed these equations reduces to 0/0 when

we substitute $z = 1$. This makes it necessary to find the limit as z approaches unity, and that is a nontrivial task (cf. exercise 6). Here we have a case where it is much easier to compute from the probabilities directly, rather than derive mean and variance from the generating function. The statistics in this case are

$$\left(\min 1, \qquad \text{ave } \frac{n+1}{2}, \qquad \max n, \qquad \text{dev } \sqrt{\frac{(n+1)(n-1)}{12}}\right). \tag{22}$$

In this case the deviation of approximately $0.289n$ gives us a recognizably *unstable* situation.

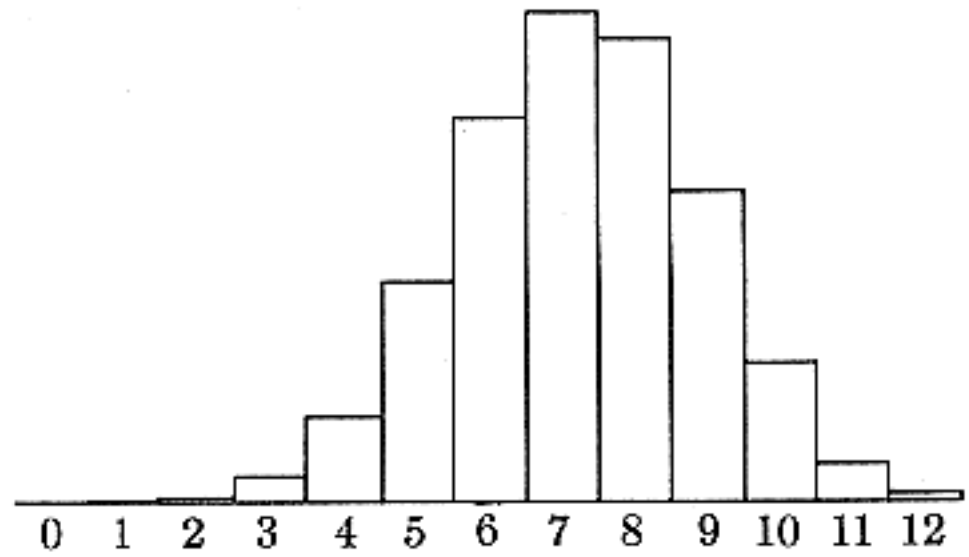


Fig. 11. Probability distribution for coin-tossing; 12 tosses with a chance of success equal to $\frac{3}{5}$ at each toss.

We conclude this section by proving Theorem A and relating our notions to classical probability theory. When $G(z) = p_0 + p_1 z + p_2 z^2 + \cdots$ represents a probability distribution for some quantity X, that is, if p_k is the probability that $X = k$, and X takes on only nonnegative integral values, we have $p_k \geq 0$ and $G(1) = 1$. The quantity $G(e^{it}) = p_0 + p_1 e^{it} + p_2 e^{2it} + \cdots$ is conventionally called the *characteristic function* of this distribution. The distribution given by the product of two such generating functions is called the *convolution* of the two distributions, and it represents the sum of two independent random variables belonging to those distributions.

The mean and variance are just two of the so-called *semi-invariants* or *cumulants* introduced by Thiele in 1903. The semi-invariants $\kappa_1, \kappa_2, \kappa_3, \ldots$ are defined by the rule

$$\frac{\kappa_1 t}{1!} + \frac{\kappa_2 t^2}{2!} + \frac{\kappa_3 t^3}{3!} + \cdots = \ln G(e^t). \tag{23}$$

We have

$$\kappa_n = \frac{d^n}{dt^n} \ln G(e^t)\bigg|_{t=0};$$

in particular,

$$\kappa_1 = \frac{e^t G'(e^t)}{G(e^t)}\bigg|_{t=0} = G'(1),$$

and

$$\kappa_2 = \frac{e^{2t} G''(e^t)}{G(e^t)} + \frac{e^t G'(e^t)}{G(e^t)} - \frac{e^{2t} G'(e^t)^2}{G(e^t)^2}\bigg|_{t=0} = G''(1) + G'(1) - G'(1)^2.$$

Since the semi-invariants are defined in terms of the *logarithm* of a generating function, Theorem A is obvious, and, in fact, it can be generalized to apply to all of the semi-invariants.

A *normal distribution* is one for which all semi-invariants are zero except the mean and variance. In a normal distribution, the difference between a random value and its mean is less than the standard deviation

$$\frac{1}{\sqrt{2\pi}}\int_{-1}^{+1} e^{-t^2/2}\,dt = 68.2689492137 09\%$$

of the time. The difference is less than twice the standard deviation 95.449973610364% of the time, and it is less than three times the standard deviation 99.730020393674% of the time. Both of the distributions specified by Eqs. (8) and (18) are *approximately* normal when n is large (see exercises 13 and 14).

EXERCISES

1. [*10*] Determine the value of p_{n0} from Eqs. (4) and (5) and, considering Algorithm M, interpret this result.

2. [*HM16*] Derive Eq. (13) from Eq. (10).

3. [*M15*] What are the minimum, maximum, average, and standard deviation of the number of times step M4 is executed, if we are using Algorithm M to find the maximum of 1000 randomly ordered, distinct items? (Give your answer as decimal approximations to these quantities.)

4. [*M10*] Give an explicit, closed formula for the values of p_{nk} in the coin-tossing experiment, Eq. (17).

5. [*M13*] What are the mean and the standard deviation of the distribution shown in Fig. 11?

6. [*HM23*] Use L'Hospital's rule to find $G'(1)$ and $G''(1)$ from Eqs. (21).

▶ 7. [*M27*] In our analysis of Algorithm M, we assumed that all the $X[k]$ were distinct. Suppose, instead, that we make only the weaker assumption that $X[1], X[2], \ldots, X[n]$ contain precisely m distinct values; the values are otherwise random, subject to this constraint. What is the probability distribution of A in this case?

▶ 8. [*M20*] Suppose that each $X[k]$ is taken at random from a set of M distinct elements, so that each of the M^n possible choices for $X[1], X[2], \ldots, X[n]$ is considered equally likely. What is the probability that all the $X[k]$ will be distinct?

9. [*M25*] Generalize the result of the preceding exercise to find a formula for the probability that exactly m distinct values occur among the X's. Express your answer in terms of Stirling numbers.

10. [*M20*] Combine the results of the preceding three exercises to obtain a formula for the probability that $A = k$ under the assumption that each X is selected at random from a set of M objects.

11. [*HM20*] Given that $G(z) = p_0 + p_1 z + p_2 z^2 + \cdots$ represents a probability distribution, let $M_n = \sum_k k^n p_k$. (M_n is called the "*n*th *moment*.") Show that $G(e^t) = 1 + M_1 t + M_2 t^2/2! + \cdots$; then using Faa di Bruno's formula (exercise 1.2.5–21), show that

$$\kappa_n = \sum_{\substack{k_1,\ldots,k_n \geq 0 \\ k_1+2k_2+\cdots=n}} \frac{n!(k_1+k_2+\cdots+k_n-1)!(-1)^{k_1+\cdots+k_n-1}}{k_1!(1!)^{k_1}\ldots k_n!(n!)^{k_n}} M_1^{k_1}\ldots M_n^{k_n}.$$

(In particular, $\kappa_1 = M_1$, $\kappa_2 = -M_1^2 + M_2$ (as we already know), $\kappa_3 = 2M_1^3 - 3M_1M_2 + M_3$, $\kappa_4 = -6M_1^4 - 3M_2^2 + 12M_1^2M_2 - 4M_1M_3 + M_4$.)

▸ **12.** [*M15*] What happens to the semi-invariants of a distribution if we change $G(z)$ to $G_1(z) = z^n G(z)$?

13. [*HM38*] A sequence of characteristic functions $G_n(z)$ with means μ_n and deviations σ_n is said to *approach a normal distribution* if

$$\lim_{n\to\infty} e^{-t\mu_n/\sigma_n} G_n(e^{t/\sigma_n}) = e^{t^2/2}$$

for all *imaginary* values of t, that is, whenever $t = ui$ for a real number u. Using $G_n(z)$ as given by Eq. (8), show that $G_n(z)$ approaches a normal distribution. (This is a theorem of Goncharov, *Izv. Akad. Nauk SSSR Ser. Math.* **8** (1944).)

Note: "Approaching the normal distribution," as defined here, can be shown to be equivalent to the fact that

$$\lim_{n\to\infty} \text{probability}\left(\frac{X_n - \mu_n}{\sigma_n} \leq x\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}\,dt,$$

where X_n is a random quantity whose probabilities are specified by $G_n(z)$. This is a special case of P. Levy's important "continuity theorem," which is a basic result in mathematical probability theory; a proof of the result would take us rather far afield, although it is not extremely difficult [for example, see *Limit Distributions for Sums of Independent Random Variables* by B. V. Gnedenko and A. N. Kolmogorov, tr. by K. L. Chung (Reading, Mass.: Addison-Wesley, 1954)].

14. [*HM30*] (A. de Moivre.) Using the conventions of the previous exercise, show that the binomial distribution $G_n(z)$ given by Eq. (18) approaches the normal distribution.

▸ **15.** [*M21*] Let z be a positive number. What is the average value of the quantity z^A taken over all permutations of order n, if A is the quantity appearing in the analysis of Algorithm M?

16. [*HM23*] When the probability that some quantity has the value k is $e^{-\mu}(\mu^k/k!)$, it is said to have the "Poisson distribution with mean μ."

a) What is the generating function for this set of probabilities?
b) What are the values of the semi-invariants?
c) Show that as $n \to \infty$ the Poisson distribution with mean np approaches the normal distribution in the sense of exercise 13.

▸ **17.** [*M27*] Let $f(z)$ and $g(z)$ be generating functions which represent probability distributions.

a) Show that $h(z) = g(f(z))$ is also a generating function representing a probability distribution.

b) Interpret the significance of $h(z)$ in terms of $f(z)$ and $g(z)$. (What is the *meaning* of the probabilities represented by the coefficients of $h(z)$?)

c) Give formulas for the mean and variance of h in terms of those for f, g.

18. [*M28*] Suppose that the distinct values taken on by $X[1], X[2], \ldots, X[n]$ in Algorithm M include exactly k_1 ones, k_2 twos, $\ldots$, k_n n's, arranged in random order. (Here

$$k_1 + k_2 + \cdots + k_n = n.$$

Note that the text's assumption is $k_1 = k_2 = \cdots = k_n = 1$.) Show that in this generalized situation, the generating function, Eq. (8), becomes

$$\left(\frac{k_{n-1}z + k_n}{k_{n-1} + k_n}\right)\left(\frac{k_{n-2}z + k_{n-1} + k_n}{k_{n-2} + k_{n-1} + k_n}\right)\cdots\left(\frac{k_1 z + k_2 + \cdots + k_n}{k_1 + k_2 + \cdots + k_n}\right),$$

using the convention $0/0 = 1$.

*1.2.11. Asymptotic Representations

We often want to find the approximate value of a quantity, instead of an exact value, in order to compare one number to another. For example, Stirling's approximation to $n!$ is a useful representation of this type, and we also have made use of the fact that $H_n \approx \ln n + \gamma$.

The derivations of such "asymptotic" formulas generally involve higher mathematics, although in the following subsections we use nothing more than elementary calculus to get the results we need.

1.2.11.1. The O-notation. A very convenient notation for dealing with approximations was introduced by P. Bachmann in the book *Analytische Zahlentheorie* in 1892. This is the "big-oh" notation which allows us to replace the "$\approx$" sign by "$=$", for example,

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right). \tag{1}$$

(Read, "H sub n equals the natural log of n plus Euler's ("Oiler's") constant plus big O of one over n.")

In general, the notation $O(f(n))$ may be used whenever $f(n)$ is a function of the positive integer n; it stands for *a quantity which is not explicitly known*, except that its magnitude isn't too large. Every appearance of $O(f(n))$ means precisely this: there is a positive constant M such that the number x_n represented by $O(f(n))$ satisfies the condition $|x_n| \leq M|f(n)|$. We do not say *what* the constant M is, and indeed the constant M is usually different for each appearance of O.

For example, Eq. (1) means that $|H_n - \ln n + \gamma| \leq M/n$; the constant M is not specified further, but even if we don't know its value, we do know that the quantity $O(\frac{1}{n})$ will be arbitrarily small if n is large enough.

Let us give some more examples. We know that

$$1^2 + 2^2 + \cdots + n^2 = \tfrac{1}{3}n(n + \tfrac{1}{2})(n + 1) = \tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n,$$

so it follows that

$$1^2 + 2^2 + \cdots + n^2 = O(n^3), \tag{2}$$

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

Equation (3) is a stronger statement than Eq. (2). To justify these equations we shall prove that *if* $P(n) = a_0 + a_1 n + \cdots + a_m n^m$ *is any polynomial of degree* m *or less,* $P(n) = O(n^m)$. This follows because

$$\begin{aligned} |P(n)| \le |a_0| + |a_1|n + \cdots + |a_m|n^m &= (|a_0|/n^m + |a_1|/n^{m-1} + \cdots + |a_m|)n^m \\ &\le (|a_0| + |a_1| + \cdots + |a_m|)n^m, \end{aligned}$$

when $n \ge 1$. So we may take $M = |a_0| + \cdots + |a_m|$.

The O-notation is a big help in approximation work, since it briefly describes a concept which occurs frequently and it suppresses detailed information which is usually irrelevant. Furthermore, it can be manipulated algebraically in familiar ways, provided that a little bit of caution is used.

Many of the rules of algebra can be used together with O-notations, but certain important differences should be mentioned. The most important consideration is the idea of *one-way equalities:* We write $\frac{1}{2}n^2 + n = O(n^2)$, but we *never* write $O(n^2) = \frac{1}{2}n^2 + n$. (Or else, since $\frac{1}{4}n^2 = O(n^2)$, we might come up with the absurd relation $\frac{1}{4}n^2 = \frac{1}{2}n^2 + n$.) We always use the convention that *the right-hand side of an equation does not give more information than the left-hand side;* the right-hand side is a "crudification" of the left.

This convention about the use of "=" may be stated more precisely as follows: "If $\alpha(n)$ and $\beta(n)$ are formulas which involve the $O(f(n))$-notation, then $\alpha(n) = \beta(n)$ means that for any positive constants $M_1, M_2, \ldots, M_k$—one for each appearance of O in $\alpha(n)$—there exist positive constants $M_{k+1}, \ldots, M_m$—one for each appearance of O in $\beta(n)$—such that, for any given value of n, $\alpha(n)$ and $\beta(n)$ represents *sets* of augmented real numbers $S(\alpha(n))$ and $S(\beta(n))$, respectively, for which $S(\alpha(n)) \subseteq S(\beta(n))$." By augmented real numbers, we mean real numbers plus the additional quantity Ω which denotes an undefined value [such as $1/0$, $\lim_{x\to 0} \sin(1/x)$, $\Omega - \Omega$, etc.]. The sets $S(\alpha(n))$ and $S(\beta(n))$ are obtained by considering all possible values x_j such that $|x_j| \le M_j|f(n)|$ substituted for the jth appearance of O. It follows that we may perform most of the operations we are accustomed to doing with the "=" sign: If $\alpha(n) = \beta(n)$ and $\beta(n) = \gamma(n)$, then $\alpha(n) = \gamma(n)$. Also, if $\alpha(n) = \beta(n)$ and if $\delta(n)$ is a formula resulting from the substitution of $\beta(n)$ for some occurrence of $\alpha(n)$ in a formula $\gamma(n)$, then $\gamma(n) = \delta(n)$. These two statements imply, for example, that if $g(x_1, x_2, \ldots, x_m)$ is any real function whatever, and if $\alpha_k(n) = \beta_k(n)$ for $1 \le k \le m$, then $g(\alpha_1(n), \alpha_2(n), \ldots, \alpha_m(n)) = g(\beta_1(n), \beta_2(n), \ldots, \beta_m(n))$.

Here are some of the simple operations we can do with the O-notation:

$$f(n) = O(f(n)), \tag{4}$$

$$c \cdot O(f(n)) = O(f(n)), \qquad \text{if } c \text{ is a constant}, \tag{5}$$

$$O(f(n)) + O(f(n)) = O(f(n)), \tag{6}$$

$$O(O(f(n))) = O(f(n)), \tag{7}$$

$$O(f(n))O(g(n)) = O(f(n)g(n)), \tag{8}$$

$$O(f(n)g(n)) = f(n)O(g(n)). \tag{9}$$

The O-notation is also used more generally with functions of a real variable x. A particular range of values of x is specified, for example, $a \le x \le b$, and we write $O(f(x))$ to stand for any quantity $g(x)$, such that $|g(x)| \le M|f(x)|$ whenever $a \le x \le b$. (As before, M is an unspecified constant.) The notation $O(f(n))$ discussed above is the special case where the variable x is restricted to positive integer values; we usually call the variable n instead of x in this case.

Suppose that $g(x)$ is a function given by an infinite series,

$$g(x) = \sum_{k \ge 0} a_k x^k, \qquad |x| \le r,$$

where the sum of absolute values $\sum_{k \le 0} |a_k x^k|$ also exists. We can then always write

$$g(x) = a_0 + a_1 x + \cdots + a_m x^m + O(x^{m+1}), \qquad |x| \le r. \tag{10}$$

For, $g(x) = a_0 + a_1 x + \cdots + a_m x^m + x^{m+1}(a_{m+1} + a_{m+2}x + \cdots)$; we must only show that the parenthesized quantity is bounded when $|x| \le r$, and it is easy to show that $|a_{m+1}| + |a_{m+2}|r + |a_{m+3}|r^2 + \cdots$ is an upper bound.

For example, consider the generating functions given in Section 1.2.9; we have the important relations

$$e^x = 1 + x + \frac{1}{2!}x^2 + \cdots + \frac{1}{m!}x^m + O(x^{m+1}),$$
$$|x| \le r, \qquad \text{any fixed } r; \tag{11}$$

$$\ln(1+x) = x - \tfrac{1}{2}x^2 + \cdots + \frac{(-1)^{m+1}}{m}x^m + O(x^{m+1}),$$
$$|x| \le r, \qquad \text{any fixed } r < 1; \tag{12}$$

$$(1+x)^\alpha = 1 + \alpha x + \binom{\alpha}{2}x^2 + \cdots + \binom{\alpha}{m}x^m + O(x^{m+1}),$$
$$|x| \le r, \qquad \text{any fixed } r < 1. \tag{13}$$

The statement that r is "fixed" means that r must have a definite value when the O-notation is used. We obviously have $e^x = O(1)$ when $|x| \le r$, since $|e^x| \le e^r$, but the constant M implied by the O-notation depends on r. In fact,

it is easy to see that if x is allowed to range over all values $-\infty < x < \infty$, then $e^x \neq O(x^m)$ for *any* m.

Let us give one simple example of the concepts we have introduced so far. Consider the quantity $\sqrt[n]{n}$; as n gets large, the operation of taking an nth root tends to decrease the value, but it is not immediately obvious whether $\sqrt[n]{n}$ decreases or increases. It turns out that $\sqrt[n]{n}$ decreases to unity. Let us consider the slightly more complicated quantity $n(\sqrt[n]{n} - 1)$. Now $(\sqrt[n]{n} - 1)$ gets smaller as n gets bigger; what happens to $n(\sqrt[n]{n} - 1)$?

This problem is rather easily solved by applying the above formulas. We have

$$\sqrt[n]{n} = e^{\ln n/n} = 1 + (\ln n/n) + O((\ln n/n)^2). \tag{14}$$

This equation proves our previous contention that $\sqrt[n]{n} \to 1$. Furthermore, it tells us that

$$\begin{aligned} n(\sqrt[n]{n} - 1) &= n(\ln n/n + O((\ln n/n)^2) \\ &= \ln n + O((\ln n)^2/n). \end{aligned}$$

So we find that $n(\sqrt[n]{n} - 1)$ is approximately equal to $\ln n$; the difference is $O((\ln n)^2/n)$, which approaches zero as n approaches infinity (see exercise 8).

EXERCISES

1. [*HM01*] What is $\lim_{n\to\infty} O(n^{-1/3})$?

▶ 2. [*M10*] Mr. B. C. Dull obtained astonishing results by using the formula $O(f(n)) - O(f(n)) = 0$; what is his mistake, and what should the right-hand side of his formula be?

3. [*M15*] Multiply $(\ln n + \gamma + O(1/n))$ by $(n + O(\sqrt{n}))$, and express your answer in O-notation.

4. [*M18*] Suppose that $f(n) \neq 0$ for all $n > 0$. Show that if $|x_n| \leq M|f(n)|$ for all $n \geq n_0$, where n_0 is some integer, then $x_n = O(f(n))$. (This is stated differently than the text's definition, which takes $n_0 = 1$.)

5. [*M20*] (a) Given that $r > 0$ and $P(x) = c_0 + c_1x + \cdots + c_mx^m$, show that $P(x) = O(x^m)$, when $x \geq r$. (b) Prove or disprove: $P(x) = O(x^m)$, when $x > 0$.

▶ 6. [*M20*] What is wrong with the following argument? "Since $n = O(n)$, $2n = O(n)$, . . . , we have

$$\sum_{1\leq k\leq n} kn = \sum_{1\leq k\leq n} O(n) = O(n^2)."$$

7. [*HM15*] Prove that if the values of x are allowed to be arbitrarily large, $e^x \neq O(x^m)$ for any power m.

8. [*HM20*] Prove that as $n \to \infty$, $(\ln n)^m/n \to 0$.

9. [*HM20*] Show that $e^{O(x^m)} = 1 + O(x^m)$, $|x| \leq r$.

10. [*HM22*] Make a statement similar to that in the previous exercise about $\ln\left(1 + O(x^m)\right)$.

11. [*M11*] Explain why Eq. (14) is true.

▶ **12.** [*M15*] Give an asymptotic expansion of $n(\sqrt[n]{a} - 1)$, if $a > 0$, to terms $O(1/n^3)$.

1.2.11.2. Euler's summation formula. Perhaps the most useful method for obtaining good approximations is the one due to Leonhard Euler in 1732; his method approximates a finite sum by an integral, and gives us a means to get better and better approximations in many cases.

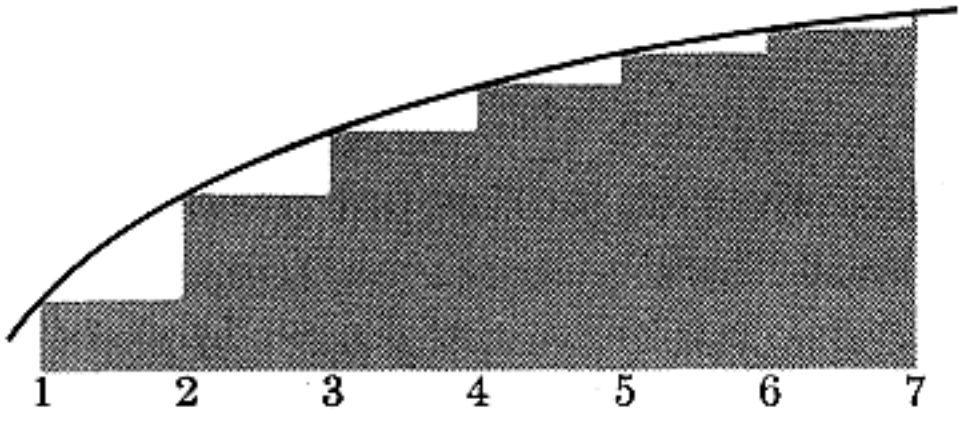


Fig. 12. Comparing a sum with an integral.

Figure 12 shows a comparison of $\int_1^n f(x)\,dx$ and $\sum_{1\le k<n} f(k)$, $n = 7$. Euler's method gives a useful formula for the difference between these two quantities, assuming that $f(x)$ is a differentiable function.

For convenience we use the notation

$$\{x\} = x \bmod 1 = x - \lfloor x \rfloor. \quad = \text{FRAC}(x) \tag{1}$$

Our derivation starts with the following identity:

$$\begin{aligned}\int_k^{k+1} (\{x\} - \tfrac{1}{2})f'(x)\,dx &= (x - k - \tfrac{1}{2})f(x)\Big|_k^{k+1} - \int_k^{k+1} f(x)\,dx \\ &= \tfrac{1}{2}\big(f(k+1) + f(k)\big) - \int_k^{k+1} f(x)\,dx. \end{aligned} \tag{2}$$

(This follows from integration by parts.) Adding both sides of this equation for $1 \le k < n$, we find that

$$\int_1^n (\{x\} - \tfrac{1}{2})f'(x)\,dx = \sum_{1\le k<n} f(k) + \tfrac{1}{2}\big(f(n) - f(1)\big) - \int_1^n f(x)\,dx,$$

that is,

$$\sum_{1\le k<n} f(k) = \int_1^n f(x)\,dx - \tfrac{1}{2}\big(f(n) - f(1)\big) + \int_1^n B_1(\{x\})f'(x)\,dx, \tag{3}$$

where $B_1(x)$ is the polynomial $x - \frac{1}{2}$. This is the desired connection between the sum and the integral.

The approximation can be carried further if we continue to integrate by parts. Before doing this, however, we shall discuss the *Bernoulli numbers*, which

are the coefficients in the following infinite series:

$$\frac{x}{e^x - 1} = B_0 + B_1 x + \frac{B_2 x^2}{2!} + \cdots = \sum_{k \geq 0} \frac{B_k x^k}{k!}. \tag{4}$$

The coefficients of this series, which occur in a wide variety of problems, were introduced by James Bernoulli in 1713. (Some books use a different notation for Bernoulli numbers, but our notation is used in most modern references.) We have

$$B_0 = 1, \qquad B_1 = -\tfrac{1}{2}, \qquad B_2 = \tfrac{1}{6}, \qquad B_3 = 0, \qquad B_4 = -\tfrac{1}{30}. \tag{5}$$

Further values are given in Appendix B. Since

$$\frac{x}{e^x - 1} + \frac{x}{2} = \frac{x}{2}\,\frac{e^x + 1}{e^x - 1} = -\frac{x}{2}\,\frac{e^{-x} + 1}{e^{-x} - 1}$$

is an even function, we see that

$$B_3 = B_5 = B_7 = B_9 = \cdots = 0. \tag{6}$$

The following formula is an immediate consequence of the definition, Eq. (4):

$$\sum_k \binom{n}{k} B_k = B_n + \delta_{n1}. \tag{7}$$

We now define the "Bernoulli polynomial,"

$$B_m(x) = \sum_k \binom{m}{k} B_k x^{m-k}. \tag{8}$$

If $m = 1$, then $B_1(x) = B_0 x + B_1 = x - \frac{1}{2}$, corresponding to the polynomial used above in Eq. (3). If $m > 1$, we have by (7) $B_m(1) = B_m = B_m(0)$; $B_m(\{x\})$ has no discontinuities at integer points x.

The relevance of Bernoulli polynomials and Bernoulli numbers to our problem will soon be clear. We find from Eq. (8) that

$$\begin{aligned} B'_m(x) = \sum_k \binom{m}{k} (m-k) B_k x^{m-k-1} &= m \sum_k \binom{m-1}{k} B_k x^{m-1-k} \\ &= m B_{m-1}(x), \end{aligned} \tag{9}$$

and therefore when $m \geq 1$, we can integrate by parts as follows:

$$\begin{aligned} \frac{1}{m!} \int_1^n B_m(\{x\}) f^{(m)}(x)\, dx = \frac{1}{(m+1)!} \left(B_{m+1}(1) f^{(m)}(n) - B_{m+1}(0) f^{(m)}(1) \right) \\ - \frac{1}{(m+1)!} \int_1^n B_{m+1}(\{x\}) f^{(m+1)}(x)\, dx. \end{aligned}$$

From this result we can continue to improve the approximation, Eq. (3), and we obtain Euler's general formula:

$$\sum_{1\le k<n} f(k) = \int_1^n f(x)\,dx - \tfrac{1}{2}(f(n) - f(1)) + \frac{B_2}{2!}(f'(n) - f'(1)) + \cdots$$
$$+ \frac{(-1)^m B_m}{m!}(f^{(m-1)}(n) - f^{(m-1)}(1)) + R_m$$
$$= \int_1^n f(x)\,dx + \sum_{1\le k\le m} \frac{B_k}{k!}(f^{(k-1)}(n) - f^{(k-1)}(1)) + R_m, \tag{10}$$

where

$$R_m = \frac{(-1)^{m+1}}{m!}\int_1^n B_m(\{x\})f^{(m)}(x)\,dx. \tag{11}$$

The remainder R_m will be small when $B_m(\{x\})f^{(m)}(x)/m!$ is very small, and in fact, it is known that $|B_m(\{x\})| \le |B_m|$ when m is even, and that

$$\left|\frac{B_m(\{x\})}{m!}\right| < \frac{4}{(2\pi)^m}. \tag{12}$$

[See K. Knopp, *Theory and Application of Infinite Series* (Glasgow: Blackie, 1951), Chapter 14.] On the other hand, it usually turns out that the size of $f^{(m)}(x)$ gets large as m increases, so there is a "best" value of m at which R_m has its least value.

It is known that

$$R_{2k} = \theta\frac{B_{2k+2}}{(2k+2)!}(f^{(2k+1)}(n) - f^{(2k+1)}(1)), \qquad 0 < \theta < 1, \tag{13}$$

provided that $f^{(2k+1)}(x)$ tends monotonically toward zero as x increases from 1 to n. (So in these circumstances the remainder has the same sign as, and is less than, the first discarded term.) A simpler version of this result appears in exercise 3.

Let us now apply Euler's formula to some important examples. First, we set $f(x) = 1/x$. The derivatives are $f^{(m)}(x) = (-1)^m m!/x^{m+1}$, so we have, by Eq. (10),

$$H_{n-1} = \ln n + \sum_{1\le k\le m} \frac{B_k}{k}(-1)^{k-1}\left(\frac{1}{n^k} - 1\right) + R_{mn}. \tag{14}$$

Now we find

$$\gamma = \lim_{n\to\infty}(H_{n-1} - \ln n) = \sum_{1\le k\le m}\frac{B_k}{k}(-1)^k + \lim_{n\to\infty} R_{mn}. \tag{15}$$

The fact that $\lim_{n\to\infty} R_{mn} = -\int_1^\infty B_m(\{x\})\,dx/x^{m+1}$ exists proves that the constant γ does in fact exist; now putting Eqs. (14) and (15) together, we

deduce a general approximation for the harmonic numbers:

$$H_{n-1} = \ln n + \gamma + \sum_{1\le k\le m} \frac{(-1)^{k-1}B_k}{kn^k} + \int_n^\infty \frac{B_m(\{x\})\,dx}{x^{m+1}}$$
$$= \ln n + \gamma + \sum_{1\le k\le m} \frac{(-1)^{k-1}B_k}{kn^k} + O\left(\frac{1}{n^m}\right). \tag{16}$$

Furthermore, by Eq. (13) we see that the error is less than the first term discarded. As a particular case we have (adding $1/n$ to both sides)

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon, \qquad 0 < \epsilon < \frac{B_6}{6n^6} = \frac{1}{252n^6}.$$

This is Eq. 1.2.7–3. The Bernoulli numbers B_k for large k get very large (approximately $2[k!/(2\pi)^k]$ when k is even), so Eq. (16) cannot be extended to a convergent infinite series for any fixed value of n.

The same technique may be applied to deduce Stirling's approximation. This time we set $f(x) = \ln x$, and applying Eq. (10), we obtain

$$\ln (n-1)! = n\ln n - n + 1 - \tfrac{1}{2}\ln n + \sum_{1<k\le m} \frac{B_k(-1)^k}{k(k-1)}\left(\frac{1}{n^{k-1}} - 1\right) + R_{mn}. \tag{17}$$

Proceeding as above, we find that

$$\lim_{n\to\infty} (\ln n! - n\ln n + n - \tfrac{1}{2}\ln n) = 1 + \sum_{1<k\le m} \frac{B_k(-1)^{k+1}}{k(k-1)} + \lim_{n\to\infty} R_{mn}$$

exists; let it be called σ ("Stirling's constant") temporarily. We get Stirling's result

$$\ln n! = (n + \tfrac{1}{2})\ln n - n + \sigma + \sum_{1<k\le m} \frac{B_k(-1)^k}{k(k-1)n^{k-1}} + O\left(\frac{1}{n^m}\right). \tag{18}$$

In particular, let $m = 5$; we have

$$\ln n! = (n + \tfrac{1}{2})\ln n - n + \sigma + \frac{1}{12n} - \frac{1}{360n^3} + O\left(\frac{1}{n^5}\right).$$

Taking exponentials, we have

$$n! = e^\sigma \sqrt{n}\left(\frac{n}{e}\right)^n \exp\left(\frac{1}{12n} - \frac{1}{360n^3} + O\left(\frac{1}{n^5}\right)\right).$$

Using the fact that $e^\sigma = \sqrt{2\pi}$ (see exercise 5), and expanding the exponential, we get our final result:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} - \frac{571}{2488320n^4} + O\left(\frac{1}{n^5}\right)\right). \tag{19}$$

EXERCISES

1. [*M18*] Prove Eq. (7).

2. [*HM16*] Note that Eq. (9) follows from Eq. (8) for *any* sequence B_n, not only the sequence defined by Eq. (4). Explain why the latter sequence is necessary for the validity of Eq. (10).

3. [*HM20*] If $f^{(2k)}(x)$ has a constant sign for $1 \le x \le n$, show that

$$|R_{2k}| \le \left| \frac{B_{2k}}{(2k)!} \left(f^{(2k-1)}(n) - f^{(2k-1)}(1)\right) \right|,$$

so the remainder has smaller absolute value than the *last* term computed.

▶ 4. [*HM20*] When $f(x) = x^m$, the high-order derivatives of f are all zero, so Euler's summation formula gives an *exact* value for $\sum_{0 \le k < n} k^m$ in terms of Bernoulli numbers. Express this value in terms of Bernoulli *polynomials*. Check your answer for $m = 0$, 1, 2. (Note that the desired sum runs from 0 to n instead of from 1 to n; Euler's summation formula may be applied with 0 replacing 1 throughout.)

5. [*HM30*] Given that

$$n! = \kappa\sqrt{n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right),$$

show that $\kappa = \sqrt{2\pi}$ by using Wallis's product (exercise 1.2.5–18). [*Hint:* Consider $\binom{2n}{n}$ for large values of n.]

▶ 6. [*HM30*] Show that Stirling's approximation holds for noninteger n as well, i.e., that

$$\Gamma(x+1) = \sqrt{2\pi x}\left(\frac{x}{e}\right)^x \left(1 + O\left(\frac{1}{x}\right)\right), \qquad x \ge a > 0.$$

[*Hint:* Let $f(x) = \ln(x + c)$ in Euler's summation formula, and apply the definition of $\Gamma(x)$ given in Section 1.2.5.]

▶ 7. [*HM32*] What is the approximate value of $1^1 \cdot 2^2 \cdot 3^3 \cdot \cdots \cdot n^n$?

1.2.11.3. Some applications. In this subsection we shall investigate the following three intriguing sums, and we shall deduce their approximate values:

$$P(n) = 1 + \frac{n-1}{n} + \frac{n-2}{n}\,\frac{n-2}{n-1} + \cdots = \sum_{0\le k\le n} \frac{(n-k)^k (n-k)!}{n!}, \tag{1}$$

$$Q(n) = 1 + \frac{n-1}{n} + \frac{n-1}{n}\,\frac{n-2}{n} + \cdots = \sum_{1\le k\le n} \frac{n!}{(n-k)!\,n^k}, \tag{2}$$

$$R(n) = 1 + \frac{n}{n+1} + \frac{n}{n+1}\,\frac{n}{n+2} + \cdots = \sum_{0\le k} \frac{n!\,n^k}{(n+k)!}. \tag{3}$$

These functions, which are similar in appearance yet intrinsically different, arise in several algorithms that we shall encounter later. Both $P(n)$ and $Q(n)$ are

finite sums, while $R(n)$ is an infinite sum. It seems that when n is large, all three of these sums will be nearly equal, although it is not obvious what the approximate value of *any* of these three functions will be. Our quest for approximate values of these functions will lead us through a number of very instructive side results. (The reader may wish to stop reading temporarily and try his hand at studying these functions before going on to see how they are attacked here.)

First, we observe an important connection between $Q(n)$ and $R(n)$:

$$Q(n) + R(n) = \frac{n!}{n^n}\left(\left(1 + n + \cdots + \frac{n^{n-1}}{(n-1)!}\right) + \left(\frac{n^n}{n!} + \frac{n^{n+1}}{(n+1)!} + \cdots\right)\right)$$
$$= \frac{n!\,e^n}{n^n}. \tag{4}$$

To get any further we must therefore consider the partial sums of the series for e^n. By using Taylor's formula with remainder,

$$f(x) = f(0) + f'(0)x + \cdots + \frac{f^{(n)}(0)x^n}{n!} + \int_0^x \frac{t^{n+1}}{(n+1)!} f^{(n+1)}(x-t)\,dt, \tag{5}$$

we are soon led to an important function which is known as the *incomplete gamma function:*

$$\gamma(a, x) = \int_0^x e^{-t} t^{a-1}\,dt. \tag{6}$$

We shall assume that $a > 0$. By exercise 1.2.5–20, we have $\gamma(a, \infty) = \Gamma(a)$; this accounts for the name "incomplete gamma function." It has two series expansions in powers of x (see exercises 2 and 3):

$$\gamma(a, x) = \frac{x^a}{a} - \frac{x^{a+1}}{a+1} + \frac{x^{a+2}}{2!(a+2)} - \cdots = \sum_{k\ge 0} \frac{(-1)^k x^{k+a}}{k!(k+a)}, \tag{7}$$

$$e^x\gamma(a, x) = \frac{x^a}{a} + \frac{x^{a+1}}{a(a+1)} + \frac{x^{a+2}}{a(a+1)(a+2)} + \cdots$$
$$= \sum_{k\ge 0} \frac{x^{k+a}}{a(a+1)\ldots(a+k)}. \tag{8}$$

From the second formula we see the connection with $R(n)$:

$$R(n) = \frac{n!\,e^n}{n^n}\left(\frac{\gamma(n, n)}{(n-1)!}\right). \tag{9}$$

This equation has purposely been written in a more complicated form than necessary, since $\gamma(n, n)$ is a fraction of $\gamma(n, \infty) = \Gamma(n) = (n-1)!$. Thus $R(n)$ lies somewhere between zero and $n!\,e^n/n^n$; by Stirling's formula, $n!\,e^n/n^n$ is approximately $\sqrt{2\pi n}$.

The problem boils down to getting good estimates of $\gamma(n, n)/(n-1)!$. *We shall now determine the approximate value of* $\gamma(x+1, x+y)/\Gamma(x+1)$, when y is fixed and x is large. The methods to be used here are more important than the results, so the reader should study the following derivation carefully.

By definition, we have

$$\begin{aligned}\frac{\gamma(x+1, x+y)}{\Gamma(x+1)} &= \frac{1}{\Gamma(x+1)}\int_0^{x+y} e^{-t}t^x\,dt \\ &= 1 - \frac{1}{\Gamma(x+1)}\int_x^{\infty} e^{-t}t^x\,dt + \frac{1}{\Gamma(x+1)}\int_x^{x+y} e^{-t}t^x\,dt \\ &= 1 - \qquad I_1 \qquad + \qquad I_2. \end{aligned} \tag{10}$$

Now we consider each integral separately.

Estimate of I_1: In the integral I_1, we convert to an integral from 0 to infinity by substituting $t = x(1+u)$:

$$\begin{aligned} I_1 &= \frac{e^{-x}x^x}{\Gamma(x+1)}\int_0^{\infty} xe^{-xu}\,(1+u)^x\,du \\ &= \frac{e^{-x}x^x}{\Gamma(x+1)}\int_0^{\infty} xe^{-xv}\left(1+\frac{1}{u}\right)dv, \end{aligned} \tag{11}$$

$$\text{if} \qquad v = u - \ln(1+u); \quad dv = \left(1 - \frac{1}{1+u}\right)du.$$

This change of variable from u to v is justified, since v is a monotone function of u.

In the last integral we will replace $1 + 1/u$ by a power series in v. We have

$$v = \tfrac{1}{2}u^2 - \tfrac{1}{3}u^3 + \tfrac{1}{4}u^4 - \tfrac{1}{5}u^5 + \cdots = (u^2/2)(1 - \tfrac{2}{3}u + \tfrac{1}{2}u^2 - \tfrac{2}{5}u^3 + \cdots).$$

If $w = \sqrt{2v}$, we have

$$\begin{aligned} w &= u(1 - \tfrac{2}{3}u + \tfrac{1}{2}u^2 - \tfrac{2}{5}u^3 + \cdots)^{1/2} \\ &= u - \tfrac{1}{3}u^2 + \tfrac{7}{36}u^3 - \tfrac{73}{540}u^4 + \tfrac{1331}{12960}u^5 + O(u^6). \end{aligned}$$

(This expansion may be obtained by the binomial theorem; efficient methods for doing this transformation, as well as the other power series manipulations done below, are considered in Section 4.7.) We can now solve for u as a power series in w:

$$u = w + \frac{1}{3}w^2 + \frac{1}{36}w^3 - \frac{1}{270}w^4 + \frac{1}{4320}w^5 + O(w^6);$$

$$\begin{aligned} 1 + \frac{1}{u} &= 1 + \frac{1}{w} - \frac{1}{3} + \frac{1}{12}w - \frac{2}{135}w^2 + \frac{1}{864}w^3 + O(w^4) \\ &= \frac{1}{\sqrt{2}}v^{-1/2} + \frac{2}{3} + \frac{\sqrt{2}}{12}v^{1/2} - \frac{4}{135}v + \frac{\sqrt{2}}{432}v^{3/2} + O(v^2). \end{aligned} \tag{12}$$

In all of these formulas, the O-notation refers to small values of the argument, that is, $|u| \leq r$, $|v| \leq r$, $|w| \leq r$ for sufficiently small positive r. Is this good enough? The substitution of $1 + 1/u$ in terms of v in Eq. (11) is supposed to be valid for $0 \leq v < \infty$, not only for $|v| \leq r$. Fortunately, it turns out that the value of the integral from 0 to ∞ depends almost entirely on the values of the integrand near zero. In fact, we have (see exercise 4)

$$\int_r^\infty xe^{-xv}\left(1 + \frac{1}{u}\right) dv = O(e^{-rx}) \tag{13}$$

for any fixed $r > 0$, and for large x. We are interested in an approximation up to terms $O(x^{-m})$, and since $O((1/e^r)^x)$ is much smaller than $O(x^{-m})$ for any positive r, m, we need integrate only from 0 to r, for any fixed positive r. We therefore take r to be small enough so that all the power series manipulations done above are justified (cf. Eqs. 1.2.11.1–10 and 12).

Now

$$\int_0^\infty xe^{-xv}v^\alpha\, dv = \frac{1}{x^\alpha}\int_0^\infty e^{-q}q^\alpha\, dq = \frac{1}{x^\alpha}\Gamma(\alpha + 1), \qquad \text{if} \qquad \alpha > -1; \tag{14}$$

so by putting the series, Eq. (12), into the integral, Eq. (11), we have finally

$$I_1 = \frac{e^{-x}x^x}{\Gamma(x+1)}\left(\sqrt{\frac{\pi}{2}}\,x^{1/2} + \frac{2}{3} + \frac{\sqrt{2\pi}}{24}\,x^{-1/2} - \frac{4}{135}\,x^{-1} + \frac{\sqrt{2\pi}}{576}\,x^{-3/2} + O(x^{-2})\right). \tag{15}$$

Estimate of I_2: In the integral I_2, we substitute $t = u + x$ and obtain

$$I_2 = \frac{e^{-x}x^x}{\Gamma(x+1)}\int_0^y e^{-u}\left(1 + \frac{u}{x}\right)^x du. \tag{16}$$

Now

$$e^{-u}\left(1 + \frac{u}{x}\right)^x = \exp\left(-u + x\ln\left(1 + \frac{u}{x}\right)\right) = \exp\left(\frac{-u^2}{2x} + \frac{u^3}{3x^2} + O(x^{-3})\right)$$
$$= 1 - \frac{u^2}{2x} + \frac{u^4}{8x^2} + \frac{u^3}{3x^2} + O(x^{-3})$$

for $0 \leq u \leq y$ and large x. Therefore we find that

$$I_2 = \frac{e^{-x}x^x}{\Gamma(x+1)}\left(y - \frac{y^3}{6}\,x^{-1} + \left(\frac{y^4}{12} + \frac{y^5}{40}\right)x^{-2} + O(x^{-3})\right). \tag{17}$$

Finally, we analyze the coefficient $e^{-x}x^x/\Gamma(x + 1)$ which appears in both Eqs. (15) and (17). By Stirling's approximation, which is valid for the gamma

function by exercise 1.2.11.2–6, we have

$$\frac{e^{-x}x^x}{\Gamma(x+1)} = \frac{e^{-1/12x+O(x^{-3})}}{\sqrt{2\pi x}}$$

$$= \frac{1}{\sqrt{2\pi}}x^{-1/2} - \frac{1}{12\sqrt{2\pi}}x^{-3/2} + \frac{1}{288\sqrt{2\pi}}x^{-5/2} + O(x^{-7/2}). \quad (18)$$

Now the grand summing up—combining Eqs. (10), (15), (17), and (18), we have

Theorem A. *For large values of x, and fixed y,*

$$\frac{\gamma(x+1, x+y)}{\Gamma(x+1)} = \frac{1}{2} + \left(\frac{y-2/3}{\sqrt{2\pi}}\right)x^{-1/2} + \frac{1}{\sqrt{2\pi}}\left(\frac{23}{270} - \frac{y}{12} - \frac{y^3}{6}\right)x^{-3/2}$$

$$+ O(x^{-5/2}). \ \blacksquare \quad (19)$$

The method we have used shows how this approximation could be extended to further powers of x as far as we please.

This theorem can be used to obtain the approximate values of $R(n)$ and $Q(n)$, by using Eqs. (4) and (9), but we shall defer that calculation until later. Let us now turn to $P(n)$, for which somewhat different methods seem to be required.

$$P(n) = \sum_{0\le k\le n} \frac{k^{n-k}k!}{n!} = \frac{\sqrt{2\pi}}{n!}\sum_{0\le k\le n} k^{n+1/2}e^{-k}\left(1 + \frac{1}{12k} + O(k^{-2})\right). \quad (20)$$

Thus to get the values of $P(n)$, we must study sums of the form

$$\sum_{0\le k\le n} k^{n+1/2}e^{-k}.$$

Let $f(x) = x^{n+1/2}e^{-x}$ and apply Euler's summation formula:

$$\sum_{0\le k\le n} k^{n+1/2}e^{-k} = \int_0^n x^{n+1/2}e^{-x}\,dx + \tfrac{1}{2}n^{n+1/2}e^{-n} + \tfrac{1}{24}n^{n-1/2}e^{-n} - R. \quad (21)$$

Analysis of the remainder (cf. exercise 5) shows that $R = O(n^{n-1/2}e^{-n})$; and since the integral is an incomplete gamma function, we have

$$\sum_{0\le k\le n} k^{n+1/2}e^{-k} = \gamma(n+\tfrac{3}{2}, n) + \tfrac{1}{2}n^{n+1/2}e^{-n} + O(n^{n-1/2}e^{-n}). \quad (22)$$

Our formula, Eq. (20), also requires an estimate of the sum

$$\sum_{0\le k\le n} k^{n-1/2}e^{-k} = \sum_{0\le k\le n-1} k^{(n-1)+1/2}e^{-k} + n^{n-1/2}e^{-n},$$

and this can also be obtained by Eq. (22).

We now have enough formulas at our disposal to determine the approximate values of $P(n)$, $Q(n)$, and $R(n)$, and it is only a matter of substituting and multiplying, etc. In this process we shall have occasion to use the expansion

$$(n+\alpha)^{n+\beta} = n^{n+\beta}e^{\alpha}\left(1+\alpha\left(\beta-\frac{\alpha}{2}\right)\frac{1}{n}+O(n^{-2})\right), \tag{23}$$

which is proved in exercise 6. The method of (21) yields only the first three terms in the asymptotic series for $P(n)$; further terms can be obtained by using the instructive technique described in exercise 14.

The result of all these calculations gives us the desired asymptotic formulas:

$$P(n) = \sqrt{\frac{\pi n}{2}} - \frac{2}{3} + \frac{11}{24}\sqrt{\frac{\pi}{2n}} + \frac{4}{135n} - \frac{71}{1152}\sqrt{\frac{\pi}{2n^3}} + O(n^{-2}), \tag{24}$$

$$Q(n) = \sqrt{\frac{\pi n}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2n}} - \frac{4}{135n} + \frac{1}{288}\sqrt{\frac{\pi}{2n^3}} + O(n^{-2}), \tag{25}$$

$$R(n) = \sqrt{\frac{\pi n}{2}} + \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2n}} + \frac{4}{135n} + \frac{1}{288}\sqrt{\frac{\pi}{2n^3}} + O(n^{-2}). \tag{26}$$

The functions studied here have received only light treatment in the published literature. The first term $\sqrt{\pi n/2}$ in the expansion of $P(n)$ was given by R. Church [see K. Iverson, *A Programming Language* (New York: Wiley, 1962), pp. 239–240]. Using Church's result, a table of $P(n)$ for $n \le 2000$, and a good slide rule, the author deduced an empirical estimate $P(n) \approx \sqrt{\pi n/2} - 0.6667 + 0.575/\sqrt{n}$. It was natural to conjecture that 0.6667 was really an approximation to $\frac{2}{3}$, and that 0.575 would perhaps turn out to be an approximation to $\gamma = 0.57721\ldots$ (why not be optimistic?). Later, as this section was being written, the above expansion of $P(n)$ was developed, and the conjecture $\frac{2}{3}$ was verified; for the 0.575 we have not γ but $\frac{11}{24}\sqrt{\pi/2} \approx 0.574$. This nicely confirms both the theory and the empirical estimates.

Formulas equivalent to the asymptotic values of $Q(n)$ and $R(n)$ were first determined by the brilliant self-taught Indian mathematician S. Ramanujan, who posed the problem of estimating $n!e^n/2n^n - Q(n)$ in *J. Indian Math. Soc.* **3** (1911), 128; **4** (1912), 151–152. In his answer to the problem, he gave the asymptotic series $\frac{1}{3} + \frac{4}{135}n^{-1} - \frac{8}{2835}n^{-2} - \frac{1}{8505}n^{-3} + \ldots$, which goes considerably beyond Eq. (25). His derivation was somewhat more elegant than the method described above; to estimate I_1, he substituted $t = x + u\sqrt{2x}$, and expressed the integrand as a sum of terms of the form $c_{jk}\int_0^\infty \exp(-u^2)u^j n^{-k/2}\,du$. The integral I_2 can be avoided completely, since $a\gamma(a,x) = x^a e^{-x} + \gamma(a+1,x)$ when $a > 0$ (see (8)). The derivation we have used, which is instructive in spite of its unnecessary complications, is due to R. Furch [*Zeitschrift für Physik* **112** (1939), 92–95], who was primarily interested in the value of y which makes $\gamma(x+1, x+y) = \frac{1}{2}\Gamma(x+1)$. For a bibliography of other investigations of $Q(n)$, see H. W. Gould, *AMM* **75** (1968), 1019–1021. The asymptotic properties

of the incomplete gamma function were later extended to complex arguments by F. G. Tricomi ["Asymptotische Eigenschaften der unvollständigen Gammafunktion," *Math. Zeitschrift* **53** (1950), 136–148].

Further study of the functions $P(n)$, $Q(n)$, and $R(n)$ would be interesting. The derivations given above use only simple techniques of elementary calculus; note that we have used different methods for each function!

For detailed information about techniques for solving similar problems, the reader is referred to the book *Asymptotic Methods in Analysis* by N. G. de Bruijn (Amsterdam: North Holland Publ., 1961).

EXERCISES

1. [*HM20*] Prove Eq. (5) by induction on n.
2. [*HM20*] Obtain Eq. (7) from Eq. (6).
3. [*M20*] Derive Eq. (8) from Eq. (7).
4. ▸ [*HM10*] Prove Eq. (13).
5. [*HM24*] Show that R in Eq. (21) is $O(n^{n-1/2}e^{-n})$.
6. ▸ [*HM20*] Prove Eq. (23).
7. ▸ [*HM30*] In the evaluation of I_2, we had to consider

$$\int_0^y e^{-u}\left(1+\frac{u}{x}\right)^x du.$$

Give an asymptotic representation of

$$\int_0^{yx^{1/4}} e^{-u}\left(1+\frac{u}{x}\right)^x du$$

to terms of $O(x^{-2})$, when y is fixed and x is large.

8. [*HM30*] Assume that $0 \le r \le \frac{1}{2}$. Suppose $f(x) = O(x^r)$; show that

$$\int_0^{f(x)} e^{-u}\left(1+\frac{u}{x}\right)^x du = \int_0^{f(x)} \exp\left(\frac{-u^2}{2x}+\frac{u^3}{3x^2}-\cdots+\frac{(-1)^{m-1}u^m}{mx^{m-1}}\right) du + O(x^{-s})$$

if $m = \lceil (s+2r)/(1-r) \rceil$. [This proves in particular a result due to Tricomi: if $f(x) = O(\sqrt{x})$, then

$$\int_0^{f(x)} e^{-u}\left(1+\frac{u}{x}\right)^x du = \sqrt{2x}\int_0^{f(x)/\sqrt{2x}} e^{-t^2}\,dt + O(1).]$$

9. ▸ [*HM36*] What is the behavior of $\gamma(x+1, px)/\Gamma(x+1)$ for large x? (Here p is a real constant; and if $p < 0$, we assume x is an integer, so that t^x is defined for negative t.) Obtain at least two terms of the asymptotic expansion, before resorting to O-terms.

10. [*HM34*] Under the assumptions of the preceding problem, with $p \neq 1$, obtain the asymptotic expansion of

$$\gamma\left(x+1, px+\frac{p}{p-1}y\right) - \gamma(x+1, px),$$

for fixed y, to terms of the same order as obtained in the previous exercise.

▶ **11.** [*HM35*] Let us generalize the functions $Q(n)$, $R(n)$ by introducing a parameter x so that

$$Q_x(n) = 1+\left(\frac{n-1}{n}\right)x+\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)x^2+\cdots,$$

and

$$R_x(n) = 1+\left(\frac{n}{n+1}\right)x+\left(\frac{n}{n+1}\right)\left(\frac{n}{n+2}\right)x^2+\cdots.$$

Explore this situation and find asymptotic formulas when $x \neq 1$.

12. [*HM20*] The function $\int_0^x e^{-t^2/2}\,dt$ which appeared in connection with the normal distribution (see Section 1.2.10) can be expressed as a special case of the incomplete gamma function. Find values of a, b, y such that $b\gamma(a, y)$ equals the above function.

13. [*HM46*] (S. Ramanujan.) Prove that $R(n) - Q(n) = \frac{2}{3} + 8/(135(n+\theta(n))$, where $\frac{2}{21} \leq \theta(n) \leq \frac{8}{45}$. (This implies the much weaker result $R(n+1) - Q(n+1) < R(n) - Q(n)$.)

▶ **14.** [*HM39*] (N. G. de Bruijn.) The purpose of this exercise is to find the asymptotic expansion of $\sum_{0\leq k\leq n} k^{n+\alpha}e^{-k}$ for fixed α, as $n \to \infty$. (a) Replacing k by $n-k$, show that the given sum equals $n^{n+\alpha}e^{-n}\sum_{0\leq k\leq n} e^{-k^2/2n}f(k, n)$, where $f(k, n) = (1-k/n)^\alpha \exp(-k^3/3n^2 - k^4/4n^3 - \cdots)$. (b) Show that for all $m \geq 0$ and $\epsilon > 0$, $f(k, n)$ can be written in the form $\sum_{0\leq i\leq j\leq m} c_{ij}k^{2i+j}n^{-i-j} + O(n^{(m+1)(-1/2+3\epsilon)})$, when $0 \leq k \leq n^{1/2+\epsilon}$. (c) Prove that as a consequence of (b), $\sum_{0\leq k\leq n} e^{-k^2/2n}f(k, n) = \sum_{0\leq i\leq j\leq m} c_{ij}n^{-i-j}\sum_{k\geq 0} k^{2i+j}e^{-k^2/2n} + O(n^{-m/2+\delta})$, for all $\delta > 0$. [Hint: Over the range $n^{1/2+\epsilon} < k < \infty$, the sums are $O(n^{-r})$ for all r.] (d) Show that the asymptotic expansion of $\sum_{k\geq 0} k^t e^{-k^2/2n}$ for fixed $t \geq 0$ can be obtained by Euler's summation formula. (e) Finally therefore

$$\sum_{0\leq k\leq n} k^{n+\alpha}e^{-k} = n^{n+\alpha}e^{-n}\left(\sqrt{\frac{\pi n}{2}} - \frac{1}{6} - \alpha + \left(\frac{1}{12}+\frac{1}{2}\alpha+\frac{1}{2}\alpha^2\right)\sqrt{\frac{\pi}{2n}} + O(n^{-1})\right);$$

this computation can in principle be extended to $O(n^{-r})$ for any desired r.

15. [*HM20*] Show that

$$\int_0^\infty \left(1+\frac{z}{n}\right)^n e^{-z}\,dz$$

is related to $Q(n)$.

16. [*M24*] Prove the identity

$$\sum_k (-1)^k \binom{n}{k} k^{n-1}Q(k) = (-1)^n(n-1)!, \qquad \text{when} \qquad n > 0.$$

1.3. MIX

In many places throughout this book we will have occasion to refer to a computer's "machine language." The machine we use is a mythical computer called "MIX." MIX is very much like nearly every computer now in existence (except that it is, perhaps, nicer). The language of MIX has been designed to be powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned.

The reader is urged to study this section carefully, since MIX language appears in so many parts of this book. There should be no hesitation about learning a new machine language; indeed, the author has found it not uncommon to be writing programs in a half dozen different machine languages during the same week! Everyone with more than a casual interest in computers will probably get to know several different machine languages in the course of his lifetime. MIX has been specially designed to be so much like most existing machine languages that its characteristics are easy to assimilate.

1.3.1. Description of MIX.

MIX is the world's first polyunsaturated computer. Like most machines, it has an identifying number—the 1009. This number was found by taking 16 actual computers which are very similar to MIX and on which MIX can be easily simulated, then averaging their numbers with equal weight:

$$\lfloor (360 + 650 + 709 + 7070 + \mathrm{U3} + \mathrm{SS80} + 1107 + 1604 + \mathrm{G20} + \mathrm{B220} + \mathrm{S2000} + 920 + 601 + \mathrm{H800} + \mathrm{PDP4} + \mathrm{II})/16 \rfloor = 1009. \qquad (1)$$

This number may also be obtained in a more simple way by taking Roman numerals.

MIX has a peculiar property in that it is both binary and decimal at the same time. *The programmer doesn't actually know whether he is programming a machine with base 2 or base 10 arithmetic.* This has been done so that algorithms written in MIX can be used on either type of machine with little change, and so that MIX can be easily simulated on either type of machine. Those programmers accustomed to a binary machine can think of MIX as binary; those accustomed to decimal may regard MIX as decimal. Programmers from another planet might choose to think of MIX as a ternary machine.

Words. The basic unit of information is a *byte.* Each byte contains an *unspecified* amount of information, but it must be capable of holding at least 64 distinct values. That is, we know that any number between 0 and 63, inclusive, can be contained in one byte. Furthermore, each byte contains *at most* 100 distinct values. On a binary computer a byte must therefore be composed of six bits; on a decimal computer we have two digits per byte.

Programs expressed in the MIX language should be written so that no more than sixty-four values are ever assumed for a byte. If we wish to treat the

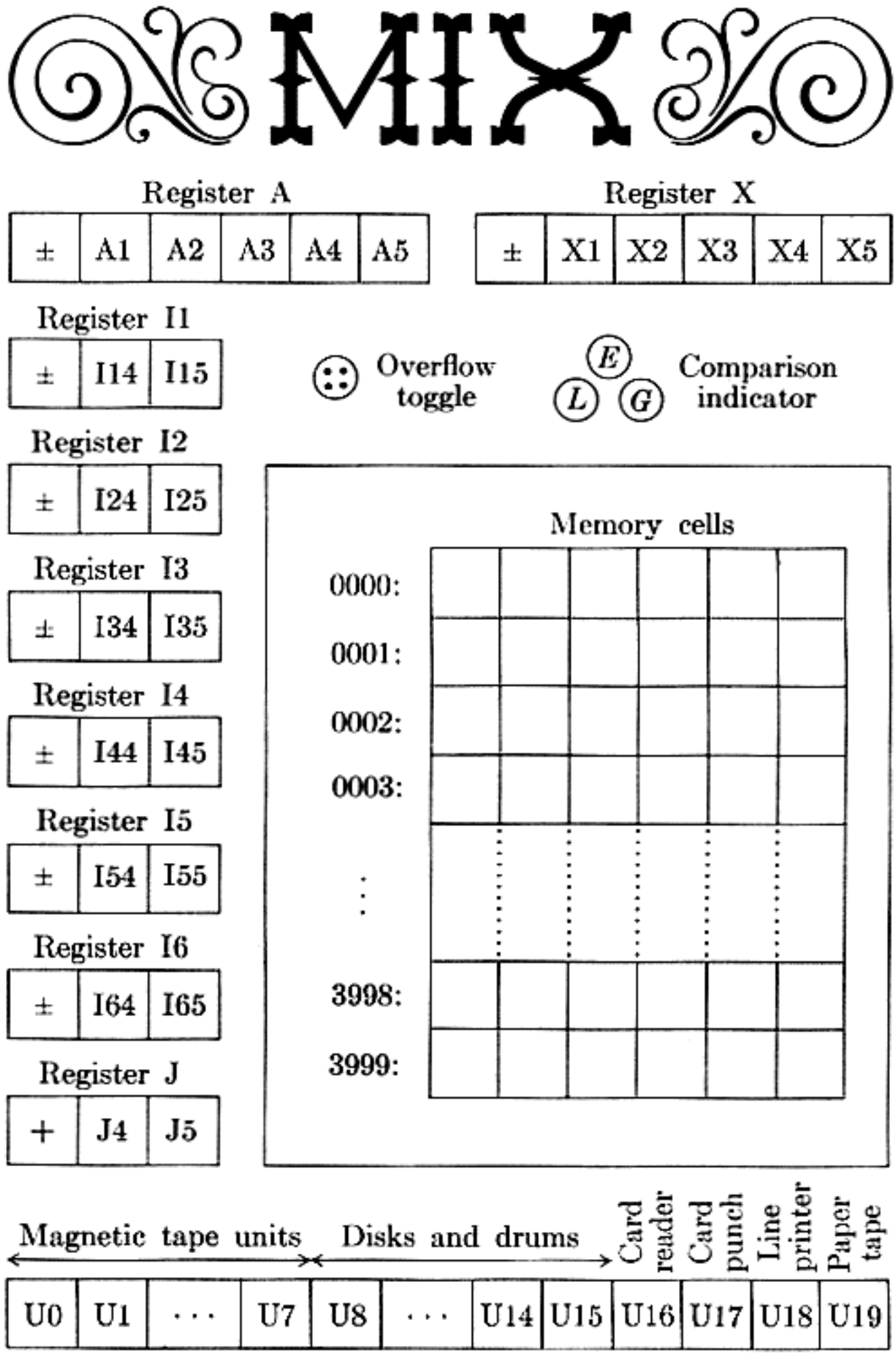


Fig. 13. The MIX computer.

number 80, we should always leave two adjacent bytes for expressing it, even though on a decimal computer one byte is sufficient. *An algorithm in* MIX *should work properly regardless of how big a byte is.* Although it is quite possible to write programs which depend on the byte size, this is an illegal act which will not be tolerated; the only legitimate programs are those which would give correct results with all byte sizes. It is not hard to abide by this ground rule, and we will thereby find that programming a decimal computer isn't so different from programming a binary one after all.

Two adjacent bytes can express the numbers 0 through 4095.
Three adjacent bytes can express the numbers 0 through 262143.
Four adjacent bytes can express the numbers 0 through 16777215.
Five adjacent bytes can express the numbers 0 through 1073741823.

A computer word is five bytes plus a sign. The sign position has only two possible values, + and −.

Registers. There are nine registers in MIX (see Fig. 13):

The A-register (Accumulator) is five bytes plus sign.

The X-register (Extension) is also five bytes plus sign.

The I-registers (Index registers) I1, I2, I3, I4, I5, and I6 each hold two bytes plus sign.

The J-register (Jump address) holds two bytes, and its sign is always +.

We shall use a small letter "r" prefixed to the name, to identify a MIX register. Thus, "rA" means "register A."

The A-register has many uses, especially for arithmetic and operating on data. The X-register is an extension on the "right-hand side" of rA, and it is used in connection with rA to hold ten bytes of a product or dividend, or it can be used to hold information shifted to the right out of rA. The index registers rI1, rI2, rI3, rI4, rI5, and rI6 are used primarily for counting and for referencing variable memory addresses. The J-register always holds the address of the instruction following the preceding "JUMP" instruction, and it is primarily used in connection with subroutines.

Besides these registers, MIX contains

an *overflow toggle* (a single bit which is either "on" or "off"),
a *comparison indicator* (which has three values: less, equal, or greater),
memory (4000 words of storage, each word with five bytes plus sign),
and *input-output devices* (card, tape, etc.).

Partial fields of words. The five bytes and sign of a computer word are numbered as follows:

0	1	2	3	4	5
±	Byte	Byte	Byte	Byte	Byte

(2)

Most of the instructions allow the programmer to use only part of a word if he chooses. In this case a "field specification" is given. The allowable fields are those which are adjacent in a computer word, and they are represented by (L:R), where L is the number of the left-hand part and R is the number of the right-hand part of the field. Examples of field specifications are:

(0:0), the sign only.
(0:2), the sign and the first two bytes.
(0:5), the whole word. This is the most common field specification.
(1:5), the whole word except for the sign.
(4:4), the fourth byte only.
(4:5), the two least significant bytes.

The use of these field specifications varies slightly from instruction to instruction, and it will be explained in detail for each instruction where it applies.

Although it is generally not important to the programmer, the field (L:R) is denoted in the machine by the single number 8L + R, and this number will fit in one byte.

Instruction format. Computer words used for instructions have the following form:

0	1	2	3	4	5
±	A	A	I	F	C

. (3)

The rightmost byte, C, is the operation code telling what operation is to be performed. For example, C = 8 is the operation `LDA`, "load the A register."

The F-byte holds a modification of the operation code. F is usually a field specification (L:R) = 8L + R; for example, if C = 8 and F = 11, the operation is "load the A-register with the (1:3) field." Sometimes F is used for other purposes; on input-output instructions, for example, F is the number of the affected input or output unit.

The left-hand portion of the instruction, ±AA, is the "address." (Note that the sign is part of the address.) The I-field, which comes next to the address, is the "index specification," which may be used to modify the address of an instruction. If I = 0, the address ±AA is used without change; otherwise I should contain a number i between 1 and 6, and the contents of index register Ii are added algebraically to ±AA; the result is used as the address of the instruction. This indexing process takes place on *every* instruction. We will use the letter M to indicate the address after any specified indexing has occurred. (If the addition of the index register to the address ±AA yields a result which does not fit in two bytes, the value of M is undefined.)

In most instructions, M will refer to a memory cell. The terms "memory cell" and "memory location" are used almost interchangeably in this book. We assume that there are 4000 memory cells, numbered from 0 to 3999; hence every memory location can be addressed with two bytes. For every instruction in which M is to refer to a memory cell we must have $0 \leq M \leq 3999$, and in this case we will write `CONTENTS`(M) to denote the value stored in memory location M.

On certain instructions, the "address" M has another significance, and it may even be negative. Thus one instruction adds M to an index register, and this takes account of the sign of M.

Notation. To discuss instructions in a readable manner, we will use the notation

```
OP   ADDRESS,I(F)
```
(4)

to denote an instruction like (3). Here `OP` is a symbolic name which is given to the operation code (the C-part) of the instruction; `ADDRESS` is the ±AA portion; and `I`, `F` represent the I- and F-fields, respectively.

If I is zero, the ",I" is omitted. If F is the *normal* F-specification for this particular operator, the "(F)" need not be written. The normal F-specification for almost all operators is (0:5), representing a whole word. If a different F is standard, it will be mentioned explicitly when we discuss a particular operator.

For example, the instruction to load a number into the accumulator is called LDA and it is operation code number 8. We have

Conventional representation	Actual numeric instruction				
LDA 2000,2(0:3)	+	2000	2	3	8
LDA 2000,2(1:3)	+	2000	2	11	8
LDA 2000(1:3)	+	2000	0	11	8
LDA 2000	+	2000	0	5	8
LDA -2000,4	−	2000	4	5	8

(5)

To render these in words, the instruction "LDA 2000,2(0:3)" may be read "Load A with the contents of location 2000 indexed by 2, the zero-three field."

To represent the numerical contents of a MIX word, we will always use a box notation like that above. Note that in the word

+	2000	2	3	8

the number +2000 is shown filling two adjacent bytes and sign; the actual contents of byte (1:1) and of byte (2:2) will vary from one MIX computer to another, since byte size is variable. As a further example of this notation for MIX words, the diagram

−	10000	3000

represents a word with two fields, a three-byte-plus-sign field containing -10000 and a two-byte field containing 3000. When a word is split into more than one field, it is said to be "packed."

Rules for each instruction. The remarks following (3) above have defined the quantities M, F, and C for every word used as an instruction. We will now define the actions corresponding to each instruction.

Loading operators

• LDA (load A). C = 8; F = field.

The specified field of CONTENTS(M) replaces the previous contents of register A.

On all operations where a partial field is used as an input, the sign is used if it is a part of the field, otherwise the sign + is understood. The field is shifted over to the right-hand part of the register as it is loaded.

Examples: If F is the normal field specification (0:5), the entire contents of location M is loaded. If F is (1:5), the absolute value of CONTENTS(M) is loaded with a plus sign. If M contains an *instruction* word and if F is (0:2), the "±AA" field is loaded as

±	0	0	0	A	A

.

Suppose location 2000 contains the word

–	80	3	5	4

; (6)

then we get the following results from loading various partial fields:

Instruction	Contents of rA afterwards					
LDA 2000	–	80		3	5	4
LDA 2000(1:5)	+	80		3	5	4
LDA 2000(3:5)	+	0	0	3	5	4
LDA 2000(0:3)	–	0	0	80		3
LDA 2000(4:4)	+	0	0	0	0	5
LDA 2000(0:0)	–	0	0	0	0	0
LDA 2000(1:1)	+	0	0	0	0	?

(The last example has a partially unknown effect since byte size is variable.)

- LDX (load X). C = 15; F = field.

This is the same as LDA, except that rX is loaded instead of rA.

- LD*i* (load *i*). C = 8 + *i*; F = field.

This is the same as LDA, except that rI*i* is loaded instead of rA. An index register contains only two bytes (not five) plus sign; bytes 1, 2, 3 are always assumed to be zero. The LD*i* instruction is considered undefined if it would result in setting bytes 1, 2, 3 to anything but zero.

In the description of all instructions, "*i*" stands for an integer, $1 \le i \le 6$. Thus, LD*i* stands for six different instructions: LD1, LD2, . . . , LD6.

- LDAN (load A negative). C = 16; F = field.
- LDXN (load X negative). C = 23; F = field.
- LD*i*N (load *i* negative). C = 16 + *i*; F = field.

These eight instructions are the same as LDA, LDX, LD*i*, respectively, except that the *opposite* sign is loaded.

Storing operators.

- STA (store A). C = 24; F = field.

The contents of rA replaces the field of CONTENTS(M) specified by F. The other parts of CONTENTS(M) are unchanged.

On a *store* operation the field F has the opposite significance from the *load* operation. The number of bytes in the field is taken from the right-hand side of the register and shifted *left* if necessary to be inserted in the proper field of CONTENTS(M). The sign is not altered unless it is part of the field. The contents of the register is not affected.

Examples: Suppose that location 2000 contains

−	1	2	3	4	5

and register A contains

+	6	7	8	9	0

.

Then:

Instruction	Contents of location 2000 afterwards					
STA 2000	+	6	7	8	9	0
STA 2000(1:5)	−	6	7	8	9	0
STA 2000(5:5)	−	1	2	3	4	0
STA 2000(2:2)	−	1	0	3	4	5
STA 2000(2:3)	−	1	9	0	4	5
STA 2000(0:1)	+	0	2	3	4	5

- STX (store X). C = 31; F = field.

Same as STA except rX is stored rather than rA.

- ST*i* (store *i*). C = 24 + *i*; F = field.

Same as STA except rI*i* is stored rather than rA. Bytes 1, 2, 3 of an index register are zero; thus if rI1 contains

±	*m*	*n*

,

this behaves as though it were

±	0	0	0	*m*	*n*

.

- STJ (store J). C = 32; F = field.

Same as ST*i* except rJ is stored, and its sign is always +.

On STJ *the normal field specification for F is* (0:2), *not* (0:5). This is natural, since STJ is almost always done into the address field of an instruction.

- STZ (store zero). C = 33; F = field.

Same as STA except plus zero is stored. In other words, the specified field of CONTENTS(M) is cleared to zero.

Arithmetic operators. On the add, subtract, multiply, and divide operations, a field specification is allowed. A field specification of "(0:6)" can be used to indicate a "floating-point" operation (see Section 4.2), but few of the programs we will write for MIX will use this feature; floating-point instructions will be used primarily in the programs written by the compilers discussed in Chapter 12.

The standard field specification is, as usual, (0:5). Other fields are treated as in LDA. We will use the letter V to indicate the specified field of CONTENTS(M); thus, V is the value which would have been loaded into register A if the operation code were LDA.

• ADD. C = 1; F = field.
V is added to rA. If the magnitude of the result is too large for register A, the overflow toggle is set on, and the remainder of the addition appearing in rA is as though a "1" had been carried into another register to the left of A. (Otherwise the setting of the overflow toggle is unchanged.) If the result is zero, the sign of rA is unchanged.

Example: The sequence of instructions below gives the sum of the five bytes of register A.

```
STA  2000
LDA  2000(5:5)
ADD  2000(4:4)
ADD  2000(3:3)
ADD  2000(2:2)
ADD  2000(1:1)
```

This is sometimes called "sideways addition."

• SUB (subtract). C = 2; F = field.
V is subtracted from rA. Overflow may occur as in ADD.

Note that because of the variable definition of byte size, overflow will occur in some MIX computers when it would not occur in others. We have not said that overflow will occur definitely if the value is greater than 1073741823; overflow occurs when the magnitude of the result is greater than the contents of five bytes, depending on the byte size. One can still write programs which work properly and which give the same final answers, regardless of the byte size.

• MUL (multiply). C = 3; F = field.
The 10-byte product of V times (rA) replaces registers A and X. The signs of rA and rX are both set to the algebraic sign of the result (i.e., + if the signs of V and rA were the same, and − if they were different).

• DIV (divide). C = 4; F = field.
The value of rA and rX, treated as a 10-byte number, with the sign of rA, is divided by the value V. If V = 0 or if the quotient is more than five bytes in magnitude (this is equivalent to the condition that $|rA| \geq |V|$), registers A and X are filled with undefined information and the overflow toggle is set on. Otherwise the quotient is placed in rA and the remainder is placed in rX. The sign of rA afterward is the algebraic sign of the quotient; the sign of rX afterward is the previous sign of rA.

Examples of arithmetic instructions: In most cases, arithmetic is done only with MIX words which are single five-byte numbers, not packed with several fields. It is possible to operate arithmetically on packed MIX words, if some caution is used. The following examples should be studied carefully. (The "?" mark designates an unknown value.)

+	1234	1	150		rA before
+	100	5	50		Cell 1000
+	1334	6	200		rA after

ADD 1000

−	1234	0	0	9	rA before
−	2000	150		0	Cell 1000
+	766	149		?	rA after

SUB 1000

−					112	rA before
?	2	?	?	?	?	Cell 1000
−					0	rA after
−					224	rX after

MUL 1000(1:1)

−	50	0	112		4	rA before
−	2	0	0	0	0	Cell 1000
+	100		0	224		rA after
+	8	0	0	0	0	rX after

MUL 1000

+					0	rA before
+					17	rX before
+					3	Cell 1000
+					5	rA after
+					2	rX after

DIV 1000

−					0	rA before
+	1235		0	3	0	rX before
−	0	0	0	2	0	Cell 1000
+	0	617		?	?	rA after
−	0	0	0	?	?	rX after

DIV 1000

(These examples have been prepared with the philosophy that it is better to give a complete, baffling description than an incomplete, straightforward one.)

Address transfer operators. In the following operations, the (possibly indexed) "address" M is used as a signed number, not as the address of a cell in memory.

- ENTA (enter A). C = 48; F = 2.

The quantity M is loaded into rA. The action is equivalent to "LDA" from a memory word containing the signed value of M. If M = 0, the sign of the instruction is loaded.

Examples: "ENTA 0" sets rA to zeros. "ENTA 0,1" sets rA to the current contents of index register 1.

- ENTX (enter X). C = 55; F = 2.
- ENTi (enter i). C = 48 + i; F = 2.

Analogous to ENTA, loading the appropriate register.

- ENNA (enter negative A). C = 48; F = 3.
- ENNX (enter negative X). C = 55; F = 3.
- ENNi (enter negative i). C = 48 + i; F = 3.

Same as ENTA, ENTX, and ENTi, except that the opposite sign is loaded.

Example: "ENN3 0,3" replaces rI3 by its negative.

- INCA (increase A). C = 48; F = 0.

The quantity M is added to rA; the action is equivalent to "ADD" from a memory word containing the value of M. Overflow is possible and it is treated just as in ADD.

Example: "INCA 1" increases the value of rA by one.

- INCX (increase X). C = 55; F = 0.

The quantity M is added to rX. If overflow occurs, the action is equivalent to ADD, except that rX is used instead of rA. Register A is never affected by this instruction.

- INCi (increase i). $C = 48 + i$; $F = 0$.

Add M to rIi. Overflow must not occur; if the magnitude of the result is more than two bytes, the result of this instruction is undefined.

- DECA (decrease A). C = 48; F = 1.
- DECX (decrease X). C = 55; F = 1.
- DECi (decrease i). C = 48 + i; F = 1.

These eight instructions are the same as INCA, INCX, and INCi, respectively, except that M is subtracted from the register rather than added.

Note that the operation code C is the same for ENTA, ENNA, INCA, and DECA; the F-field is used to distinguish the various operations in this case.

Comparison operators. The comparison operators all compare the value contained in a register with a value contained in memory. The comparison indicator

is then set to LESS, EQUAL, or GREATER according to whether the value of the *register* is less than, equal to, or greater than the value of the *memory cell*. A minus zero is *equal to* a plus zero.

- CMPA (compare A). C = 56; F = field.

The specified field of A is compared with the *same* field of CONTENTS(M). If the field F does not include the sign position, the fields are both thought of as positive; otherwise the sign is taken into account in the comparison. (If F is (0:0) an equal comparison always occurs, since minus zero equals plus zero.)

- CMPX (compare X). C = 63; F = field.

This is analogous to CMPA.

- CMPi (compare i). $C = 56 + i$; F = field.

Analogous to CMPA. Bytes 1, 2, and 3 of the index register are treated as zero in the comparison.

Jump operators. Ordinarily, instructions are executed in sequential order; i.e., the instruction executed after the one in location P is the instruction found in location P + 1. Several "jump" instructions allow this sequence to be interrupted. Whenever a jump of any kind takes place, the J-register is set to the address of the next instruction (that is, the address of the instruction which would have been next if we hadn't jumped). A "store J" instruction then can be used by the programmer, if desired, to set the address field of another command which will later be used to return to the original place in the program. The J-register is changed whenever a jump actually occurs in a program (except JSJ) and it is never changed except when a jump occurs.

- JMP (jump). C = 39; F = 0.

Unconditional jump: the next instruction is taken from location M.

- JSJ (jump, save J). C = 39; F = 1.

Same as JMP except that the contents of rJ are unchanged.

- JOV (jump on overflow). C = 39; F = 2.

If the overflow toggle is on, it is turned off and a JMP occurs; otherwise nothing happens.

- JNOV (jump on no overflow). C = 39; F = 3.

If the overflow toggle is off, a JMP occurs; otherwise it is turned off.

- JL, JE, JG, JGE, JNE, JLE (jump on less, equal, greater, greater-or-equal, unequal, less-or-equal). C = 39; F = 4, 5, 6, 7, 8, 9, respectively.

Jump if the comparison indicator is set to the condition indicated. For example, JNE will jump if the comparison indicator is LESS or GREATER. The comparison indicator is not changed by these instructions.

- JAN, JAZ, JAP, JANN, JANZ, JANP (jump A negative, zero, positive, nonnegative, nonzero, nonpositive). C = 40; F = 0, 1, 2, 3, 4, 5, respectively.

If the contents of rA satisfy the stated condition, a JMP occurs, otherwise nothing

happens. "Positive" means *greater* than zero (not zero); "nonpositive" means the opposite, i.e., zero or negative.

- JXN, JXZ, JXP, JXNN, JXNZ, JXNP (jump X negative, zero, positive, nonnegative, nonzero, nonpositive). C = 47; F = 0, 1, 2, 3, 4, 5, respectively.

- J*i*N, J*i*Z, J*i*P, J*i*NN, J*i*NZ, J*i*NP (jump *i* negative, zero, positive, nonnegative, nonzero, nonpositive). C = 40 + *i*; F = 0, 1, 2, 3, 4, 5, respectively.
These are analogous to the corresponding operations for rA.

Miscellaneous operators.

- MOVE. C = 7; F = number.
The number of words specified by F is moved, starting from location M to the location specified by the contents of index register 1. The transfer occurs one word at a time, and rI1 is increased by the value of F at the end of the operation. If F = 0, nothing happens.

Care must be taken when the groups of locations involved overlap; for example, suppose that F = 3 and M = 1000. Then if (rI1) = 999, we transfer (1000) to (999), (1001) to (1000), and (1002) to (1001). Nothing unusual occurred here; but if (rI1) were 1001 instead, we would move (1000) to (1001), then (1001) to (1002), then (1002) to (1003), so we have moved the *same* word (1000) into three places.

- SLA, SRA, SLAX, SRAX, SLC, SRC (shift left A, shift right A, shift left AX, shift right AX, shift left AX circularly, shift right AX circularly). C = 6; F = 0, 1, 2, 3, 4, 5, respectively.

These are the "shift" commands. Signs of registers A, X are not affected in any way. M specifies the number of *bytes* to be shifted left or right; M must be nonnegative. SLA and SRA do not affect rX; the other shifts affect both registers as though they were a single 10-byte register. With SLA, SRA, SLAX, and SRAX, zeros are shifted into the register at one side, and bytes disappear at the other side. The instructions SLC and SRC call for a "circulating" shift, in which the bytes that leave one end enter in at the other end. Both rA and rX participate in a circulating shift.

Examples:

	Register A						Register X					
Initial contents	+	1	2	3	4	5	−	6	7	8	9	10
SRAX 1	+	0	1	2	3	4	−	5	6	7	8	9
SLA 2	+	2	3	4	0	0	−	5	6	7	8	9
SRC 4	+	6	7	8	9	2	−	3	4	0	0	5
SRA 2	+	0	0	6	7	8	−	3	4	0	0	5
SLC 501	+	0	6	7	8	3	−	4	0	0	5	0

• NOP (no operation). C = 0.
No operation occurs, and this instruction is bypassed. F and M are ignored.

• HLT (halt). C = 5; F = 2.
The machine stops. When the computer operator restarts it, the net effect is equivalent to NOP.

Input-output operators. MIX has a fair amount of input-output equipment (all of which is optional at extra cost). Each device is given a number as follows:

Unit number	Peripheral device	Record size
t	Tape unit no. t ($0 \le t \le 7$)	100 words
d	Disk or drum unit no. d ($8 \le d \le 15$)	100 words
16	Card reader	16 words
17	Card punch	16 words
18	Printer	24 words
19	Typewriter and paper tape	14 words

Not every MIX installation will have all of this equipment available; we will occasionally make appropriate assumptions about the presence of certain devices. Some devices may not be used both for input and for output. The number of words mentioned in the above table is a fixed record size associated with each unit.

Input or output with magnetic tape, disk, or drum units reads or writes full words (five bytes plus sign). Input or output with cards or typewriter, however, is always done in a *character code* where each byte represents one alphameric character. Thus, five characters per MIX word are transmitted. The character code is given at the top of Table 1, which appears at the close of this section and on the end papers of this book. The code 00 corresponds to "␣", which denotes a *blank space*. Codes 01–29 are for the letters A through Z with a few Greek letters thrown in; codes 30–39 represent the digits 0, 1, . . . , 9; and further codes 40, 41, . . . represent punctuation marks and other special characters. It is not possible to read in or write out all possible values a byte may have, since certain combinations are undefined. Not all input-output devices are capable of handling all the symbols in the character set; for example, the symbols Φ and Π which appear amid the letters will perhaps not be acceptable to the card reader. When input of character code is being done, the signs of all words are set to "+"; on output, signs are ignored.

The disk and drum units are large external memory devices each containing b^2 100-word records, where b is the byte size. On every IN, OUT, or IOC instruction as defined below, the particular 100-word record referred to by the instruction is specified by the current contents of the two least significant bytes of rX.

• IN (input). C = 36; F = unit.
This instruction initiates the transfer of information from the input unit specified into sequential locations starting with M. The number of locations transferred

is the record size for this unit (see the table above). The machine will wait at this point if a preceding operation for the same unit is not yet complete. The transfer of information which starts with this instruction will not be complete until a "jump ready" or "jump busy" instruction (see below) indicates the unit is again ready, so a program must not refer to the information in memory until this time. It is improper to attempt to read any record from magnetic tape which follows the latest record written on that tape.

• `OUT` (output). C = 37; F = unit.
This instruction starts the transfer of information from memory locations starting at M to the output unit specified. (The machine waits until the unit is ready, if it is not initially ready.) The transfer will not be complete until a "jump ready" or "jump busy" instruction (see below) indicates the unit is again ready, so a program must not alter the information in memory before this time.

• `IOC` (input-output control). C = 35; F = unit.
The machine waits, if necessary, until the specified unit is not busy. Then a control operation is performed, depending on the particular device being used. The following examples are used in various parts of this book:

Magnetic tape: If M = 0, the tape is rewound. If M < 0 the tape is skipped backward −M records, or to the beginning of the tape, whichever comes first. If M > 0, the tape is skipped forward; it is improper to skip forward over any records following the one last written on that tape.

For example, the sequence "`OUT 1000(3); IOC -1(3); IN 2000(3)`" writes out one hundred words onto tape 3, then reads it back in again. Unless the tape reliability is questioned, the last two instructions of that sequence are only a slow way to move words 1000–1099 to locations 2000–2099. The sequence "`OUT 1000(3); IOC +1(3)`" is improper.

Disk or drum: M should be zero. The effect is to position the device according to rX so that the next `IN` or `OUT` operation on this unit will take less time if it uses the same rX setting.

Printer: M should be zero. "`IOC 0(18)`" skips the printer to the top of the following page.

Paper tape reader: Rewind the tape. (M should be zero.)

• `JRED` (jump ready). C = 38; F = unit.
A jump occurs if the specified unit is ready, i.e., finished with the preceding operation initiated by `IN`, `OUT`, or `IOC`.

• `JBUS` (jump busy). C = 34; F = unit.
Same as `JRED` except the jump occurs under the opposite circumstances, i.e., when the specified unit is *not* ready.

Example: In location 1000, the instruction "`JBUS 1000(16)`" will be executed repeatedly until unit 16 is ready.

The simple operations above complete `MIX`'s repertoire of input-output instructions. There is no "tape check" indicator, etc., to cover exceptional

conditions on the peripheral devices. Any such condition (e.g., paper jam, unit turned off, out of tape, etc.) causes the unit to remain busy, a bell rings, and the skilled computer operator fixes things manually using ordinary maintenance procedures.

Conversion Operators.

• NUM (convert to numeric). C = 5; F = 0.
This operation is used to change the character code into numeric code. M is ignored. Registers A, X are assumed to contain a 10-byte number in character code; the NUM instruction sets the magnitude of rA equal to the numerical value of this number (treated as a decimal number). The value of rX and the sign of rA are unchanged. Bytes 00, 10, 20, 30, 40, . . . convert to the digit zero; bytes 01, 11, 21, . . . convert to the digit one; etc. Overflow is possible, and in this case the remainder modulo the word size is retained.

• CHAR (convert to characters). C = 5; F = 1.
This operation is used to change numeric code into character code suitable for output to cards or printer. The value in rA is converted into a 10-byte decimal number which is put into register A and X in character code. The signs of rA, rX are unchanged. M is ignored.

Examples:

	Register A						Register X					
Initial contents	–	00	00	31	32	39	+	37	57	47	30	30
NUM 0	–				12977700		+	37	57	47	30	30
INCA 1	–				12977699		+	37	57	47	30	30
CHAR 0	–	30	30	31	32	39	+	37	37	36	39	39

Timing. To give quantitative information as to how "good" MIX programs are, each of MIX's operations is assigned an *execution time* typical for present day computers.

ADD, SUB, all LOAD operations, all STORE operations (including STZ), all shift commands, and all comparison operations take *two units* of time. MOVE requires one unit plus two for each word moved. MUL requires 10 and DIV requires 12 units. Execution time for floating-point operations is unspecified. All remaining operations take one unit of time, plus the time the computer may be idle on the IN, OUT, IOC, or HLT instructions.

Note in particular that ENTA takes one unit of time, while LDA takes two units. The timing rules are easily remembered because of the fact that, except for shifts, MUL, and DIV, the number of units equals the number of references to memory (including the reference to the instruction itself).

The "unit" of time is a relative measure which we will denote simply by u. It may be regarded as, say, 10 microseconds (for a relatively inexpensive computer) or as 1 microsecond (for a relatively high-priced machine).

Example: Execution of the sequence

```
LDA  1000
INCA 1
STA  1000
```

takes a time of $5u$.

Summary. We have now discussed all of the features of MIX, except for its "GO button" which is discussed in exercise 26. Although MIX has nearly 150 different operations, they fit into a few simple patterns so they can be easily remembered. Table 1 summarizes the operations for each C-setting. The name of each operator is followed by the standard F-field for that operator in parentheses.

The following exercises give a good review of the material in this section; most of them are very simple, and the reader should try to do nearly all of them.

EXERCISES

1. [*00*] If MIX were a ternary (base 3) computer, how many "trits" would there be per byte?

2. [*02*] If a value to be represented within MIX may get as large as 99999999, how many adjacent bytes should be used to contain this quantity?

3. [*02*] Give the partial field specifications, (L:R), for the (a) address field, (b) index field, (c) field field, and (d) operation code field of a MIX instruction.

4. [*00*] The last example in (5) is "LDA -2000,4"—how can this be legitimate in view of the fact that memory addresses should not be negative?

5. [*10*] What is the symbolic notation [as in (4)] corresponding to the word (6)?

▶ **6.** [*10*] Assume that location 3000 contains

+	5	1	200	15

.

What is the result of the following instructions? (State if any of these are undefined or only partially defined.) (a) LDAN 3000; (b) LD2N 3000(3:4); (c) LDX 3000(1:3); (d) LD6 3000; (e) LDXN 3000(0:0).

7. [*15*] Give a precise definition of the results of the DIV instruction for all cases in which overflow does not occur, using the algebraic operations $X \bmod Y$ and $\lfloor X \rfloor$.

8. [*15*] The last example of the DIV instruction which appears in the text has "rX before" equal to

+	1235	0	3	0

.

If this were

−	1234	0	3	0

instead, but other parts of that example were unchanged, what would registers A, X contain after the DIV instruction?

Table 1

Character code:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
⊔	A	B	C	D	E	F	G	H	I	Θ	J	K	L	M	N	O	P	Q	R	Φ	Π	S	T	U

00	*1*	**01**	*2*	**02**	*2*	**03**	*10*
No operation NOP(0)		rA ← rA + V ADD(0:5) FADD(6)		rA ← rA − V SUB(0:5) FSUB(6)		rAX ← rA × V MUL(0:5) FMUL(6)	
08	*2*	**09**	*2*	**10**	*2*	**11**	*2*
rA ← V LDA(0:5)		rI1 ← V LD1(0:5)		rI2 ← V LD2(0:5)		rI3 ← V LD3(0:5)	
16	*2*	**17**	*2*	**18**	*2*	**19**	*2*
rA ← −V LDAN(0:5)		rI1 ← −V LD1N(0:5)		rI2 ← −V LD2N(0:5)		rI3 ← −V LD3N(0:5)	
24	*2*	**25**	*2*	**26**	*2*	**27**	*2*
F(M) ← rA STA(0:5)		F(M) ← rI1 ST1(0:5)		F(M) ← rI2 ST2(0:5)		F(M) ← rI3 ST3(0:5)	
32	*2*	**33**	*2*	**34**	*1*	**35**	$1 + T$
F(M) ← rJ STJ(0:2)		F(M) ← 0 STZ(0:5)		Unit F busy? JBUS(0)		Control, unit F IOC(0)	
40	*1*	**41**	*1*	**42**	*1*	**43**	*1*
rA:0, jump JA[+]		rI1:0, jump J1[+]		rI2:0, jump J2[+]		rI3:0, jump J3[+]	
48	*1*	**49**	*1*	**50**	*1*	**51**	*1*
rA ← [rA]? ± M INCA(0)DECA(1) ENTA(2)ENNA(3)		rI1 ← [rI1]? ± M INC1(0)DEC1(1) ENT1(2)ENN1(3)		rI2 ← [rI2]? ± M INC2(0)DEC2(1) ENT2(2)ENN2(3)		rI3 ← [rI3]? ± M INC3(0)DEC3(1) ENT3(2)ENN3(3)	
56	*2*	**57**	*2*	**58**	*2*	**59**	*2*
rA(F):V → CI CMPA(0:5) FCMP(6)		rI1(F):V → CI CMP1(0:5)		rI2(F):V → CI CMP2(0:5)		rI3(F):V → CI CMP3(0:5)	

General form:

C	*t*
Description OP(F)	

C = operation code, (5:5) field of instruction
F = op variant, (4:4) field of instruction
M = address of instruction after indexing
V = F(M) = contents of F field of location M
OP = symbolic name for operation
(F) = standard F setting
t = execution time; T = interlock time

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9	.	,	(	)	+	−	*	/	=	$	<	>	@	;	:	'

04	*12*	05	*1*	06	*2*	07	*1 + 2F*
rA ← rAX/V rX ← remainder DIV(0:5) FDIV(6)		Special NUM(0) CHAR(1) HLT(2)		Shift M bytes SLA(0) SRA(1) SLAX(2) SRAX(3) SLC(4) SRC(5)		Move F words from M to rI1 MOVE(1)	
12	*2*	**13**	*2*	**14**	*2*	**15**	*2*
rI4 ← V LD4(0:5)		rI5 ← V LD5(0:5)		rI6 ← V LD6(0:5)		rX ← V LDX(0:5)	
20	*2*	**21**	*2*	**22**	*2*	**23**	*2*
rI4 ← −V LD4N(0:5)		rI5 ← −V LD5N(0:5)		rI6 ← −V LD6N(0:5)		rX ← −V LDXN(0:5)	
28	*2*	**29**	*2*	**30**	*2*	**31**	*2*
F(M) ← rI4 ST4(0:5)		F(M) ← rI5 ST5(0:5)		F(M) ← rI6 ST6(0:5)		F(M) ← rX STX(0:5)	
36	*1 + T*	**37**	*1 + T*	**38**	*1*	**39**	*1*
Input, unit F IN(0)		Output, unit F OUT(0)		Unit F ready? JRED(0)		Jumps JMP(0) JSJ(1) JOV(2)JNOV(3) also [*] below	
44	*1*	**45**	*1*	**46**	*1*	**47**	*1*
rI4:0, jump J4[+]		rI5:0, jump J5[+]		rI6:0, jump J6[+]		rX:0, jump JX[+]	
52	*1*	**53**	*1*	**54**	*1*	**55**	*1*
rI4 ← [rI4]? ± M INC4(0)DEC4(1) ENT4(2)ENN4(3)		rI5 ← [rI5]? ± M INC5(0)DEC5(1) ENT5(2)ENN5(3)		rI6 ← [rI6]? ± M INC6(0)DEC6(1) ENT6(2)ENN6(3)		rX ← [rX]? ± M INCX(0)DECX(1) ENTX(2)ENNX(3)	
60	*2*	**61**	*2*	**62**	*2*	**63**	*2*
rI4(F):V → CI CMP4(0:5)		rI5(F):V → CI CMP5(0:5)		rI6(F):V → CI CMP6(0:5)		rX(F):V → CI CMPX(0:5)	

rA = register A
rX = register X
rAX = registers AX as one
rIi = index reg. i, $1 \leq i \leq 6$
rJ = register J
CI = comparison indicator

[*]:		[+]:
JL(4)	<	N(0)
JE(5)	=	Z(1)
JG(6)	>	P(2)
JGE(7)	≧	NN(3)
JNE(8)	≠	NZ(4)
JLE(9)	≦	NP(5)

▶ **9.** [*20*] List all the MIX operators which can possibly affect the setting of the overflow toggle. (Do not include floating-point operators.)

10. [*20*] List all the MIX operators which can possibly affect the setting of the comparison indicators.

▶ **11.** [*20*] List all the MIX operators which can possibly affect the setting of rI1.

12. [*10*] Find a single instruction which has the effect of multiplying the current contents of rI3 by two and leaving the result in rI3.

▶ **13.** [*10*] Suppose location 1000 contains the instruction "JOV 1001". This instruction turns off the overflow toggle if it is on (and the next instruction executed will be in location 1001, in any case). If this instruction were changed to "JNOV 1001", would there be any difference? What if it were changed to "JOV 1000" or "JNOV 1000"?

14. [*25*] For each MIX operator, consider whether there is a way to set the ±AA-, I-, and F-portions of the instruction so that the result of the instruction is precisely equivalent to NOP, except that the execution time may be longer. Assume that nothing is known about the contents of any registers or any memory locations. Whenever it is possible to produce a NOP, state how it can be done. *Examples:* INCA is a no-op if the address and index parts are zero. JMP can never be a no-op, since it affects rJ.

15. [*10*] How many *alphameric characters* are there in a typewriter record? in a card-reader or card-punch record? in a printer record?

16. [*20*] Write a program which sets memory cells 0000–0099 all to zero, and which is (a) as short a program as possible; (b) as fast a program as possible. [*Hint:* Consider using the MOVE command.]

17. [*26*] This is the same as the previous exercise, except that locations 0000 through N, inclusive, are to be set to zero, where N is the current contents of rI2. The programs should work for any value $0 \leq N \leq 2999$; they should start in location 3000.

▶ **18.** [*22*] After the following "number one" program has been executed, what changes to registers, toggles, and memory have taken place? (For example, what is the final setting of rI1? of rX? of the overflow and comparison indicators?)

```
STZ  1
ENNX 1
STX  1(0:1)
SLAX 1
ENNA 1
INCX 1
ENT1 1
SRC  1
ADD  1
DEC1 -1
STZ  1
CMPA 1
MOVE -1,1(1)
NUM  1
CHAR 1
HLT  1
```

▶ **19.** [*14*] What is the execution time of the program in the preceding exercise, not counting the HLT instruction?

20. [*20*] Write a program which sets *all 4000* memory cells equal to a "HLT" instruction, and then stops.

▶ **21.** [*24*] (a) Can the J-register ever be zero? (b) Write a program which, given a number N in rI4, sets register J equal to N, assuming $0 < N \leq 3000$. Your program should start in location 3000. When your program has finished its execution, the contents of all memory cells must be unchanged.

▶ **22.** [*28*] Location 2000 contains an integer number, X. Write two programs which compute X^{13} and halt with the result in register A. One program should use the minimum number of MIX memory locations; the other should require the minimum execution time possible. Assume that X^{13} fits into a single word.

23. [*27*] Location 0200 contains a word

+	*a*	*b*	*c*	*d*	*e*

;

write two programs which compute the "reflected" word

+	*e*	*d*	*c*	*b*	*a*

and halt with the result in register A. One program should do this without using the "partial field" feature of MIX. Both programs should take the minimum possible number of memory locations under the stated conditions (including those locations used for the program and for temporary storage of intermediate results).

24. [*21*] Assuming registers A and X contain

+	0	*a*	*b*	*c*	*d*

and

+	*e*	*f*	*g*	*h*	*i*

,

respectively, write two programs which change the contents of these registers to

+	*a*	*b*	*c*	*d*	*e*

and

+	0	*f*	*g*	*h*	*i*

,

respectively, using (a) minimum memory space and (b) minimum execution time.

▶ **25.** [*30*] Suppose that the manufacturer of MIX wishes to come out with a more powerful computer ("Mixmaster"?), and he wants to convince as many as possible of those people now owning a MIX computer to invest in the more expensive machine. He wants to design this new hardware to be an *extension* of MIX, in the sense that all programs correctly written for MIX will work on the new machines without change. Suggest desirable things which could be incorporated in this extension. (For example, can you make better use of the I-field of an instruction?)

▶ **26.** [*32*] This problem is to write a card-loading routine. Every computer has its own peculiar problems for getting information initially into the machine and correctly started up, etc. In MIX's case, the contents of a card can only be read in character code,

and this *includes* the cards which contain the loading program itself. Not all possible byte values can be read from a card, and each word read in from cards is positive.

MIX has one feature that has not been explained in the text: There is a "GO-button," which is used to get the computer started from scratch when its memory contains arbitrary information. When this button is pushed by the computer operator, the following actions take place:

a) A single card is read into locations 0000–0015; this is essentially equivalent to the instruction "IN 0(16)".

b) When the card has been completely read and the card reader is no longer busy, a JMP to location 0000 occurs. The J-register is also set to zero.

c) The machine now begins to execute the program which it has read from the card.

(Note: Those MIX computers without card readers have their GO-button attached to the paper tape reader, unit 19, but in this problem we will assume the presence of a card reader, unit 16.)

The loading routine to be written must satisfy the following conditions:

a) The input deck begins with the loading routine, followed by information cards containing the numbers to be loaded, then a "transfer card" which shuts down the loading routine and jumps to the beginning of the program. The loading routine must fit onto *two cards*. You are to design a suitable transfer card.

b) The information cards have the following format:

Columns 1–5, ignored by the loading routine.
Column 6, the number of consecutive words to be loaded on this card (1 through 7).
Columns 7–10, the location of word 1, which is always greater than 100 (so it does not overlay the loading routine).
Columns 11–20, word 1.
Columns 21–30, word 2 (if column 6 $\geq$ 2).
. . .
Columns 71–80, word 7 (if column 6 = 7).

The information for word 1, word 2, . . . , is punched numerically as a decimal number. If the word is to be negative, a minus ("11-punch") is *overpunched* over the least significant digit, e.g., in column 20. Assume that this causes the character code input to be 10, 11, 12, . . . , 19, rather than 30, 31, 32, . . . , 39. For example, a card which has

ABCDE3100001234567890000000001000000010$\bar{0}$

punched in columns 1–40, should cause the following information to be loaded:

1000: +0123456789; 1001: +0000000001; 1002: −0000000100.

c) The loading routine should work for all byte sizes without any changes to the cards bearing the loading routine. No card should contain any of the characters corresponding to bytes 20, 21, 49, 50, . . . (i.e., the characters Φ, Π, $, <, . . .) since these characters cannot be read by all card readers. In particular, the instructions ENT1 and INC1 cannot be used (C = 49) since they cannot be punched on the card.

1.3.2. The MIX Assembly Language

A symbolic language is used to make MIX programs considerably easier to read and to write, and to save the programmer from worrying about tedious clerical details which often lead to unnecessary errors. This language, MIXAL ("MIX Assembly Language"), is an extension of the notation used for instructions in the previous section; the main features of this extension are the optional use of alphabetic names to stand for numbers, and a location field for associating names with memory locations.

MIXAL can be readily comprehended if we consider first a simple example. The following code is part of a larger program; it is a subroutine to find the maximum of n elements $X[1], \ldots, X[n]$, according to Algorithm 1.2.10M.

Program M (*Find the maximum*). Register assignments: rA $\equiv m$, rI1 $\equiv n$, rI2 $\equiv j$, rI3 $\equiv k$, $X[i] \equiv$ cell(X $+ i$).

	Assembled instructions					Line no.	LOC	OP	ADDRESS	Times	Remarks
						01	X	EQU	1000		
						02		ORIG	3000		
3000:	+	3009	0	2	32	*03*	MAXIMUM	STJ	EXIT	1	Subroutine linkage
3001:	+	0	1	2	51	*04*	INIT	ENT3	0,1	1	*M1. Initialize.* $k \leftarrow n$.
3002:	+	3005	0	0	39	*05*		JMP	CHANGEM	1	$j \leftarrow n$, $m \leftarrow X[n]$, $k \leftarrow n - 1$.
3003:	+	1000	3	5	56	*06*	LOOP	CMPA	X,3	$n - 1$	*M3. Compare.*
3004:	+	3007	0	7	39	*07*		JGE	*+3	$n - 1$	
3005:	+	0	3	2	50	*08*	CHANGEM	ENT2	0,3	$A + 1$	*M4. Change m.* $j \leftarrow k$.
3006:	+	1000	3	5	08	*09*		LDA	X,3	$A + 1$	$m \leftarrow X[k]$.
3007:	+	1	0	1	51	*10*		DEC3	1	n	*M5. Decrease k.*
3008:	+	3003	0	2	43	*11*		J3P	LOOP	n	*M2. All tested?*
3009:	+	3009	0	0	39	*12*	EXIT	JMP	*	1	Return to main program. ▮

This program is an example of several things simultaneously:

a) The columns headed "LOC OP ADDRESS" are of principal interest; they contain a program in the MIXAL symbolic machine language, and we shall explain the details of this program below.

b) The column headed "Assembled instructions" shows the actual numeric machine language which corresponds to the MIXAL program. MIXAL has been designed so that it is a relatively simple matter to translate any MIXAL program into numeric machine language; this process may be carried out by another computer program called an *assembly program*. Thus, a programmer may do all of his "machine language" programming in MIXAL, never bothering to determine the equivalent numeric machine language himself. Virtually all MIX programs in this book are written in MIXAL. Chapter 9 includes a complete description of an assembly program which converts MIXAL programs to machine language in a form that is readily loaded into MIX's memory.

c) The column headed "Line no." is not an essential part of the MIXAL program; it is merely incorporated with the MIXAL programs of this book so that the text can refer to parts of the program.

d) The column headed "Remarks" gives explanatory information about the program, and it is cross-referenced to the steps of Algorithm 1.2.10M. The reader should refer to this algorithm. Note that a little "programmer's license" was used during the transcription of that algorithm into a MIX program; for example, step M2 has been put last. Note also the "register assignments" stated at the beginning of Program M; this shows what components of MIX correspond to the variables in the algorithm.

e) The column headed "Times" will be given for many of the MIX programs in this book; it represents the number of times the instruction on that line will be executed during the course of the program. Thus, line 6 will be performed $n - 1$ times, etc. From this information we can determine the length of time required to perform the subroutine; it is $(5 + 5n + 3A)u$, where A is the quantity which was carefully analyzed in Section 1.2.10.

Now we will discuss the MIXAL part of Program M. Line 1, "X EQU 1000", says that symbol X is to be *equivalent* to the number 1000. The effect of this may be seen on line 6, where the numeric equivalent of the instruction "CMPA X,3" appears as

+	1000	3	5	56

i.e., "CMPA 1000,3".

Line 2 says that the locations for succeeding lines should be chosen sequentially, originating with 3000. Therefore the symbol MAXIMUM which appears in the LOC field of line 3 becomes equivalent to the number 3000, INIT is equivalent to 3001, LOOP is equivalent to 3003, etc.

On lines 3 through 12 the OP field contains the symbolic names of MIX instructions STJ, ENT3, etc. The "OP" in lines 1 and 2, on the other hand, contains "EQU" and "ORIG" which are *not* MIX operators. They are called *pseudo-operators* because they appear only in the MIXAL symbolic program. Pseudo-operators are used to specify the form of a symbolic program; they are not instructions of the program itself. Thus the line "X EQU 1000" only talks *about* the program, it does not signify that any variable is to be set equal to 1000 when Program M is run. Note that no instruction is assembled for lines 1 and 2.

Line 3 is a "store J" instruction which stores the contents of register J into the (0:2) field of location "EXIT", i.e., into the address part of the instruction found on line 12.

As mentioned earlier, Program M is intended to be part of a larger program; elsewhere the sequence

```
ENT1 100
JMP  MAXIMUM
STA  MAX
```

would; for example, jump to Program M with n set to 100. Program M would then find the largest of the elements $X[1], \ldots, X[100]$ and would return to the

instruction "STA MAX" with the maximum value in rA and with its position, j, in rI2. (Cf. exercise 3.)

Line 5 jumps the control to line 8. Lines 4, 5, 6 need no further explanation. Line 7 introduces a new notation: an asterisk (read "self") refers to the location of this line; "*+3" ("self plus three") therefore refers to three locations past the current line. Since line 7 is an instruction which corresponds to location 3004, "*+3" appearing there refers to location 3007.

The rest of the symbolic code is self-explanatory; note the appearance of an asterisk again on line 12. (Cf. exercise 2.)

Our next example shows a few more features of the assembly language. The object is to print a table of the first 500 prime numbers, with 10 columns of 50 numbers each. The table should appear as follows:

```
FIRST FIVE HUNDRED PRIMES
     0002 0233 0547 0877 1229 1597 1993 2371 2749 3187
     0003 0239 0557 0881 1231 1601 1997 2377 2753 3191
     0005 0241 0563 0883 1237 1607 1999 2381 2767 3203
       :                                            :
     0229 0541 0863 1223 1583 1987 2357 2741 3181 3571
```

We shall use the following method.

Algorithm P (*Print table of 500 primes*). This algorithm has two distinct parts: steps P1–P8 prepare an internal table of 500 primes, and steps P9–P11 print the answer in the form shown above. The latter part of the program uses two "buffer" areas, i.e., sections of memory in which a line image is formed; while one buffer is being printed, the other is being filled.

P1. [Start table.] Set PRIME[1] $\leftarrow 2$, N $\leftarrow 3$, J $\leftarrow 1$. (N will run through the odd numbers which are candidates for primes; J keeps track of how many primes have been found so far.)

P2. [N is prime.] Set J $\leftarrow$ J $+ 1$, PRIME[J] $\leftarrow$ N.

P3. [500 found?] If J $= 500$, go to step P9.

P4. [Advance N.] Set N $\leftarrow$ N $+ 2$.

P5. [K $\leftarrow 2$.] Set K $\leftarrow 2$. (PRIME[K] will run through the possible prime divisors of N.)

P6. [PRIME[K]\N?] Divide N by PRIME[K]; let Q be the quotient and R the remainder. If R $= 0$, N is not prime, so go to P4.

P7. [PRIME[K] large?] If Q $\leq$ PRIME[K], go to P2. (In such a case, N must be prime; the proof of this fact is interesting and a little unusual—see exercise 6.)

P8. [Advance K.] Increase K by 1, and go to P6.

P9. [Print title.] Now we are ready to print the table. Advance the printer to the next page. Set BUFFER[0] to the title line and print this line. Set B ← 1, M ← 1.

P10. [Set up line.] Put PRIME[M], PRIME[50 + M], . . . , PRIME[450 + M] in proper format into BUFFER[B].

P11. [Print line.] Print BUFFER[B]; set B ← 1 − B (thereby switching to the other buffer); and increase M by 1. If M ≤ 50, return to P10; otherwise the algorithm terminates. ▮

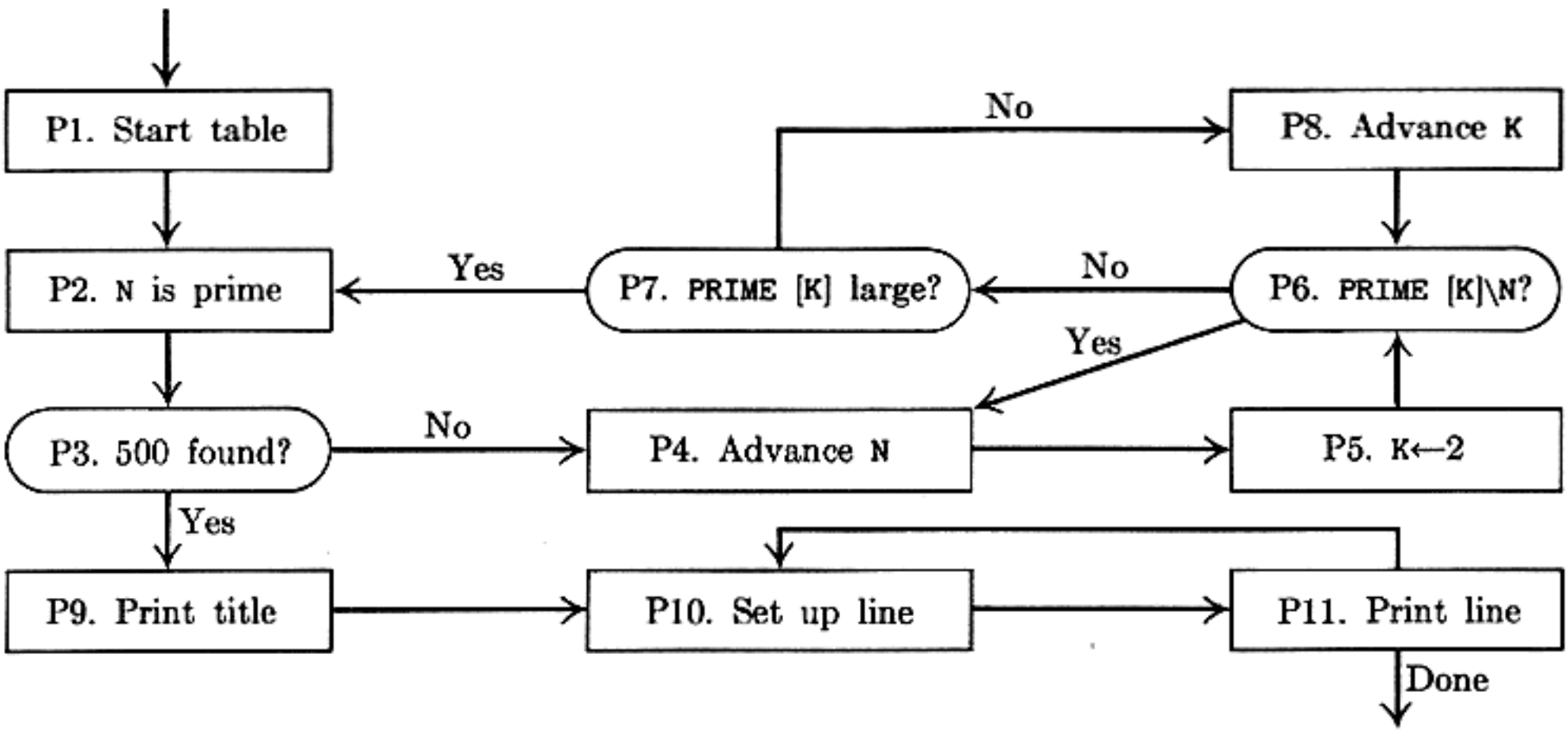


Fig. 14. Algorithm P.

Program P (*Print table of 500 primes*). This program has deliberately been written in a slightly clumsy fashion in order to illustrate most of the features of MIXAL in a single program. rI1 ≡ J − 500; rI2 ≡ N; rI3 ≡ K; rI4 indicates B; rI5 is M plus multiples of 50.

```
01    * EXAMPLE PROGRAM ... TABLE OF PRIMES
02    *
03    L        EQU   500              Number of primes to find
04    PRINTER  EQU   18               Unit number of printer
05    PRIME    EQU   -1               Memory area for table of primes
06    BUF0     EQU   2000             Memory area for BUFFER[0]
07    BUF1     EQU   BUF0+25          Memory area for BUFFER[1]
08             ORIG  3000
09    START    IOC   0(PRINTER)       Skip to new page.
10             LD1   =1-L=            P1. Start table. J ← 1.
11             LD2   =3=                  N ← 3.
12    2H       INC1  1                P2. N is prime. J ← J + 1.
13             ST2   PRIME+L,1            PRIME[J] ← N.
14             J1Z   2F               P3. 500 found?
```

```
15    4H      INC2  2                  P4.  Advance N.
16            ENT3  2                  P5.  K ← 2.
17    6H      ENTA  0                  P6.  PRIME[K]\N?
18            ENTX  0,2
19            DIV   PRIME,3
20            JXZ   4B                       R = 0?
21            CMPA  PRIME,3            P7.  PRIME[K] large?
22            INC3  1                  P8.  Advance K.
23            JG    6B                       Jump if Q > PRIME[K].
24            JMP   2B                       Otherwise N is prime.
25    2H      OUT   TITLE(PRINTER)     P9.  Print title.
26            ENT4  BUF1+10                  Set B ← 1.
27            ENT5  -50                      Set M ← 0.
28    2H      INC5  L+1                      Advance M.
29    4H      LDA   PRIME,5            P10. Set up line. (Right to left)
30            CHAR
31            STX   0,4(1:4)
32            DEC4  1
33            DEC5  50                       (rI5 goes down by 50 until
34            J5P   4B                           nonpositive)
35            OUT   0,4(PRINTER)       P11. Print line.
36            LD4   24,4                     Switch buffers.
37            J5N   2B                       If rI5 = 0, we are done.
38            HLT
39    * INITIAL CONTENTS OF TABLES AND BUFFERS
40            ORIG  PRIME+1
41            CON   2                  First prime is 2.
42            ORIG  BUF0-5
43    TITLE   ALF   FIRST              Alphabetic information for
44            ALF    FIVE                  title line
45            ALF    HUND
46            ALF   RED P
47            ALF   RIMES
48            ORIG  BUF0+24
49            CON   BUF1+10            Each buffer refers to the other.
50            ORIG  BUF1+24
51            CON   BUF0+10
52            END   START              End of routine. ▮
```

The following points of interest are to be noted about this program:

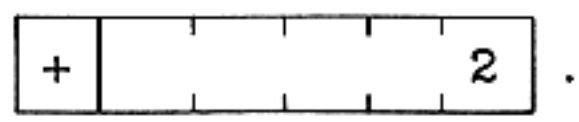
1. Lines 01, 02, and 39 begin with an asterisk: this signifies a "comment" line which is merely explanatory, having no actual effect on the assembled program.

2. As in Program M, the "EQU" in line 03 sets the equivalent of a symbol; in this case, the equivalent of L is set to 500. (In the program of lines 10–24,

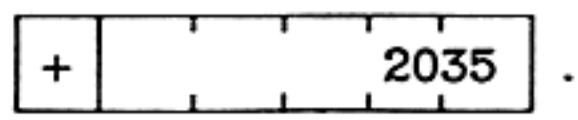
L represents the number of primes to be computed.) Note that in line 05 the symbol PRIME gets a *negative* equivalent; the equivalent of a symbol may be any five-byte-plus-sign number. In line 07 the equivalent of BUF1 is calculated as BUF0+25, namely 2025. MIXAL provides a limited amount of arithmetic on numbers; for another example, see line 13 where the value of PRIME+L (in this case, 499) is calculated by the assembly program.

3. Note that the symbol PRINTER has been used in the F-part on lines 25 and 35. The F-part, which is always enclosed in parentheses, may be a single number or symbol, or two numbers and/or symbols separated by a colon, as in the "1:4" of line 31.

4. MIXAL contains several ways to specify non-instruction words. Line 41 indicates an ordinary constant, "2", using the operation code CON; the result of line 41 is to assemble the word



Line 49 shows a slightly more complicated constant, "BUF1+10", which assembles as the word



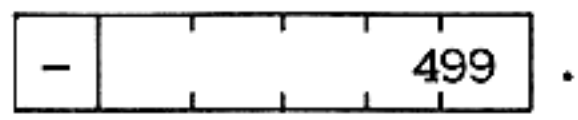
A constant may be enclosed in equal signs and it then becomes a *literal constant* (see lines 10 and 11). The assembler automatically creates internal names and inserts "CON" lines for literal constants. For example, lines 10 and 11 of Program P would effectively be changed to

```
10        LD1  con1
11        LD2  con2
```

and then at the end of the program, between lines 51 and 52, the lines

```
51a  con1  CON  1-L
51b  con2  CON  3
```

are effectively inserted as part of the assembly procedure for literal constants. Line 51a will assemble into the word



The use of literal constants is a decided convenience, because it means that the programmer does not have to invent a name for the constant and that he does not have to insert that constant at the end of the program; he can keep his mind on the central problems and not worry about such routine matters while writing his programs. Of course, in Program P we did not make an

especially good use of literal constants, since lines 10 and 11 would more properly be written "ENT1 1-L; ENT2 3"!

5. A good assembly language should mimic the way a programmer *thinks* about writing programs, so he can express himself fluently. One example of this philosophy is the use of literal constants, as we have just mentioned; another example is the use of "*", which was explained in Program M. A third example is the idea of *local symbols* such as the symbol 2H, which appears in the location field of lines 12, 25, and 28.

Local symbols are special symbols whose equivalents can be *redefined* as many times as desired. A symbol like PRIME has but one significance throughout a program, and if it were to appear in the location field of more than one line an error would be indicated by the assembly program. Local symbols have a different nature; we write, for example, 2H ("2 here") in the location field, and 2F ("2 forward") or 2B ("2 backward") in the address field of a MIXAL line:

2B means the closest *previous* location 2H
2F means the closest *following* location 2H

As examples, the "2F" in line 14 refers to line 25; the "2B" in line 24 refers back to line 12; and the "2B" in line 37 refers to line 28. An address of 2F or 2B never refers to the *same* line; e.g., the three lines

```
2H  EQU  10
2H  MOVE 2F(2B)
2H  EQU  2B-3
```

are virtually equivalent to the single line

```
MOVE *-3(10).
```

The symbols 2F, 2B are never to be used in the location field, and 2H is never to be used in the address field. There are ten local symbols, which can be obtained by replacing "2" in the above examples by any digit from 0 to 9.

The idea of local symbols was introduced by M. E. Conway in 1958, in connection with an assembly program for the UNIVAC 1. Local symbols spare the programmer from the necessity to think of a symbolic name for an address, when all he wants to do is refer to an instruction a few lines away. When making reference to a nearby location in the program there often is no appropriate name with much significance, so programmers have tended to use symbols like X1, X2, X3, etc.; this leads to the danger of using the same symbol twice. That is why the reader will soon find that the use of local symbols comes naturally to him when he writes MIXAL programs, if he is not already familiar with this idea.

6. In lines 30 and 38 the address part is blank. This means the address is to be zero. Similarly, we could have left the address blank in line 17, but the program would have been less readable.

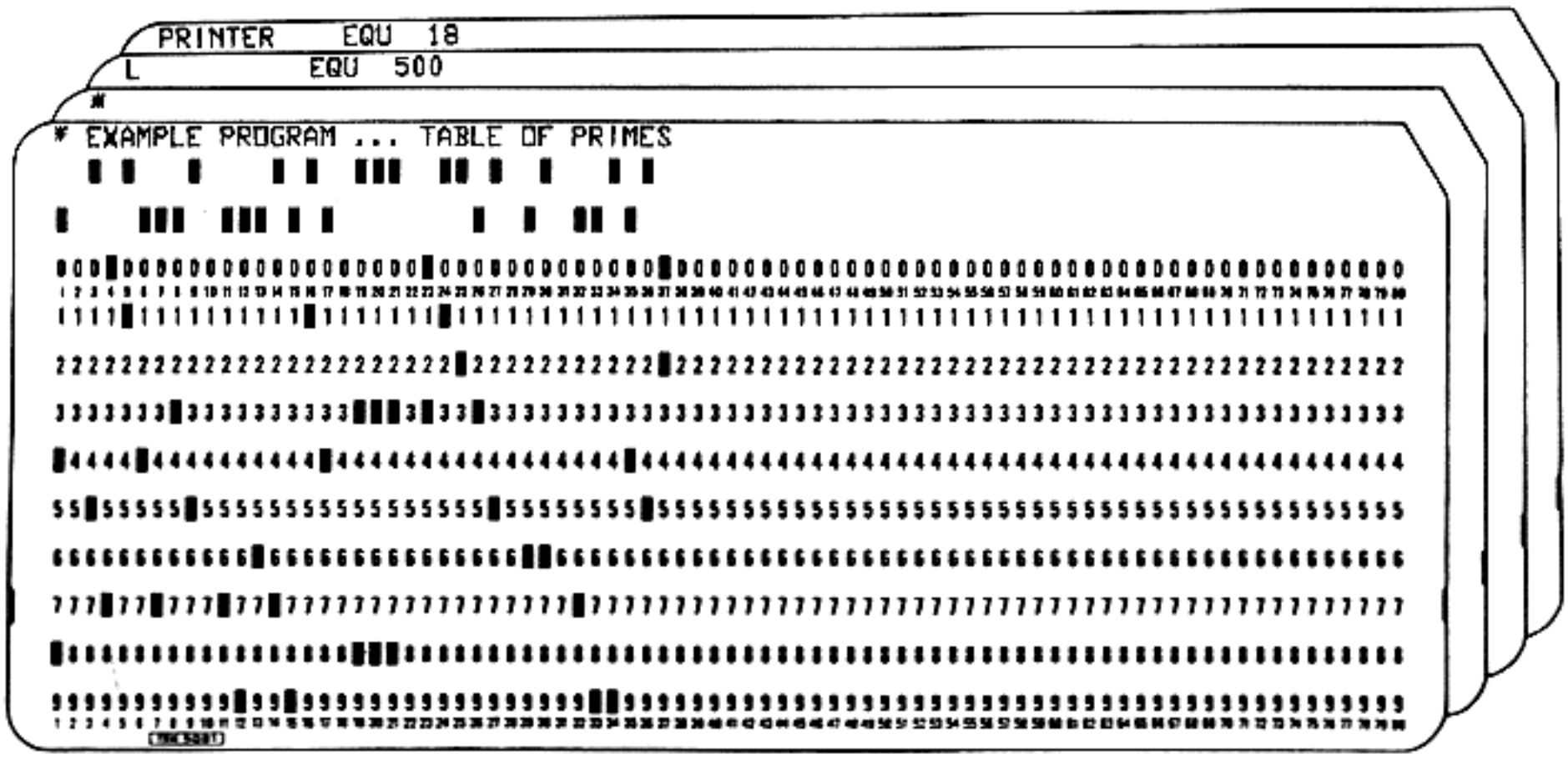


Fig. 15. The first four lines of Program M punched onto cards.

7. Lines 43–47 use the "ALF" operation, which creates a five-byte constant in MIX alphameric character code. For example, line 45 causes the word

+	00	08	24	15	04

to be assembled.

These lines are used as the first 25 characters of the title line. *All locations whose contents are not specified in the* MIXAL *program are ordinarily set to zero* (except the locations which are used by the loading routine, usually 3700–3999); thus there is no need to set the other words of the title line to blanks.

8. Note that arithmetic may be used on ORIG lines, e.g., lines 40, 42, and 48.

9. The last line of a complete MIXAL program always has the operation code END. The address on this line is the location at which the program is to begin once it has been loaded into memory.

10. As a final note about Program P, the reader may observe how the coding has been written so that index registers are counted towards zero, and tested against zero, whenever possible. For example, the quantity J–500, not J, is kept in rI1. Lines 26–34 are particularly noteworthy, although perhaps a bit tricky.

It may be of interest to note a few of the statistics observed when Program P was actually run. The division instruction in line 19 was executed 9538 times; the time to perform lines 10–24 was 182144*u*.

MIXAL programs can be punched onto cards, as shown in Fig. 15. The following format is used:

Columns 1–10	LOC (location) field.
Columns 12–15	OP field,
Columns 17–80	ADDRESS field and optional remarks,
Columns 11, 16	blank.

However, if column 1 contains an asterisk, the entire card is treated as a comment. The ADDRESS field ends with the first blank column following column 16; any explanatory information may be punched to the right of this first blank column with no effect on the assembled program. (*Exception:* When the OP field is "ALF", the remarks always start in column 22.)

The MIX assembly program (see Section 9.3) accepts card decks prepared in this manner and converts them to machine language programs in loadable form. Under favorable circumstances the reader will have access to a MIX assembly program and MIX simulator, on which various exercises in this book can be worked out.

Now we have seen what can be done in MIXAL. We conclude this section by describing the rules more carefully, and in particular we shall observe what is *not* allowed in MIXAL. The following comparatively few rules define the language.

1. A *symbol* is a string of one to ten letters and/or digits, containing at least one letter. *Examples:* PRIME TEMP 20BY20. The special symbols *d*H, *d*F, *d*B, where *d* is a single digit, will for the purposes of this definition be replaced by other unique symbols according to the "local symbol" convention described above.

2. A *number* is a string of one to ten digits. *Example:* 00052.

3. Each appearance of a symbol in a MIXAL program is said to be either a "defined symbol" or a "future reference." A *defined symbol* is a symbol which has appeared in the LOC field of a preceding line of this MIXAL program. A *future reference* is a symbol which has not yet been defined in this way.

4. An *atomic expression* is either

a) a number, or
b) a defined symbol (denoting the numerical equivalent of that symbol, see rule 13), or
c) an asterisk (denoting the value of ⊛; see rules 10 and 11).

5. An *expression* is either

a) an atomic expression, or
b) a plus or minus sign followed by an atomic expression, or
c) an expression followed by a binary operation followed by an atomic expression.

The six admissible binary operations are +, −, *, /, //, : ; they are defined on numeric MIX words as follows:

```
C = A+B      LDA A; ADD B; STA C
C = A-B      LDA A; SUB B; STA C
C = A*B      LDA A; MUL B; STX C
C = A/B      LDA A; SRAX 5; DIV B; STA C
C = A//B     LDA A; ENTX 0; DIV B; STA C
C = A:B      LDA A; MUL =8=; SLAX 5; ADD B; STA C.
```

Operations within an expression are carried out from left to right. *Examples:*

`-1+5`	equals 4
`-1+5*20/6`	equals 4*20/6 equals 80/6 equals 13 (going from left to right)
`1//3`	equals a `MIX` word whose value is approximately $(b^5/3)$ where b is the byte size; i.e., a word representing the fraction $\frac{1}{3}$ with decimal point at the left.
`1:3`	equals 11 (usually used in partial field specification)
`*-3`	equals ⊛ minus three
`***`	equals ⊛ times ⊛!

6. An *A-part* (which is used to describe the address field of a `MIX` instruction) is either

a) vacuous (denoting the value zero), or
b) an expression, or
c) a future reference (denoting the eventual equivalent of the symbol, see rule 13).

7. An *index part* (which is used to describe the index field of a `MIX` instruction) is either

a) vacuous (denoting the value zero), or
b) a comma followed by an expression (denoting the value of that expression).

8. An *F-part* (which is used to describe the F-field of a `MIX` instruction) is either

a) vacuous (denoting the *standard* F-setting, based on the context), or
b) a left parenthesis followed by an expression followed by a right parenthesis (denoting the value of the expression).

9. A *W-value* (which is used to describe a *full-word* `MIX` constant) is either

a) an expression followed by an F-part [in this case a vacuous F-part denotes (0:5)], or
b) a W-value followed by a comma followed by a W-value of the form (a).

A W-value denotes the value of a numeric `MIX` word determined as follows: Let the W-value have the form "$\mathtt{E}_1(\mathtt{F}_1), \mathtt{E}_2(\mathtt{F}_2), \ldots, \mathtt{E}_n(\mathtt{F}_n)$" where $n \geq 1$, the `E`'s are expressions, and the `F`'s are fields. The desired result is the final value which would appear in memory location `CON` if the following hypothetical program were executed: "`STZ CON`; `LDA` $\mathtt{C}_1$; `STA CON(`$\mathtt{F}_1$`)`; . . . ; `LDA` $\mathtt{C}_n$; `STA CON(`$\mathtt{F}_n$`)`". Here $\mathtt{C}_1, \ldots, \mathtt{C}_n$ denote locations containing the values of expressions $\mathtt{E}_1, \ldots, \mathtt{E}_n$. Each $\mathtt{F}_i$ must have the form $8\mathtt{L}_i + \mathtt{R}_i$ where $0 \leq \mathtt{L}_i \leq \mathtt{R}_i \leq 5$. *Examples:*

`1`	is the word	+		1
`1,-1000(0:2)`	is the word	−	1000	1
`-1000(0:2),1`	is the word	+		1

10. The assembly process makes use of a value denoted by ⊛ (called the *location counter*) which is initially zero. The value of ⊛ should always be a

nonnegative number which can fit in two bytes. When the location field of a line is not blank, it must contain a symbol which has not been previously defined. The equivalent of that symbol is then defined to be the current value of ⊛.

11. After processing the LOC field as described in rule 10, the assembly process depends on the value of the OP field. There are six possibilities for OP:

a) OP is a symbolic MIX operator (see Table 1 at the end of the previous section). The chart defines the standard C and F values for this operator. In this case the ADDRESS should be an A-part (rule 6), followed by an index part (rule 7), followed by an F part (rule 8). We thereby obtain four values: C, F, A, and I; the effect is to assemble the word determined by the sequence "LDA C; STA WORD; LDA F; STA WORD(4:4); LDA I; STA WORD(3:3); LDA A; STA WORD(0:2)" into the location specified by ⊛, and to advance ⊛ by 1.

b) OP is "EQU". The ADDRESS should be a W-value (see rule 9); if the LOC field is nonblank, the equivalent of the symbol appearing there is set equal to the value specified in ADDRESS. This rule takes precedence over rule 10. The value of ⊛ is unchanged. (As a nontrivial example, consider the line

```
BYTESIZE  EQU  1(4:4)
```

which allows the programmer to have a symbol whose value depends on the byte size. This is an acceptable situation so long as the resulting program is meaningful with each possible byte size.)

c) OP is "ORIG". The ADDRESS should be a W-value (see rule 9); the location counter, ⊛, is set to this value. (Note that because of rule 10, a symbol appearing in the LOC field of an ORIG card gets as its equivalent the value of ⊛ before it has changed. *Example:*

```
TABLE     ORIG *+100
```

sets the equivalent of TABLE to the *first* of 100 locations.)

d) OP is "CON". The ADDRESS should be a W-value; the effect is to assemble a word, having this value, into the location specified by ⊛, and to advance ⊛ by 1.

e) OP is "ALF". The effect is to assemble the word of character codes formed by columns 17–21 of the card, otherwise behaving like CON.

f) OP is "END". The ADDRESS should be a W-value, which specifies in its (4:5) field the location of the instruction at which the program begins. The END card signals the end of a MIXAL program. Lines are inserted, in arbitrary order, corresponding to all undefined symbols and literal constants (see rules 12 and 13).

12. Literal constants: A W-value of 9 characters or less in length may be enclosed between "=" signs and used as a future reference. The effect is as though a new symbol were created and inserted just before the END card (see remark 4 following Program P).

13. Every symbol has one and only one equivalent value; this is a full-word MIX number which is either determined by the symbol's appearance in LOC according to rule 10 or rule 11(b), or else a line, having the name of the symbol in LOC with OP = "CON" and ADDRESS = "0", is effectively inserted before the END card.

Note: The most significant consequence of the above rules is the restriction on future references. A symbol which has not been defined in the LOC field of a previous card may not be used except as the A-part of an instruction. In particular, it may not be used (a) in connection with arithmetic operations; or (b) in the ADDRESS field of EQU, ORIG, or CON. For example,

```
LDA  2F+1     and     CON  3F
```

are both illegal. This restriction has been imposed in order to allow more efficient assembly of programs, and the experience gained in writing this set of books has shown that it is a very mild restriction which rarely makes much difference.

EXERCISES—First set

1. [*00*] The text remarked that "X EQU 1000" does not indicate any instruction which sets the value of a variable. Suppose that you are writing a MIX program in which you wish to set the value contained in a certain memory cell (whose symbolic name is X) equal to 1000. How could you write this in MIXAL?

▶ 2. [*10*] Line 12 of Program M says "JMP *"; since * denotes the location of the line, why doesn't the program go into an infinite loop, endlessly repeating this instruction?

▶ 3. [*23*] What is the effect of the following program, if it is used in conjunction with Program M?

```
START  IN     X+1
       JBUS   *
       ENT1   100
1H     JMP    MAXIMUM
       LDX    X,1
       STA    X,1
       STX    X,2
       DEC1   1
       J1P    1B
       OUT    X+1(1)
       HLT
       END    START
```

▶ 4. [*25*] Assemble Program P by hand; i.e., what are the actual numerical contents of memory, corresponding to that symbolic program?

5. [*HM46*] Analyze the time required for steps P1–P8 of Algorithm P to compute the first T primes, as a function of T.

6. [*M20*] (a) Show that if n is not prime, n has a divisor d with $1 < d \leq \sqrt{n}$. (b) Use this fact to show that the test in step P7 of Algorithm P proves that N is not prime.

7. [*10*] What is the meaning of "4B" in line 34 of Program P? What effect, if any, would be caused if the location of line 15 were changed to "2H" and the address of line 20 were changed to "2B"?

▶ **8.** [*24*] What does the following program do? (Do not run it on a computer, figure it out by hand!)

```
* MYSTERY PROGRAM
PRINTER  EQU   18
BUF      ORIG  *+3000
1H       ENT1  1
         ENT2  0
         LDX   4F
2H       ENT3  0,1
3H       STZ   BUF,2
         INC2  1
         DEC3  1
         J3P   3B
         STX   BUF,2
         INC2  1
         INC1  1
         CMP1  =75=
         JL    2B
         ENN2  2400
         OUT   BUF+2400,2(PRINTER)
         INC2  24
         J2N   *-2
         HLT
4H       ALF   AAAAA
         END   1B
```

EXERCISES—Second set

These exercises are short programming problems, representing typical computer applications and covering a wide range of techniques. It is recommended that each reader choose a few of these programs, in order to get some experience using MIX as well as a good review of basic programming skills. If desired, these exercises may be worked concurrently as the rest of Chapter 1 is being read.

The following list indicates the types of programming methods which arise in the exercises below:

Use of switching tables (multiway decisions): exercises 9 and 23.
Use of index registers; two-dimensional arrays: exercises 10, 21, 22, and 23.
Unpacking characters: exercises 13 and 23.

Integer and scaled decimal arithmetic: exercises 14, 16 and 18.
Real-time control: exercise 20.
Graphical display: exercise 23.
Input buffering: exercise 13.
Output buffering: exercises 21 and 23.
Use of subroutines: exercises 14 and 20.

Whenever an exercise in this book says, "write a MIX program" or "write a MIX subroutine", it suffices to write only the symbolic code for what is asked, which will only be a fragment of a larger program; perhaps no input or output is done in this fragment, etc. One need only write LOC, OP, and ADDRESS fields of MIXAL lines (possibly also remarks, especially if someone else is to be grading the solutions!), but not the numeric machine language, line no., or "times" columns unless requested to do so.

If the exercise says, "Write a *complete* MIX program," it implies that an executable program is to be written in MIXAL (including in particular the final END card); hopefully, an assembler and MIX simulator on which complete programs can be tested will be available to most readers.

▶ **9.** [*25*] Location INST contains a MIX word which purportedly is a MIX instruction. Write a program which jumps to location GOOD if the word has a valid C-field, valid $\pm$AA-field, valid I-field, and valid F-field, and which jumps to location BAD otherwise. Remember that the test for a valid F-field depends on the C-field; for example, if C = 7 (MOVE), any F-field is acceptable, but if C = 8 (LDA), the F-field must have the form $8L + R$ where $0 \le L \le R \le 5$. The "$\pm$AA"-field is to be considered valid *unless* C specifies an instruction requiring a memory address, I = 0, and $\pm$AA is not a valid memory address.

Note: Inexperienced programmers tend to tackle a problem like this by writing a long series of tests on C, e.g., `LDA C; JAZ 1F; DECA 5; JAN 2F; JAZ 3F; DECA 2; JAN 4F`; etc. This is *not* good practice! Whenever a multiway decision such as this is to be made, it is best to prepare an auxiliary *table* containing information which facilitates the desired decisions. If there were, for example, a table of 64 entries, we could write "`LD1 C; LD1 TABLE,1; JMP 0,1`"—thereby jumping very speedily to the desired routine. Other information can also be kept in such a table. The tabular approach in this case makes the program only a little bit longer (including the table) and it greatly increases the speed.

▶ **10.** [*31*] Assume that we have a 9×8 matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{18} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{28} \\ \vdots & & & & \vdots \\ a_{91} & a_{92} & a_{93} & \ldots & a_{98} \end{pmatrix}$$

stored in memory so that a_{ij} is in location $1000 + 8i + j$. In memory the matrix therefore appears as follows:

$$\begin{pmatrix} (1009) & (1010) & (1011) & \ldots & (1016) \\ (1017) & (1018) & (1019) & \ldots & (1024) \\ \vdots & & & & \vdots \\ (1073) & (1074) & (1075) & \ldots & (1080) \end{pmatrix}.$$

A matrix is said to have a "saddle point" if some position is the smallest value in its row and the largest value in its column. In symbols, a_{ij} is a saddle point if

$$a_{ij} = \min_{1 \le k \le 8} a_{ik} = \max_{1 \le k \le 9} a_{kj}.$$

Write a MIX program which computes

a) the location of a saddle point, if there is at least one;
b) zero, if there is no saddle point;
and which then stops with this value in rI1.

11. [*M29*] What is the *probability* that the matrix in the preceding exercise has a saddle point, assuming that the 72 elements are distinct and assuming that all 72! arrangements are equally probable? What is the probability if we assume instead that the elements of the matrix are zeros and ones, and all 2^{72} such matrices are equally probable?

12. [*M50*] The "answers to the exercises" give two solutions to exercise 10, and suggest a third solution, and it is not clear which of the given solutions is better. Analyze the algorithms, using each of the assumptions of exercise 11, and decide which is the better method.

13. [*28*] A cryptanalyst wants a frequency count of the letters in a certain code. The code has been punched on paper tape; the end is signaled by an asterisk. Write a complete program which reads in the tape, counts the frequency of each character up to the first asterisk, and then types out the results in the form

```
A  0010257
B  0000179
D  0794301
```

etc., one character per line. The number of blanks should not be counted, nor should characters for which the count is zero (e.g., C in the above) be printed. For efficiency, "buffer" the input, i.e., while reading a record into one area of memory you can be counting characters from another area. You may assume that an extra record (following that which contains the terminating asterisk) is present on the input tape.

▶ **14.** [*31*] The following algorithm, due to the Neapolitan astronomer Aloysius Lilius and the German Jesuit mathematician Christopher Clavius in the late 16th century, is used by most Western churches to determine the date of Easter Sunday for any year after 1582. [For previous years, see *CACM* **5** (1962), 209–210. The first systematic algorithm for calculating the date of Easter was the *canon paschalis* due to Victorius of Aquitania (457 A.D.). There are many indications that the sole important application of arithmetic in Europe during the Middle Ages was the calculation of Easter date, and so such algorithms are historically significant. For further commentary, see *Puzzles and Paradoxes* by T. H. O'Beirne, (London: Oxford University Press, 1965), Chapter 10.]

Algorithm E. (*Date of Easter*). Let Y be the year for which the date of Easter is desired.

E1. [Golden number.] Set $G \leftarrow (Y \bmod 19) + 1$. ($G$ is the so-called "golden number" of the year in the 19-year Metonic cycle.)

E2. [Century.] Set $C \leftarrow \lfloor Y/100 \rfloor + 1$. (When Y is not a multiple of 100, C is the century number; i.e., 1970 is in the twentieth century.)

E3. [Corrections.] Set $X \leftarrow \lfloor 3C/4 \rfloor - 12$, $Z \leftarrow \lfloor (8C + 5)/25 \rfloor - 5$. ($X$ is the number of years, such as 1900, in which leap year was dropped in order to keep in step with the sun. Z is a special correction designed to synchronize Easter with the moon's orbit.)

E4. [Find Sunday.] Set $D \leftarrow \lfloor 5Y/4 \rfloor - X - 10$. [March $((-D) \bmod 7)$ actually will be a Sunday.]

E5. [Epact.] Set $E \leftarrow (11G + 20 + Z - X) \bmod 30$. If $E = 25$ and the golden number G is greater than 11, or if $E = 24$, then increase E by 1. (E is the so-called "epact," which specifies when a full moon occurs.)

E6. [Find full moon.] Set $N \leftarrow 44 - E$. If $N < 21$ then set $N \leftarrow N + 30$. (Easter is supposedly the "first Sunday following the first full moon which occurs on or after March 21." Actually perturbations in the moon's orbit do not make this strictly true, but we are concerned here with the "calendar moon" rather than the actual moon. The Nth of March is a calendar full moon.)

E7. [Advance to Sunday.] Set $N \leftarrow N + 7 - ((D + N) \bmod 7)$.

E8. [Get month.] If $N > 31$, the date is $(N - 31)$`APRIL`; otherwise the date is N `MARCH`. ▮

Write a subroutine to calculate and print Easter date given the year, assuming the year is less than 100000. (The output should have the form "*dd* `MONTH`, *yyyyy*" where *dd* is the day, *yyyyy* is the year.) Write a complete `MIX` program which uses this subroutine to prepare a table of the dates of Easter from 1950 through 2000.

15. [*M30*] A fairly common error in the coding of the previous exercise is to fail to realize that the quantity $(11G + 20 + Z - X)$ in step E5 may be negative, and so the positive remainder mod 30 is sometimes not computed. (See *CACM* **5** (1962), 556.) For example, in the year 14250 we would find $G = 1$, $X = 95$, $Z = 40$; so if we had $E = -24$ instead of $E = +6$ we would get the ridiculous answer "37 `APRIL`". Write a complete program which finds the *earliest* year for which this error would actually cause the wrong date to be calculated for Easter.

16. [*31*] We showed in Section 1.2.7 that the sum $1 + \frac{1}{2} + \frac{1}{3} + \cdots$ becomes infinitely large. But if it is calculated with finite accuracy by a computer, the sum actually exists, in some sense, because the terms eventually get so small they contribute nothing to the sum if added one by one. For example, suppose we calculate the sum by rounding to one decimal place; then we have $1 + 0.5 + 0.3 + 0.3 + 0.2 + 0.2 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 3.9$.

More precisely, let $r_n(x)$ be the number x rounded to n decimal places; we define $r_n(x) = \lfloor 10^n x + \frac{1}{2} \rfloor / 10^n$. Then we wish to find

$$S_n = r_n(1) + r_n(\tfrac{1}{2}) + r_n(\tfrac{1}{3}) + \cdots;$$

we know that $S_1 = 3.9$, and the problem is to write a complete `MIX` program which calculates and prints S_n for $n = 2, 3, 4$, and 5.

Note: There is a much faster way to do this than the simple procedure of adding $r_n(1/m)$, one number at a time, until $r_n(1/m)$ becomes zero. (For example, we have

$r_5(1/m) = 0.00001$ for all values of m from 6667 to 200000. It is a good idea to save calculating $1/m$ all 193,334 times!) An algorithm along the following lines should rather be used:

A. Start with $m_h = 1$, $S = 1$.
B. Set $m_e = m_h + 1$ and calculate $r_n(1/m_e) = r$.
C. Find m_h, the largest m for which $r_n(1/m) = r$.
D. Add $(m_h - m_e + 1)r$ to S and return to step B.

17. [*M38*] Using the notation of the preceding exercise, prove or disprove

$$\lim_{n\to\infty} (S_{n+1} - S_n) = \ln 10.$$

18. [*25*] The ascending sequence of all reduced fractions between 0 and 1 which have denominators $\leq n$ is called the "Farey series of order n." For example, the Farey series of order 7 is

$$\tfrac{0}{1}, \tfrac{1}{7}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{2}{7}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{3}{7}, \tfrac{1}{2}, \tfrac{4}{7}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{5}{7}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \tfrac{6}{7}, \tfrac{1}{1}.$$

If we denote this series by $x_0/y_0, x_1/y_1, x_2/y_2, \ldots$, it can be shown that

$$\begin{aligned} x_0 = 0,\quad y_0 = 1;\qquad x_1 = 1,\quad y_1 = n;\\ x_{k+2} = \lfloor (y_k + n)/y_{k+1} \rfloor x_{k+1} - x_k;\\ y_{k+2} = \lfloor (y_k + n)/y_{k+1} \rfloor y_{k+1} - y_k. \end{aligned} \tag{*}$$

Write a MIX subroutine which computes the Farey series of order n, by storing the values of x_k and y_k in locations X + k, Y + k, respectively. (The total number of terms in the series is approximately $3n^2/\pi^2$ so you may assume n is rather small.)

19. [*M30*] (a) Show that the numbers x_k, y_k defined by (*) in the preceding exercise satisfy the relation $x_{k+1}y_k - x_k y_{k+1} = 1$. (b) Show that the numbers x_k, y_k given by (*) are indeed the Farey series of order n, using the fact proved in (a).

▶ **20.** [*33*] Assume the X-register and the overflow toggle of MIX have been wired up to the traffic signals at the corner of Del Mar Boulevard and Berkeley Avenue, as follows:

$$\left.\begin{aligned} \text{rX(2:2)} &= \text{Del Mar traffic light}\\ \text{rX(3:3)} &= \text{Berkeley traffic light} \end{aligned}\right\} \text{0 off, 1 green, 2 amber, 3 red;}$$

$$\left.\begin{aligned} \text{rX(4:4)} &= \text{Del Mar pedestrian light}\\ \text{rX(5:5)} &= \text{Berkeley pedestrian light} \end{aligned}\right\} \text{0 off, 1 "WALK", 2 "DON'T WALK";}$$

Cars or pedestrians wishing to travel on Berkeley across the boulevard must trip a switch which causes the overflow toggle of MIX to go on. If this condition never occurs, the light for Del Mar should remain green.

Cycle times are as follows:

Del Mar traffic light is green $\geq$ 30 sec, amber 8 sec;

Berkeley traffic light is green 20 sec, amber 5 sec.

When a traffic light is green or amber for one direction, the other direction has a red light. When the traffic light is green, the corresponding "walk" light is on; except that

28	19	10	01	48	39	30
29	27	18	09	07	47	38
37	35	26	17	08	06	46
45	36	34	25	16	14	05
04	44	42	33	24	15	13
12	03	43	41	32	23	21
20	11	02	49	40	31	22

Fig. 16. A magic square.

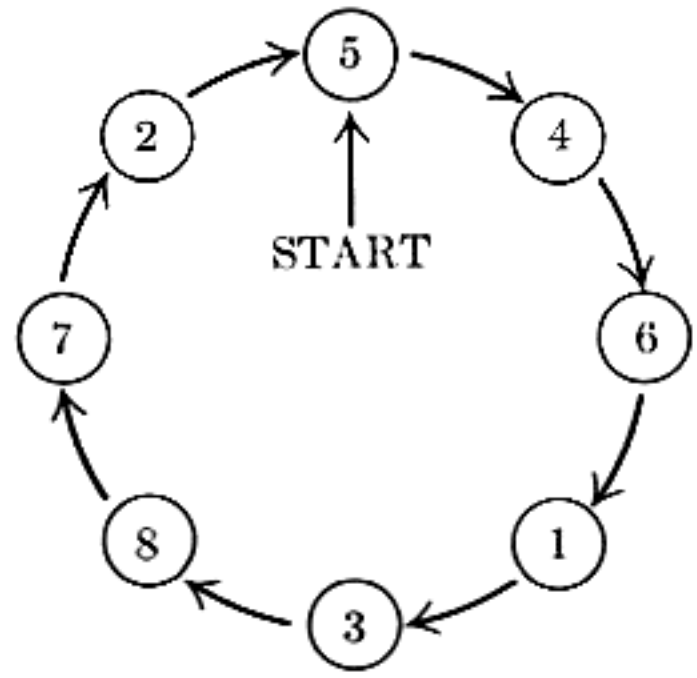


Fig. 17. Josephus' problem, $n = 8$, $m = 4$.

before the green light turns to amber, the "don't walk" light flashes for 12 sec, as follows:

DON'T WALK	$\frac{1}{2}$ sec	repeat 8 times;
off	$\frac{1}{2}$ sec	
DON'T WALK	4 sec (and remains on through amber and red cycles).	

If the overflow is tripped while the Berkeley light is green, the car or pedestrian will pass on that cycle, but if it is tripped during the amber or red portions, another cycle will be necessary after the Del Mar traffic has passed.

Assume that one MIX time unit equals 10 μsec. Write a complete MIX program which controls this traffic light by manipulating rX, according to the input given by the overflow toggle. The stated times are to be followed exactly unless it is impossible to do so. *Note:* The setting of rX changes precisely at the *completion* of a LDX or INCX instruction. *Further note:* Do not worry about the economic unfeasability of the exercise.

21. [*28*] A *magic square of order n* is an arrangement of the numbers 1 through n^2 in a square array so that the sum of each row, column, and diagonal is the same, as well as the sum of the two main diagonals. Figure 16 shows a magic square of order 7. The rule for generating it is easily seen: Start with 1 in the middle of the top row, then go up and to the left diagonally (when running off the edge imagine an entire plane tiled with squares) until reaching a filled square; then drop down one space and continue. This method works whenever n is odd.

Using memory allocated in a fashion like that of exercise 10, write a complete MIX program to generate the 23×23 magic square by the above method; then print out this magic square. [The above algorithm was brought from Siam to France by S. de La Loubère in 1687. For numerous other interesting magic square constructions, many of which are good programming exercises, see W. W. Rouse Ball, *Mathematical Recreations and Essays*, rev. by H. S. M. Coxeter (New York: Macmillan, 1962), Chapter 7.]

22. [*31*] "Josephus' problem." There are n men arranged in a circle. Beginning at a particular position, we count around the circle and brutally execute every mth man (the circle closing as men are decapitated). For example, the execution order when

$n = 8$, $m = 4$ is 54613872, as shown in Fig. 17. Write a complete MIX program which prints out the order of execution when $n = 24$, $m = 11$. Try to design a clever algorithm which works at high speed when n and m are large (it may save your life). *Reference:* W. W. Rouse Ball, as in the previous exercise, pp. 32–36.

23. [*37*] This is an exercise designed to give some experience in the many applications of computers for which the output is to be displayed graphically rather than in the usual tabular form. In this case, the object is to "draw" a crossword puzzle diagram.

You are given as input a matrix of zeros and ones. An entry of zero indicates a white square; a one indicates a black square. The output should be a diagram of the puzzle, with the appropriate squares numbered for words "across" and "down."

For example, given the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

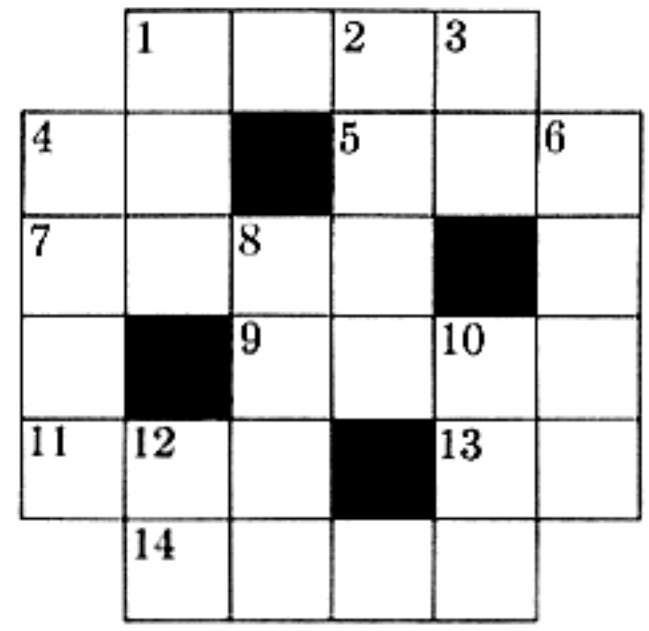


Fig. 18. Diagram corresponding to the matrix in exercise 23.

the corresponding puzzle diagram would be as shown in Fig. 18. A square is numbered if it is a white square and either (a) the square below it is white and there is no white square immediately above, or (b) there is no white square immediately to its left and the square to its right is white. If black squares are given at the edges, they should be removed from the diagram. This is illustrated in Fig. 18, where the black squares at the corners were dropped. A simple way to accomplish this is to artificially insert rows and columns of -1's at the top, bottom, and sides of the given input matrix, and then to change every "$+1$" which is adjacent to a "-1" into "-1" until no "$+1$" remains next to any "-1".

The following method should be used to print the final diagram: Each box of the puzzle should correspond to 5 columns and 3 rows of the output page. These 15 positions should be filled as follows:

```
                                  ␣␣␣␣+                                    nn␣␣+
Unnumbered white squares:         ␣␣␣␣+     Number nn white squares:       ␣␣␣␣+
                                  +++++                                    +++++

                                  +++++
                                  +++++
                Black squares:    +++++
                                  +++++
```

"-1" squares, depending on whether there are -1's to the right or below:

```
␣␣␣␣+   ␣␣␣␣+   ␣␣␣␣␣   ␣␣␣␣␣   ␣␣␣␣␣
␣␣␣␣+   ␣␣␣␣+   ␣␣␣␣␣   ␣␣␣␣␣   ␣␣␣␣␣
+++++   ␣␣␣␣+   +++++   ␣␣␣␣+   ␣␣␣␣␣
```

The diagram shown in Fig. 18 would then be printed as shown in Fig. 19.

The width of a printer line—120 characters—is enough to allow up to 23 columns in the crossword puzzle. The data supplied as input will be a 23×23 matrix of zeros and ones, each row punched in columns 1–23 of an input card. In the above example, the first card would have been punched "10000111111111111111111".

```
     +++++++++++++++++++++
     +01  +    +02  +03  +
     +    +    +    +    +
+++++++++++++++++++++++++++++
+04  +    ++++++05  +    +06  +
+    +    ++++++   +    +    +
+++++++++++++++++++++++++++++
+07  +    +08  +    ++++++    +
+    +    +    +    ++++++    +
+++++++++++++++++++++++++++++
+    +++++++09  +    +10  +    +
+    ++++++    +    +    +    +
+++++++++++++++++++++++++++++
+11  +12  +    ++++++13  +    +
+    +    +    ++++++   +    +
+++++++++++++++++++++++++++++
     +14  +    +    +    +
     +    +    +    +    +
     +++++++++++++++++++++
```

Fig. 19. Representation of Fig. 18 for printer output.

1.3.3. Applications to Permutations

In this section we shall give several more examples of MIX programs, and at the same time some important properties of permutations will be introduced. These investigations will also bring out some interesting aspects of computer programming in general.

Permutations were discussed earlier in Section 1.2.5; we treated the permutation *cdfbea* as an *arrangement* of the six objects a, b, c, d, e, f in a straight line. Another viewpoint is possible: We may think of a permutation as a *rearrangement* or renaming of the objects. With this interpretation it is customary to use a two-line notation, for example,

$$\begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix}, \tag{1}$$

to mean "a becomes c, b becomes d, c becomes f, d becomes b, e becomes e, f becomes a." Considered as a rearrangement, this means object c moves to the place formerly occupied by object a; considered as a renaming, this means object a is renamed c. The two-line notation is unaffected by changes in the order of the columns; i.e., the permutation (1) could also be written

$$\begin{pmatrix} c & d & f & b & a & e \\ f & b & a & d & c & e \end{pmatrix}$$

and 718 other ways.

A *cycle notation* is often used in connection with this interpretation. Permutation (1) could be written

$$(a\ \ c\ \ f)(b\ \ d), \tag{2}$$

again meaning "a becomes c, c becomes f, f becomes a, b becomes d, d becomes b." A cycle $(x_1\ x_2 \ldots x_n)$ means "x_1 becomes $x_2, \ldots, x_{n-1}$ becomes x_n, x_n becomes x_1." Since e is "fixed" under the permutation, it does not appear in the cycle notation; that is, singleton cycles like "(e)" are conventionally not written. If

a permutation fixes *all* elements, so there are only singleton cycles present, it is called the *identity permutation*, and it is customarily denoted by "(1)" for no really good reason.

The cycle notation is not unique; for example,

$$(b\ \ d)(a\ \ c\ \ f), \qquad (c\ \ f\ \ a)(b\ \ d), \qquad (d\ \ b)(f\ \ a\ \ c), \tag{3}$$

etc., are all equivalent to (2). However, "$(a\,f\,c)(b\,d)$" is not the same, since it says a goes to f.

It is easy to see why the cycle notation is always possible. Starting with any element x_1, the permutation takes x_1 into x_2, say, and x_2 into x_3, etc., until finally (since there are only finitely many elements) we get to some element x_{n+1} which has already appeared among $x_1, \ldots, x_n$. Now x_{n+1} must equal x_1, for if it were equal to, say, x_3, we already know x_2 goes into x_3 and by assumption $x_n \neq x_2$ goes to x_{n+1}. So we have a cycle $(x_1\, x_2 \ldots x_n)$, $n \geq 1$, as part of our permutation. If this does not account for the entire permutation, we find another element y_1 and in the same way get another cycle $(y_1\, y_2 \ldots y_m)$. None of the y's can equal any of the x's, since $x_i = y_j$ implies that $x_{i+1} = y_{j+1}$, etc., and we would ultimately find $x_k = y_1$ for some k, contradicting the choice of y_1. All cycles will eventually be found in this way.

The application of these concepts to programming comes up whenever some set of n objects is to be rearranged. To rearrange these objects without auxiliary storage, we must essentially follow the cycle structure. For example, to do the rearrangement (1), i.e., to set

$$(a, b, c, d, e, f) \leftarrow (c, d, f, b, e, a),$$

we would essentially follow the cycle structure (2) and successively set

$$t \leftarrow a, \qquad a \leftarrow c, \qquad c \leftarrow f, \qquad f \leftarrow t; \qquad t \leftarrow b, \qquad b \leftarrow d, \qquad d \leftarrow t.$$

It is frequently useful to realize that any such transformation takes place in disjoint cycles like this.

Products of permutations. We can "multiply" two permutations together, with the understanding that multiplication means the application of one permutation after the other. For example, if permutation (1) is followed by the permutation

$$\begin{pmatrix} a & b & c & d & e & f \\ b & d & c & a & f & e \end{pmatrix},$$

we have a becomes c which then becomes c; b becomes d which becomes a; etc.:

$$\begin{aligned} \begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix} \times \begin{pmatrix} a & b & c & d & e & f \\ b & d & c & a & f & e \end{pmatrix} &= \begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix} \times \begin{pmatrix} c & d & f & b & e & a \\ c & a & e & d & f & b \end{pmatrix} \\ &= \begin{pmatrix} a & b & c & d & e & f \\ c & a & e & d & f & b \end{pmatrix}. \end{aligned} \tag{4}$$

It should be clear that multiplication of permutations is not commutative, i.e., $\pi_1 \times \pi_2$ is not necessarily equal to $\pi_2 \times \pi_1$ when π_1 and π_2 are permutations. The reader may verify that the product in (4) gives a different result if the two factors are interchanged (see exercise 3).

Some people multiply permutations from right to left rather than the somewhat more natural left-to-right order shown in (4). In fact, mathematicians are divided into two camps in this regard; should the result of applying transformation T_1, then T_2, be denoted by T_1T_2 or by T_2T_1? Here we use T_1T_2.

Equation (4) would be written as follows, using the cycle notation:

$$(a\ \ c\ \ f)(b\ \ d)(a\ \ b\ \ d)(e\ \ f) = (a\ \ c\ \ e\ \ f\ \ b). \tag{5}$$

Note that the multiplication sign "$\times$" is conventionally dropped; this does not conflict with the cycle notation since it is easy to see that the permutation $(a\ \ c\ \ f)(b\ \ d)$ is really the product of the permutations $(a\ \ c\ \ f)$ and $(b\ \ d)$.

Multiplication of permutations can be done directly in terms of the cycle notation. For example, to compute the product of several permutations

$$(a\ \ c\ \ f\ \ g)(b\ \ c\ \ d)(a\ \ e\ \ d)(f\ \ a\ \ d\ \ e)(b\ \ g\ \ f\ \ a\ \ e), \tag{6}$$

we find (proceeding from left to right) that "a goes to c, then c goes to d, then d goes to a, then a goes to d, then d is unchanged"; so the net result is that a goes to d under (6), and we write down "$(a\ \ d$" as the partial answer. Now we consider the effect on d: "d goes to b goes to g," and we have the partial result "$(a\ \ d\ \ g$". Considering g, we find that "g goes to a, to e, to f, to a" and so the first cycle is closed, "$(a\ \ d\ \ g)$". Now pick a new element which hasn't appeared yet, say c; we find that c goes to e, and the reader may verify that ultimately the answer "$(a\ \ d\ \ g)(c\ \ e\ \ b)$" is obtained for (6).

Let us now try to do this process by computer. The following algorithm formalizes the method described in the preceding paragraph, in a way that is amenable to machine calculation.

Algorithm A (*Multiply permutations in cycle form*). This algorithm takes a product of cycles, such as (6), and computes the resulting permutation in the form of a product of disjoint cycles. We do not describe the removal of singleton cycles, which is a fairly simple extension of the algorithm. As this algorithm is performed, we successively "tag" the elements of the input formula, i.e., mark somehow those symbols of the input formula which have essentially been processed.

A1. [First pass.] Tag all left parentheses, and replace all right parentheses by a tagged copy of the element following their matching left parentheses. (See the example in Table 1.)

A2. [Open.] Searching from left to right, find the first untagged element of the input. (If all elements are tagged, the algorithm terminates.) Set START equal to it; output a left parenthesis; output the element; and tag it.

Table 1

ALGORITHM A APPLIED TO (6).

An "x" indicates a tagged element.

After step no.	START	CURRENT	(	*a*	*c*	*f*	*g*	*a*	(	*b*	*c*	*d*	*b*	(	*a*	*e*	*d*	*a*	(	*f*	*a*	*d*	*e*	*f*	(	*b*	*g*	*f*	*a*	*e*	*b*	Output
A1			x					x	x				x	x				x	x					x	x						x	
A2	*a*		x	x↑				x	x				x	x				x	x					x	x						x	(*a*
A3	*a*	*c*	x	x↑				x	x				x	x				x	x					x	x						x	
A4	*a*	*c*	x	x				x	x		x↑		x	x				x	x					x	x						x	
...																																
A4	*a*	*d*	x	x				x	x		x		x	x			x↑	x	x					x	x						x	
...																																
A4	*a*	*a*	x	x				x	x		x		x	x			x	x	x		x↑			x	x						x	
...																																
A5	*a*	*d*	x	x				x	x		x		x	x			x	x	x		x			x	x						x↑	*d*
...																																
A5	*a*	*g*	x	x				x	x		x	x	x	x			x	x	x		x			x	x	x					x↑	*g*
...																																
A5	*a*	*a*	x	x			x	x	x		x	x	x	x	x		x	x	x		x		x	x	x	x		x			x↑	
A6	*a*	*a*	x	x			x	x	x		x	x	x	x	x		x	x	x		x		x	x	x	x		x			x↑	)
...																																
A2	*c*	*a*	x	x	x↑		x	x	x		x	x	x	x	x		x	x	x		x		x	x	x	x		x			x	(*c*
...																																
A5	*c*	*e*	x	x	x		x	x	x		x	x	x	x	x		x	x	x	x	x		x	x	x	x		x	x		x↑	*e*
...																																
A5	*c*	*b*	x	x	x		x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x↑	*b*
...																																
A6	*c*	*c*	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x↑	)
...																																
A6	*f*	*f*	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x↑	(*f*)

A3. [Set CURRENT.] Set CURRENT equal to the next element of the formula.

A4. [Scan formula.] Proceed to the right until either reaching the end of the formula, or finding an element equal to CURRENT; in the latter case, tag it and go to step A3.

A5. [CURRENT = START?] If CURRENT ≠ START, output CURRENT and go back to step A4 starting again at the left of the formula (thereby continuing the development of a cycle in the output).

A6. [Close.] (A complete cycle in the output has been found.) Output a right parenthesis, and go back to step A2. ▮

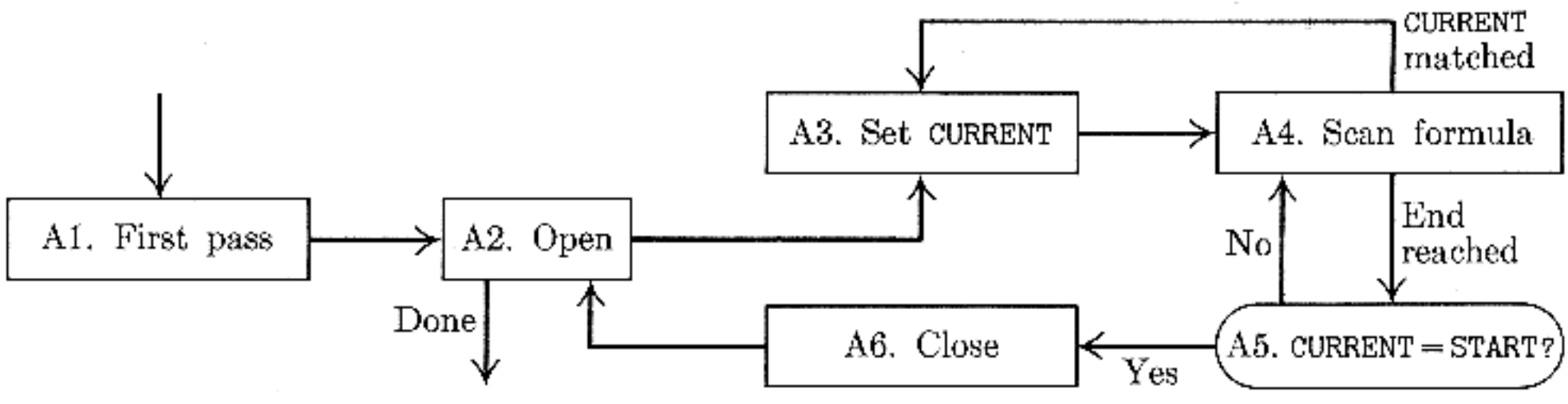


Fig. 20. Algorithm A for multiplying permutations.

For example, consider formula (6); Table 1 shows successive stages in its processing. The first line of that table shows the formula after right parentheses have been replaced by the leading element of the corresponding cycle; succeeding lines of the table show which elements have been tagged. An arrow shows the current point of interest in the formula. The output is "$(a\ d\ g)(c\ e\ d)(f)$"; note that singleton cycles will appear in the output.

A MIX program. To implement this algorithm for MIX, the "tagging" can be done by using the sign of a word. Suppose our input is punched onto cards in the following format: An 80-column card is divided into 16 five-character fields. Each field is either (a) "␣␣␣␣(", representing the left parenthesis beginning a cycle; (b) ")␣␣␣␣", representing the right parenthesis ending a cycle; (c) "␣␣␣␣␣", all blanks, which may be inserted anywhere to fill space; or (d) anything else, representing an element to be permuted. The last card of the input is recognized by having columns 76–80 equal to "␣␣␣␣=". For example, (6) might be punched on two cards as follows:

(A C F G)	(B C D)	(A E D)
(F A D E)	(B G F A E)	=

The output of our program will consist of a copy of the input followed by the answer in essentially the same format.

Program A (*Multiply permutations in cycle form*). This program implements Algorithm A, and it also includes provision for input, output, and the removing of singleton cycles.

```
01  CARDS    EQU   16                  Unit number for card reader
02  PRINTER  EQU   18                  Unit number for printer
03  ANS      ORIG  *+1000              Place for answer
04  OUTBUF   ORIG  *+24                For copies of input
05  PERM     ORIG  *+1000              The input permutation
06  BEGIN    IN    PERM(CARDS)         Read first card.
07           ENT2  0
08           LDA   EQUALS
09  1H       JBUS  *(CARDS)            Wait for cycle complete.
10           CMPA  PERM+15,2
11           JE    *+2                 Is it the last card?
12           IN    PERM+16,2(CARDS)    No, read another.
13           ENT1  OUTBUF
14           JBUS  *(PRINTER)          Print input card.
15           MOVE  PERM,2(16)
16           OUT   OUTBUF(PRINTER)
17           INC2  16
18           JNE   1B                  Repeat until input complete.
```

```
*                                       At this point, (rI2) words of
        DEC2  1              1              input are in PERM, PERM + 1, . . .
        ST2   SIZE           1
        ENT3  0              1          A1. First pass.
2H      LDAN  PERM,3         A          Get next element of input.
        CMPA  LPREN(1:5)     A          Is it "("?
        JNE   1F             A
        STA   PERM,3         B          Tag it.
        INC3  1              B          Put next nonblank element
        LDXN  PERM,3         B              in rX.
        JXZ   *-2            B
1H      CMPA  RPREN(1:5)     C
        JNE   *+2            C
        STX   PERM,3         D          Replace ")" by tagged rX.
        INC3  1              C
        CMP3  SIZE           C          Have all elements been processed?
        JL    2B             C
*
        LDA   LPREN          1          Prepare for main program.
        ENT1  ANS            1          rI1 = place to store next answer
OPEN    ENT3  0              E          A2. Open.
1H      LDXN  PERM,3         F          Look for untagged element.
        JXN   GO             F
        INC3  1              G
        CMP3  SIZE           G
        JL    1B             G
*                                       All are tagged. Now comes the output.
DONE    CMP1  =ANS=
        JNE   *+2                       Is answer the identity permutation?
        MOVE  LPREN(3)                  If so, change to "(1)".
        MOVE  =0=                       Put 23 words of blanks after answer.
        MOVE  -1,1(22)
        ENT3  0
        OUT   ANS,3(PRINTER)
        INC3  24
        LDX   ANS,3                     Print as many lines as necessary.
        JXNZ  *-3
        HLT
LPREN   ALF     (                       Constants used in program
        ALF   1
RPREN   ALF   )
EQUALS  ALF     =
*
GO      MOVE  LPREN          H          Open a cycle in the output.
        MOVE  PERM,3         H
        STX   START          H
SUCC    STX   PERM,3         J          Tag an element.
        INC3  1              J          Move one step to right.
```

```
67          LDXN  PERM,3(1:5)  J    A3. Set CURRENT (namely rX).
68          JXN   1F           J    Skip past blanks.
69          JMP   *-3          0
70    4H    CMPX  PERM,3(1:5)  K    A4. Scan formula.
71          JE    SUCC         K    Element = CURRENT?
72    1H    INC3  1            L    Move to right.
73          CMP3  SIZE         L    End of formula?
74          JL    4B           L
75          CMPX  START(1:5)   P    A5. CURRENT = START?
76          JE    CLOSE        P
77          STX   0,1          Q    No, output CURRENT.
78          INC1  1            Q
79          ENT3  0            Q    Scan formula again.
80          JMP   4B           Q    Go back to A4.
81    CLOSE MOVE  RPREN        R    A6. Close.
82          CMPA  -3,1         R    Note: rA = "(".
83          JNE   OPEN         R
84          INC1  -3           S    Suppress singleton cycles.
85          JMP   OPEN         S
86          END   BEGIN
```

This program of approximately 70 instructions is quite a bit longer than the programs of the previous section, and indeed it is longer than most of the programs we will meet in this book. Its length is not formidable, however, since it divides into several small parts which are fairly independent. Lines 06–18 read in the input cards and print a copy of each card; lines 20–35 accomplish step A1 of the algorithm, the preconditioning of the input; lines 37–44 and 62–85 do the main business of Algorithm A; and lines 46–55 output the answer. The reader will find it instructive to study as many of the MIX programs given in this book as he can—it is exceedingly important to acquire skill in reading other people's computer programs, yet such training has been sadly neglected in too many computer courses and it has led to some horribly inefficient uses of computing machinery.

Timing. The parts of Program A which are not concerned with input-output have been given "timing" indications (cf. Program 1.2.10M); thus, line 27 is supposedly executed B times. For convenience it has been assumed that no blank words appear in the input except at the extreme right end; hence line 69 is never executed and the jump in line 29 never occurs.

By simple addition the total time to execute the program is

$$(7 + 5A + 6B + 7C + 2D + E + 3F + 4G + 8H + 6J + 3K + 4L + 3P + 5Q + 6R + 2S)u \qquad (7)$$

plus the time for input and output. In order to understand the meaning of formula (7), we need to examine the fifteen unknowns $A, B, C, D, E, F, G, H, J, K, L, P, Q, R, S$ and we must relate them to pertinent characteristics about

the input. We will now illustrate the general principles of attack for problems of this kind.

First we apply "Kirchhoff's law" of electrical circuit theory: the number of times an instruction is executed must equal the number of times we transfer to that instruction. This seemingly obvious rule often relates several quantities in a nonobvious way. Analyzing the flow of Program A, we get the following equations.

From lines	*We deduce*
23, 35	$A = 1 + (C - 1)$
30, 25	$C = B + (A - B)$
39, 83, 85	$E = 1 + R$
40, 44	$F = E + (G - 1)$
62, 41	$H = F - G$
65, 68, 71	$J = H + (K - (L - J))$
70, 74, 80	$K = (L - P) + Q$
81, 76	$R = P - Q$

As usual, not all of the equations given by Kirchhoff's law will be independent; in the above case, the first and second equations are obviously equivalent. Furthermore, the last two equations are equivalent, since the third, fourth, and fifth imply that $H = R$; hence the sixth says that $K = L - R$. At any rate we have already eliminated six of our fifteen unknowns:

$$\begin{aligned} A &= C, & E &= R + 1, & F &= R + G, \\ H &= R, & K &= L - R, & Q &= P - R. \end{aligned} \tag{8}$$

Kirchhoff's law is an effective tool which is analyzed more closely in Section 2.3.4.1; it is also called the *conservative law*.

The next step is to try to identify the variables with important characteristics of the data. We find from lines 21, 22, 27, and 33 that

$$B + C = \text{number of words of input} = 16X - 1, \tag{9}$$

where X is the number of input cards. From line 25,

$$B = \text{number of "(" in input} = \text{number of cycles in input.} \tag{10}$$

Similarly, from line 31,

$$D = \text{number of ")" in input} = \text{number of cycles in input.} \tag{11}$$

Now (10) and (11) give us a fact that could not be deduced by Kirchhoff's law:

$$B = D. \tag{12}$$

From line 62,

$$H = \text{number of cycles in output (including singletons).} \tag{13}$$

Line 81 says R is equal to this same quantity; the fact that $H = R$ *was* in this case deducible from Kirchhoff's law, since it already appears in (8).

Using the fact that each nonblank word is ultimately tagged, and lines 26, 32, and 65, we find that

$$J = Y - 2B, \tag{14}$$

where Y is the number of nonblank words appearing in the input permutations. From the fact that every *distinct* element appearing in the input permutation is written into the output just once, either at line 63 or line 77, we have (see Eqs. 8)

$$P = H + Q = \text{number of distinct elements in input.} \tag{15}$$

A moment's reflection makes this clear from line 75 as well. Finally, we see from line 84 that

$$S = \text{number of singleton cycles in output.} \tag{16}$$

Clearly the quantities B, C, H, J, P, and S that we have now interpreted are essentially independent parameters which may be expected to enter into the timing of Program A.

The results we have obtained so far leave us with only the unknowns G and L to be analyzed. For these we must use a little more ingenuity. The scans of the input which start at lines 39 and 79 always terminate either at line 45 (the last time) or at line 75. During each one of these $P + 1$ loops, the instruction "`INC3 1`" is performed $B + C$ times; this takes place only at lines 42, 66, and 72, so we get the nontrivial relation

$$G + J + L = (B + C)(P + 1) \tag{17}$$

connecting our unknowns G and L. Fortunately, the timing formula is a function of $G + L$ (it involves $\cdots + 3F + 4G \cdots + 3K + 4L + \cdots = \cdots + 7G + \cdots + 7L + \cdots$) so we need not try to analyze the individual quantities G and L any further.

Summing up all the above results, we find that the total time, excluding input-output, comes to

$$(112NX + 304X + N - 2M - Y + 10U + 2V - 11)u; \tag{18}$$

in this formula, new names for the data characteristics have been used as follows:

$$\begin{aligned} X &= \text{number of cards of input,} \\ Y &= \text{number of nonblank fields in input (excluding final "="),} \\ M &= \text{number of cycles in input,} \\ N &= \text{number of distinct element names in input,} \\ U &= \text{number of cycles in output (including singletons),} \\ V &= \text{number of singleton cycles in output.} \end{aligned} \tag{19}$$

In this way we have found that analysis of a program like Program A is in many respects like solving an amusing puzzle.

We will show below that, if the output permutation is assumed to be random, the quantities U and V will be H_N and 1, respectively, on the average.

Another approach. Algorithm A multiplies permutations together much as people ordinarily do the same job. Quite often we find that problems to be solved by computer are very similar to problems that have confronted humans for many years; therefore time-honored methods of solution which have evolved for use by mortals such as we are also appropriate procedures for computer algorithms.

Just as often, however, we find that some methods which are quite unsuitable for human use are really superior for computers. The central reason is that the computer "thinks" differently; it has a different kind of memory for facts. An instance of this difference may be seen in our permutation-multiplication problem—using the algorithm below, a computer can do the multiplication in one sweep over the formula, simultaneously remembering the current state of the permutations being multiplied. While Algorithm A scans once through the formula for each element of the output, the new algorithm does all in one scan; this is a feat which could not be reliably done by man.

Let us now look into this computer-oriented method for multiplying permutations. It is convenient to go from right to left; consider the following table:

	(	a	c	f	g	)	(	b	c	d	)	(	a	e	d	)	(	f	a	d	e	)	(	b	g	f	a	e	)
a →	d	d	a	a	a	a	a	a	a	a	a	a	a	d	d	d	d	d	d	e	e	e	e	e	e	e	e	a	a
b →	c	c	c	c	c	c	c	c	g	g	g	g	g	g	g	g	g	g	g	g	g	g	g	g	b	b	b	b	b
c →	e	e	e	d	d	d	d	d	d	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
d →	g	g	g	g	g	g	g	)	)	)	d	d	)	)	)	b	b	b	b	b	d	d	d	d	d	d	d	d	d
e →	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	a	a	a	)	)	)	)	b	b	)	)	)	)	)	e
f →	f	f	f	f	e	e	e	e	e	e	e	e	e	e	e	e	e	e	a	a	a	a	a	a	a	a	f	f	f
g →	a	)	)	)	)	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	g	g	g	g

The column below each character of the cycle form represents what permutation is represented by the partial cycles *to the right;* for example, the fragmentary formula "... $d\ e)(b\ g\ f\ a\ e)$" represents the permutation

$$\begin{pmatrix} a & b & c & d & e & f & g \\ e & g & c & b & ? & a & f \end{pmatrix},$$

which appears under the rightmost d of the table.

Inspection of this table shows how it was constructed, going from right to left. The column below letter x differs from that on its right only in row x; the new value in that row and column is the one which disappeared in the preceding change. More precisely, we have the following algorithm:

Algorithm B (*Multiply permutations in cycle form*). This algorithm accomplishes essentially the same result as Algorithm A. Assume that the elements permuted are named $x_1, x_2, \ldots, x_n$. We use an auxiliary table $T[1], T[2], \ldots, T[n]$; upon termination of this algorithm, x_i goes to x_j under the input permutation if and only if $T[i] = j$.

B1. [Initialize.] Set $T[k] \leftarrow k$ for $1 \le k \le n$. Also, prepare to scan the input from right to left.

B2. [Next element.] Examine the next element of the input (right to left). If the input has been exhausted, the algorithm terminates. If the element is a ")", set $Z \leftarrow 0$ and repeat step B2; if it is a "(", go to B4; otherwise the element is x_i for some i, go on to B3.

B3. [Change $T[i]$.] Exchange $Z \leftrightarrow T[i]$. If this makes $T[i] = 0$, set $j \leftarrow i$. Return to step B2.

B4. [Change $T[j]$.] Set $T[j] \leftarrow Z$. (At this point, j is the row which shows a ")" entry in (21), corresponding to the right parenthesis which matches this left parenthesis.) Return to step B2. ∎

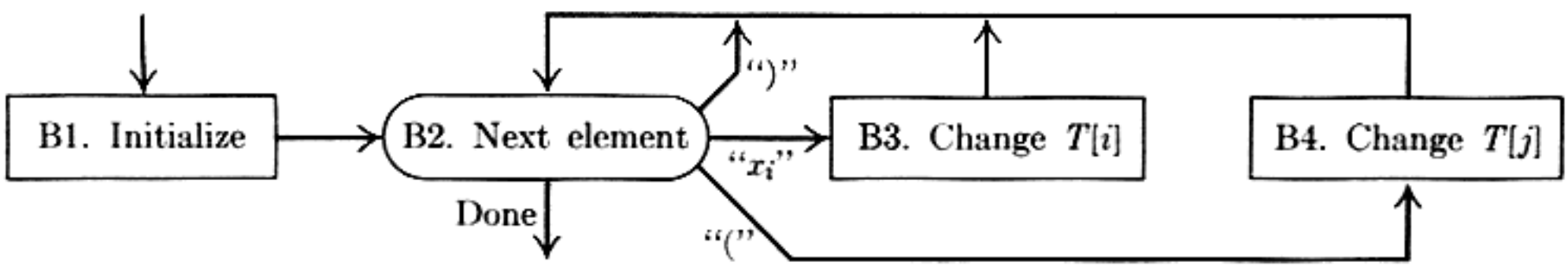


Fig. 21. Algorithm B for multiplying permutations.

Of course, after this algorithm has been performed, we still must output the contents of Table T in cycle form; this is easily done by a "tagging" method, as we shall see below.

Let us now write a MIX program based on the new algorithm. We wish to use the same ground rules as those in Program A, i.e., the form of the input and output should be essentially the same. A slight problem presents itself; namely, how can we implement Algorithm B without knowing in advance what the elements $x_1, x_2, \ldots, x_n$ are? We don't know n, and we don't know whether the element named b is to be x_1, or x_2, etc. A simple way to solve this problem is to keep a table of the element names encountered so far, and to search for the current name each time (see lines 31–36 in the program below).

Program B (*Same effect as Program A*). rX ≡ Z; rI4 ≡ i; rI1 ≡ j; rI3 ≡ 2 × (size of names table) + 1. The table of names consists of two-word entries:

Word 1	+	Name of x_i				(in character code)
Word 2	+	0	0	0	'$T[i]$'	(address of entry for x_j, if x_i goes to x_j).

```
01    NAMES   ORIG   *+1000                Table of names
02    CARDS   EQU    16        }
 . . .                         } Same as lines 01–19 of Program A
20    *                        }                  At this point, (rI2) words of
21            DEC2   1                1           input are in PERM, PERM+1 . . .
22            ENT3   1                1           Size of NAMES table
23    RIGHT   ENTX   0                A           Set Z ← 0.
24    SCAN    DEC2   1                B           B2. Next element.
25            LDA    PERM,2           B
26            JAZ    CYCLE            B           Skip over blanks.
27            CMPA   RPREN            C
28            JE     RIGHT            C           Is the next element ")"?
29            CMPA   LPREN            D
30            JE     LEFT             D           Is it "("?
31            ENT4   -2,3             E
32            J4N    1F               E           Is the names table empty?
33    2H      CMPA   NAMES,4          F           Search through names table.
34            JE     FOUND            F           Has the name appeared before?
35            DEC4   2                G
36            J4NN   2B               G
37    1H      STA    NAMES,3          H           Put new entry into table.
38            ST3    NAMES+1,3        H           Set T[i] ← i.
39            ENT4   0,3              H
40            INC3   2                H           Increase size of table.
41    FOUND   LDA    NAMES+1,4        J           B3. Change T[i].
42            STX    NAMES+1,4        J           Store Z.
43            SRC    5                J           Set Z.
44            JANZ   SCAN             J
45            ENT1   0,4              K           If Z was zero, set j ← i.
46            JMP    SCAN             K
47    LEFT    STX    NAMES+1,1        L           B4. Change T[j].
48    CYCLE   J2P    SCAN             P           Return to B2, unless finished.
49    *
50    OUTPUT  ENT1   ANS              1           All input has been scanned.
51            DEC3   2                1           Names table contains the answer.
52    1H      LDAN   NAMES,3          Q           Now we construct cycle notation.
53            JAP    SKIP             Q           Has name been tagged?
```

```
54              CMP3  NAMES+1,3     R     Is there a singleton cycle?
55              JE    SKIP          R
56              ENT2  0,3           S
57              MOVE  LPREN         S     Open a cycle.
58      2H      MOVE  NAMES,2       T
59              STA   NAMES,2       T     Tag the name.
60              LD2   NAMES+1,2     T     Find successor of element.
61              LDAN  NAMES,2       T
62              JAN   2B            T     Is it already tagged?
63              MOVE  RPREN         W     Yes, cycle closes.
64      SKIP    DEC3  2             Z     Move to next name.
65              J3P   1B            Z
66      *
67      DONE    CMP1  =ANS=  ⎫
 . . .                       ⎬ Same as lines 46–60 of Program A
81      EQUALS  ALF      =   ⎭
82              END   BEGIN   ▌
```

Lines 50–65, which construct the cycle notation from the T table (i.e., the names table), make a rather pretty little algorithm which merits some study. The quantities $A, B, \ldots, R, S, T, W. Z$ which enter into the timing of this program are, of course, different from the quantities of the same name in the analysis of Program A. The reader will find it an interesting exercise to analyze these times (see exercise 10).

Experience shows that the main portion of the execution time of Program B will be spent in searching the NAMES table—this is quantity G in the timing. Actually much better algorithms for searching and building such a NAMES table are available; these are called *symbol table algorithms*, and they are of great importance in computer applications. Section 6.5 contains a thorough discussion of efficient symbol table algorithms.

Inverses. The inverse π^{-1} of a permutation π is the rearrangement which undoes the effect of π; if i goes to j under π, then j goes to i under π^{-1}. Thus the product $\pi\pi^{-1}$ equals the identity permutation.

Every permutation has an inverse; for example, the inverse of

$$\begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix} \quad \text{is} \quad \begin{pmatrix} c & d & f & b & e & a \\ a & b & c & d & e & f \end{pmatrix} = \begin{pmatrix} a & b & c & d & e & f \\ f & d & a & b & e & c \end{pmatrix}.$$

We will now consider some simple algorithms for computing the inverse of a permutation.

For the rest of this section, let us assume we are dealing with permutations of the numbers $\{1, 2, \ldots, n\}$. If $X[1]\,X[2]\cdots X[n]$ is such a permutation, there is a simple method to compute its inverse: set $Y[X[k]] \leftarrow k$ for $1 \le k \le n$. Then $Y[1]\,Y[2]\cdots Y[n]$ is the desired inverse. This method uses $2n$ memory cells, n for X and n for Y.

Just for fun, suppose that n is very large and suppose also that we wish to compute the inverse of $X[1]\,X[2]\cdots X[n]$ without using much additional memory space; we want to compute the inverse "in place" so that after our algorithm is finished $X[1]\,X[2]\cdots X[n]$ is the inverse of the original permutation. Merely setting $X[X[k]] \leftarrow k$ for $1 \le k \le n$ will certainly fail, but by considering the cycle structure we can derive the following simple algorithm:

Algorithm I (*Inverse in place*). Replace $X[1]X[2]\cdots X[n]$, a permutation on $\{1, 2, \ldots, n\}$, by its inverse. *Reference: CACM* **8** (1965), 670.

I1. [Initialize.] Set $m \leftarrow n$.

I2. [Next element.] Set $i \leftarrow X[m]$. If $i < 0$, set $X[m] \leftarrow -i$ and go to I6 (this element has already been processed). If $i = m$, go to I6 (this element is fixed by the permutation).

I3. [Open.] Set $k \leftarrow m$.

I4. [Invert one.] Set $j \leftarrow X[i]$, $X[i] \leftarrow -k$.

I5. [End cycle.] If $j = m$, then set $X[m] \leftarrow i$; otherwise set $k \leftarrow i$, $i \leftarrow j$, and return to I4.

I6. [More?] Decrease m by 1; if $m > 0$, go to I2. Otherwise the algorithm terminates. ▮

For an example of this algorithm, see Table 2. The method is based on inversion of successive cycles of the permutation.

Table 2

COMPUTING THE INVERSE OF 6 2 1 5 4 3 BY ALGORITHM I.
(Read columns from left to right.) At point *, the cycle (163) has been inverted.

After step:	I1	I3	I4	I4	I6*	I3	I4	I6	I6	I6	I6	I6
$X[1]$	6	6	6	−3	−3	−3	−3	−3	−3	−3	−3	3
$X[2]$	2	2	2	2	2	2	2	2	2	2	2	2
$X[3]$	1	1	−6	−6	−6	−6	−6	−6	−6	6	6	6
$X[4]$	5	5	5	5	5	5	−5	−5	5	5	5	5
$X[5]$	4	4	4	4	4	4	4	4	4	4	4	4
$X[6]$	3	3	3	3	1	1	1	1	1	1	1	1
m	6	6	6	6	5	5	5	4	3	2	1	0
i		3	3	1	1	4	4	4	−5	−6	2	−3
j			1	6	6	6	5	5	5	5	5	5
k		6	6	3	3	5	5	5	5	5	5	5

Algorithm I resembles parts of Algorithm A, and it very strongly resembles the cycle-finding algorithm in Program B (lines 50–65). Thus it is typical of a number of algorithms involving rearrangements. A MIX program to implement it is quite simple; see Program I on the next page.

Program I. $m \equiv$ rI1; $i \equiv$ rI2; $(-k) \equiv$ rI3; $j \equiv$ rI4; $n \equiv$ N, a symbol to be defined when this program is assembled as part of a larger program.

```
01  INVERT  ENT1   N          1          I1. Initialize. m ← n.
02  2H      LD2    X,1        N          I2. Next element. i ← X[m].
03          J2NN   *+3        N
04          STZ    X,1(0:0)   N − C      Set X[m] positive.
05          JMP    6F         N − C
06          CMP1   X,1        C          i = m?
07          JE     6F         C
08          ENN3   0,1        C − S      I3. Open. k ← m.
09          JMP    4F         C − S
10  3H      INC4   0,1        N − 2C + S
11          ENN3   0,2        N − 2C + S k ← i.
12          ENT2   0,4        N − 2C + S i ← j.
13  4H      LD4    X,2        N − C      I4. Invert one. j ← X[i].
14          ST3    X,2        N − C          X[i] ← −k.
15          DEC4   0,1        N − C      I5. End cycle.
16          J4NZ   3B         N − C          j = m?
17          ST2    X,1        C − S          Yes, set X[m] ← i.
18  6H      DEC1   1          N          I6. More?
19          J1P    2B         N
20          HLT                          Inverse found. ▮
```

The timing for this program is easily worked out in the manner shown earlier: it is $(17N - 8C - S + 1)u$, where N is the order of the permutation, C is the total number of cycles, and S is the number of fixed elements (singleton cycles). The quantities C, S in a random permutation are analyzed below.

There is almost always more than one algorithm to do any given task, so we would expect there may be another way to invert a permutation. The following ingenious algorithm is due to J. Boothroyd:

Algorithm J (*Inverse in place*). This algorithm has the same effect as Algorithm I but uses a different method.

J1. [Negate all.] Set $X[k] \leftarrow -X[k]$, for $1 \le k \le n$. Also set $m \leftarrow n$.

J2. [Initialize j.] Set $j \leftarrow m$.

J3. [Find negative entry.] Set $i \leftarrow X[j]$. If $i > 0$, set $j \leftarrow i$ and repeat this step.

J4. [Invert.] Set $X[j] \leftarrow X[-i]$, $X[-i] \leftarrow m$.

J5. [Loop on m.] Decrease m by 1; if $m > 0$, go back to J2. Otherwise the algorithm terminates. ▮

See Table 3 for an example of this algorithm. Again the method is essentially based on the cycle structure, but this time it is less obvious that the algorithm really works! Verification is left to the reader (see exercise 13).

Table 3

COMPUTING THE INVERSE OF 6 2 1 5 4 3 BY ALGORITHM J.

After step:	J2	J3	J5	J3	J5	J3	J5	J3	J5	J3	J5	J3	J5
$X[1]$	−6	−6	−6	−6	−6	−6	−6	−6	3	3	3	3	3
$X[2]$	−2	−2	−2	−2	−2	−2	−2	−2	−2	−2	2	2	2
$X[3]$	−1	−1	6	6	6	6	6	6	6	6	6	6	6
$X[4]$	−5	−5	−5	−5	5	5	5	5	5	5	5	5	5
$X[5]$	−4	−4	−4	−4	−5	−5	4	4	4	4	4	4	4
$X[6]$	−3	−3	−1	−1	−1	−1	−1	−1	−6	−6	−6	−6	1
m	6	6	5	5	4	4	3	3	2	2	1	1	0
i		−3	−3	−4	−4	−5	−5	−1	−1	−2	−2	−6	−6
j	6	6	6	5	5	5	5	6	6	2	2	6	6

Program J (*Analogous to Program I*). $m \equiv$ rI1; $j \equiv$ rI2; $(-i) \equiv$ rI3.

```
01   INVERT  ENN1  N                 1    J1. Negate all.
02           ST1   X+N+1,1(0:0)      N        Set sign negative.
03           INC1  1                 N
04           J1N   *-2               N        More?
05           ENT1  N                 1        m ← n.
06   2H      ENN3  0,1               N    J2. Initialize j.
07           ENN2  0,3               A
08           LD3N  X,2               A    J3. Find negative entry.
09           J3N   *-2               A        i > 0?
10           LDA   X,3               N    J4  Invert.
11           STA   X,2               N        X[j] ← X[-i].
12           ST1   X,3               N        X[-i] ← m.
13           DEC1  1                 N    J5. Loop on m.
14           J1P   2B                N
15           HLT                     1    Inverse found. ▮
```

This program is a little shorter than the preceding one. To decide how fast it runs, we need to know the quantity A; this quantity is so interesting and instructive, it has been left as an exercise (see exercise 14).

In spite of the elegance of Algorithm J, we must reluctantly report that the results of an analysis of these two algorithms show that Algorithm I is definitely superior. In fact, it turns out that the running time for Algorithm I is essentially proportional to n, while that of Algorithm J is essentially proportional to $n \ln n$. As n approaches infinity (and the algorithms were intended for large n), the ratio of execution times goes to zero. It is perhaps a shame that the more subtle algorithm loses out in this case; but the analysis of algorithms is expressly intended to tell us the true facts, however greatly they might run contrary to personal taste. Maybe some day someone will find a use for Algorithm J (or some related modification); it is a bit too pretty to be forgotten altogether.

An unusual correspondence. We have already remarked that the cycle notation for a permutation is not unique; the permutation on six elements (1 6 3)(4 5) may be written (5 4)(3 1 6), etc. It will be useful to consider a *canonical form* for the cyclic notation; the canonical form *is* unique. To get the canonical form, proceed as follows:

a) Write all singleton cycles explicitly.
b) Within each cycle, put the smallest number first.
c) Order the cycles in *decreasing* order of the first number in the cycle.

For example, starting with (3 1 6)(5 4) we would get

$$\text{(a): } (3\ 1\ 6)(5\ 4)(2); \quad \text{(b): } (1\ 6\ 3)(4\ 5)(2); \quad \text{(c): } (4\ 5)(2)(1\ 6\ 3). \tag{20}$$

The important property of this canonical form is that the parentheses may be dropped and uniquely reconstructed again. Thus there is only one way to insert parentheses in "4 5 2 1 6 3" to get a canonical cycle form; one must insert a left parenthesis just before each *left-to-right minimum* (i.e., just before each element which is preceded by no smaller elements).

This insertion and removal of parentheses gives us an unusual one-to-one correspondence between the set of all permutations expressed in cycle form and the set of all permutations expressed in linear form. [*Example:* the permutation 6 2 1 5 4 3 in cycle form is (4 5)(2)(163); remove parentheses to get 4 5 2 1 6 3 which in cycle form is (2 5 6 3)(1 4); remove parentheses to get 2 5 6 3 1 4 which in cycle form is (3 6 4)(1 2 5); etc.]

This correspondence has numerous applications to the study of permutations of different types. For example, let us ask "How many cycles does a permutation on n elements have, on the average?" To answer this question we consider the set of all $n!$ permutations expressed in canonical form, and drop the parentheses; we are left with the set of all $n!$ permutations in some order. Our original question is therefore equivalent to, "How many left-to-right minima does a permutation on n elements have, on the average?" We have already answered this question in Section 1.2.10 (actually, we discussed the average number of right-to-left maxima, which is essentially the same by symmetry); this was the quantity $(A + 1)$ in the analysis of algorithm 1.2.10M, for which we found the statistics

$$\min 1, \qquad \text{ave } H_n, \qquad \max n, \qquad \text{dev } \sqrt{(H_n - H_n^{(2)})}. \tag{21}$$

Furthermore, we found that *a permutation of n objects has k cycles* (i.e., k left-to-right minima) *with probability* $\left[{n \atop k}\right]/n!$.

We can also ask about the average distance *between* left-to-right minima, which becomes equivalent to the average length of a cycle. By (21), the *total* number of cycles among all the $n!$ permutations is $n!H_n$ (since it is $n!$ times the *average* number of cycles). If we pick a cycle at random, what is its average length?

Imagine all $n!$ permutations of $\{1, 2, \ldots, n\}$ written down in cycle notation; how many three-cycles are present? To answer this question, let us consider how many times a particular three-cycle $(x\ y\ z)$ appears: clearly, the cycle $(x\ y\ z)$ appears in exactly $(n-3)!$ of the permutations, since this is the number of ways the remaining $n-3$ elements may be permuted. Now the number of different possible three-cycles $(x\ y\ z)$ is $n(n-1)(n-2)/3$, since there are n choices for x, $(n-1)$ for y, $(n-2)$ for z, and among these $n(n-1)(n-2)$ choices each different three-cycle has appeared in three forms $(x\ y\ z)$, $(y\ z\ x)$, $(z\ x\ y)$. Therefore the total number of three-cycles among all $n!$ permutations is $n(n-1)(n-2)/3$ times $(n-3)!$, namely $n!/3$. Similarly, the total number of m-cycles is $n!/m$, $1 \le m \le n$. (This provides another simple proof of the fact that the total number of cycles is $n!H_n$; hence the average number of cycles in a permutation is H_n, as we already know.) If we consider the $n!H_n$ *cycles* equally probable, the average length of a randomly chosen cycle is n/H_n; if an *element* is chosen at random in a random permutation, the average length of the cycle containing it is somewhat longer than this (see exercise 17).

To complete our analyses of Algorithms A, B, and I, we would like to know the average number of *singleton cycles* in a random permutation. This is an interesting problem. Suppose we write down the $n!$ permutations, listing first those with no singleton cycles, then those with just one, etc.; for example, if $n = 4$,

no fixed elements:	2143	2341	2413	3142	3412	3421	4123	4312	4321
one fixed element:	$\underline{1}342$	$\underline{1}423$	$3\underline{2}41$	$4\underline{2}13$	$24\underline{3}1$	$41\underline{3}2$	$231\underline{4}$	$312\underline{4}$	
two fixed elements:	$\underline{12}43$	$\underline{1}4\underline{3}2$	$\underline{1}32\underline{4}$	$4\underline{23}1$	$3\underline{2}1\underline{4}$	$21\underline{34}$			
three fixed elements:									
four fixed elements:	$\underline{1234}$								

(Singleton cycles, i.e. fixed elements, have been specially designated in this list.)

Let P_{nk} be the number of permutations of order n having exactly k fixed elements, so that for example,

$$P_{40} = 9, \qquad P_{41} = 8, \qquad P_{42} = 6, \qquad P_{43} = 0, \qquad P_{44} = 1.$$

Study of the list above shows us the principal relationship between these numbers: we can get all permutations with k fixed elements by first choosing the k that are to be fixed [this can be done in $\binom{n}{k}$ ways] and then permuting the remaining $n-k$ elements in all $P_{(n-k)0}$ ways that leave no further elements fixed. Hence

$$P_{nk} = \binom{n}{k} P_{(n-k)0}. \tag{22}$$

We also have the rule that "the whole is the sum of its parts":

$$n! = P_{nn} + P_{n(n-1)} + P_{n(n-2)} + P_{n(n-3)} + \cdots. \tag{23}$$

Combining Eqs. (22) and (23) and rewriting the result slightly, we find that

$$n! = P_{00} + \frac{1}{1!} nP_{10} + \frac{1}{2!} n(n-1)P_{20} + \frac{1}{3!} n(n-1)(n-2)P_{30} + \cdots, \quad (24)$$

an equation that must be true for all positive integers n. This equation already has confronted us before—it appears in Section 1.2.5 in connection with Stirling's attempt to generalize the factorial function—and a simple derivation of the coefficients was given in Section 1.2.6 (Eq. 32 and following). We conclude that

$$\frac{1}{m!} P_{m0} = 1 - \frac{1}{1!} + \frac{1}{2!} - \cdots + (-1)^m \frac{1}{m!}. \quad (25)$$

Now let p_{nk} be the probability that a permutation of order n has exactly k singleton cycles; since $p_{nk} = P_{nk}/n!$, we have from Eqs. (22) and (25)

$$p_{nk} = \frac{1}{k!}\left(1 - \frac{1}{1!} + \frac{1}{2!} - \cdots + (-1)^{n-k} \frac{1}{(n-k)!}\right). \quad (26)$$

The generating function $G_n(z) = p_{n0} + p_{n1}z + p_{n2}z^2 + \cdots$ is therefore

$$G_n(z) = 1 + \frac{1}{1!}(z-1) + \cdots + \frac{1}{n!}(z-1)^n = \sum_{0 \le j \le n} \frac{1}{j!}(z-1)^j. \quad (27)$$

From this formula it follows that $G_n'(z) = G_{n-1}(z)$, and from the methods of Section 1.2.10 we obtain the following statistics on the number of singleton cycles:

$$(\min 0,\ \text{ave } 1,\ \max n,\ \text{dev } 1), \qquad \text{if} \qquad n \ge 2. \quad (28)$$

A somewhat more direct way to count the number of permutations having no singleton cycles follows from the "principle of inclusion and exclusion," which is an important method for many enumeration problems. The general principle of inclusion and exclusion may be formulated as follows: We are given N elements, and M subsets, $S_1, S_2, \ldots, S_M$, of these elements; and our goal is to count how many of the elements lie in none of these subsets. Let $||S||$ denote the number of elements in a set S; then the desired number of objects in none of the sets S_j is

$$N - \sum_{1 \le j \le M} ||S_j|| + \sum_{1 \le j < k \le M} ||S_j \cap S_k|| - \sum_{1 \le i < j < k \le M} ||S_i \cap S_j \cap S_k|| + \cdots$$
$$+ (-1)^M ||S_1 \cap \cdots \cap S_M||. \quad (29)$$

(Thus we first subtract the number of elements in $S_1, \ldots, S_M$ from the total number, N, but this underestimates the desired total; so we add back the number of elements which are common to pairs of sets, $S_j \cap S_k$, for each pair S_j and S_k, then subtract the elements common to triples of sets, etc.) There are several ways to prove this formula, and the reader is invited to discover one of these for himself.

To count the number of permutations on n elements having no singleton cycles, we consider the $N = n!$ permutations and let S_j be the set of permutations in which element j forms a singleton cycle. If $1 \leq j_1 < j_2 < \cdots < j_k \leq n$, the number of elements in $S_{j_1} \cap S_{j_2} \cap \cdots \cap S_{j_k}$ is the number of permutations in which $j_1, \ldots, j_k$ are singleton cycles, and this is clearly $(n - k)!$. Thus formula (29) becomes

$$n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \binom{n}{3}(n-3)! + \cdots + (-1)^n\binom{n}{n}0!$$

and this agrees with (25).

EXERCISES

1. [*10*] Show that the transformation of the numbers $\{0, 1, 2, 3, 4, 5, 6\}$, defined by the rule that x goes to $(2x) \bmod 7$, is a permutation, and write it in cycle form.

2. [*10*] The text shows how we might set $(a, b, c, d, e, f) \leftarrow (c, d, f, b, e, a)$ by using a series of replacement operations and one auxiliary variable t. Show how to do this by using a series of *exchange* operations (i.e., $x \leftrightarrow y$) and no auxiliary variables.

3. [*10*] Compute the product

$$\begin{pmatrix} a & b & c & d & e & f \\ b & d & c & a & f & e \end{pmatrix} \times \begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix},$$

and express the answer in two-line notation (cf. Eq. 4).

4. [*10*] Express $(a\ b\ d)(e\ f)(a\ c\ f)(b\ d)$ in terms of disjoint cycles.

▶ **5.** [*M10*] Equation (3) shows several equivalent ways to express the same permutation in cycle form. How many different ways of writing that permutation are possible, if all singleton cycles are suppressed?

6. [*M23*] What changes are made to the timing of Program A if we remove the assumption that all blank words occur at the extreme right?

7. [*10*] If Program A is presented with the input (6), what are the quantities X, Y, M, N, U, and V of (19)? What is the time required by Program A, exclusive of input-output?

▶ **8.** [*23*] Would it be feasible to modify Algorithm B to go from left to right instead of from right to left through the input?

9. [*10*] Both Programs A and B accept the same input and give the answer in essentially the same form. Is the output *exactly* the same under both programs?

▶ **10.** [*M28*] Examine the timing characteristics of Program B, viz. the quantities $A, B, \ldots, Z$ shown there; express the total time in terms of X, Y, M, N, U, V [cf. (19)] and of the quantity G. Compare the total time for Program B with the total time for Program A when given the input (6), using the fact that $G = 55$ in this case (cf. exercise 7).

11. [*15*] Find a simple rule for writing π^{-1} in cycle form, if the permutation π is given in cycle form.

12. [*M27*] (Transposing a rectangular matrix.) Suppose an $m \times n$ matrix (a_{ij}), $m \neq n$, is stored in memory in a fashion like that of exercise 1.3.2–10, so that the value of a_{ij} appears in location $L + n(i - 1) + (j - 1)$, where L is the location of a_{11}. The problem is to find a way to *transpose* this matrix, obtaining an $n \times m$ matrix (b_{ij}), where $b_{ij} = a_{ji}$ and b_{ij} is stored in location $L + m(i - 1) + (j - 1)$. Thus the matrix is to be transposed "on itself." (a) Show that this transposition transformation moves the value which appears in cell $L + x$ to cell $L + (mx) \bmod N$, where $0 \leq x < N = mn - 1$. (b) Discuss methods for doing this transposition by computer.

▶ **13.** [*M24*] Prove that Algorithm J is valid.

▶ **14.** [*M34*] Find the average value of the quantity A which enters into the timing of Algorithm J.

15. [*M12*] Is there a permutation which represents exactly the same transformation both in the canonical cycle form without parentheses and in the linear form?

16. [*M15*] Start with the permutation 1324 in linear notation; convert it to canonical cycle form and then remove the parentheses; repeat this process until arriving at the original permutation. What permutations occur during this process?

17. [*M24*] (a) The text demonstrates that there are $n!H_n$ cycles in all among the permutations on n elements. If these cycles (including singleton cycles) are individually written on $n!H_n$ slips of paper, and if one of these slips of paper is chosen at random, what is the average length of the cycle that is thereby picked? (b) If we write the $n!$ permutations on $n!$ slips of paper, and if we choose a number k at random and also choose one of these slips of paper, what is the probability that the cycle containing the element k is an m-cycle? What is the average length of the cycle containing k?

▶ **18.** [*M27*] What is p_{nkm}, the probability that a permutation of n objects has exactly k m-cycles? What is the corresponding generating function $G_{nm}(z)$? What is the average number of m-cycles and what is the standard deviation? (The text considers only the case $m = 1$.)

19. [*HM21*] Show that, in the notation of Eq. (25),

$$P_{n0} = \lfloor n!/e + \tfrac{1}{2} \rfloor, \qquad n \geq 1.$$

20. [*M20*] Given that all singleton cycles are written out explicitly, how many different ways are there to write the cycle notation of a permutation which has α_1 one-cycles, α_2 two-cycles, . . . ? (Cf. exercise 5.)

21. [*M22*] What is the probability $P(n; \alpha_1, \alpha_2, \ldots)$ that a permutation of order n has exactly α_1 one-cycles, α_2 two-cycles, etc.?

▶ **22.** [*HM34*] (The following approach, due to L. Shepp and S. P. Lloyd, gives a convenient and powerful method for solving problems related to the cycle structure of random permutations.) Instead of regarding the number, n, of objects as fixed, and the permutation variable, let us instead suppose that we independently choose the quantities $\alpha_1, \alpha_2, \alpha_3, \ldots$ appearing in exercises 20 and 21 according to some probability distribution. Let w be any real number between 0 and 1. (a) Suppose that we choose the random variables $\alpha_1, \alpha_2, \alpha_3, \ldots$ according to the rule that "the probability

$\alpha_m = k$ is $f(w, m, k)$," for some function $f(w, m, k)$. Determine the value of $f(w, m, k)$ so that the following two conditions hold:

i) $\sum_{k\geq 0} f(w, m, k) = 1$, for $0 < w < 1$ and $m \geq 1$.

ii) The probability that $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots = n$ *and* that $\alpha_1 = k_1$, $\alpha_2 = k_2$, $\alpha_3 = k_3, \ldots,$ is $(1 - w)w^n P(n; k_1, k_2, k_3, \ldots)$ as in exercise 21.

b) A permutation whose cycle structure is $\alpha_1, \alpha_2, \alpha_3, \ldots$ clearly permutes exactly $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots$ objects. Show that if the α's are randomly chosen according to the probability distribution in part (a), the probability that $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots = n$ is $(1 - w)w^n$; the probability that $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots$ is *infinite* is zero.

c) Let $\phi(\alpha_1, \alpha_2, \ldots)$ be any function of the infinitely many numbers $\alpha_1, \alpha_2, \ldots$; show that if the α's are chosen according to the probability distribution in (a), the average value of ϕ is $(1 - w) \sum_{n\geq 0} w^n \phi_n$; here ϕ_n denotes the average value of ϕ taken over all permutations of n objects, where $\alpha_1, \alpha_2, \ldots$ represent the number of cycles of the permutation. [For example, if $\phi(\alpha_1, \alpha_2, \ldots) = \alpha_1$, the text showed that $\phi_n = 1$, the average number of singleton cycles, regardless of n.]

d) Use this method to find the average number of cycles of *even* length in a random permutation of n objects.

e) Use this method to solve exercise 17. (Cf. exercise 1.2.10–15.)

23. [*HM45*] (Golomb, Shepp, Lloyd.) If l_n denotes the average length of the *longest* cycle in a permutation of n objects, show that $l_n \approx \lambda n + \frac{1}{2}\lambda$, where $\lambda \approx 0.62433$ is a constant. Show in fact that $\lim_{n\to\infty} (l_n - \lambda n - \frac{1}{2}\lambda) = 0$.

24. [*M41*] Find the generating function for the quantity A which enters into the timing of Algorithm J. (Cf. exercise 14.)

25. [*M22*] Prove Eq. (29).

▶ **26.** [*M24*] Extend the principle of inclusion and exclusion to obtain a formula for the number of elements which are in exactly r of the subsets $S_1, S_2, \ldots, S_M$. (The text considers only the case $r = 0$.)

27. [*M20*] Use the principle of inclusion and exclusion to count the number of integers n in the range $0 \leq n < am_1m_2 \cdots m_t$, where $a, m_1, m_2, \ldots, m_t$ are positive integers, with $\gcd(m_j, m_k) = 1$ when $j \neq k$.

1.4. SOME FUNDAMENTAL PROGRAMMING TECHNIQUES

1.4.1. Subroutines

When a certain task is to be performed at several different places in a program, it is usually undesirable to repeat the coding in each place. To avoid this situation, the coding (called a "subroutine") can be put into one place only, and a few extra instructions can be added to restart the outer program properly after the subroutine is finished. Transfer of control between subroutines and main programs is called "subroutine linkage."

Each machine has its own peculiar manner for achieving efficient subroutine linkage, usually involving special instructions. In MIX, the J-register is used for this purpose; our discussion will be based on MIX machine language, but similar remarks will apply to subroutine linkage on other computers.

Subroutines are used to save space in a program; they do not save any time, other than the time implicitly saved by having less space (e.g., less time to load the program, or fewer passes necessary in the program, or better use of high-speed memory on machines with several grades of memory). The extra time taken to enter and leave a subroutine is usually negligible.

Subroutines have several other advantages. They make it easier to visualize the structure of a large and complex program; they form a logical segmentation of the entire problem, and this usually makes debugging of the program easier. Many subroutines have additional value because they can be used by people other than the programmer of the subroutine.

Most computer installations have built up a large "library" of useful subroutines, and such a library greatly facilitates the programming of standard computer applications which arise. A programmer should not think of this as the *only* purpose of subroutines, however; subroutines should not always be regarded as "general purpose" programs to be used by the community. Even the use of very special purpose subroutines, which are intended to appear in only one program, is an important technique.

The simplest subroutines are those which have only one entrance and one exit, such as the MAXIMUM subroutine we have already considered (see Section 1.3.2, Program M). For reference, we will recopy that program here, changing it so that a fixed number of cells, 100, is searched for the maximum:

```
MAX100  STJ   EXIT    Subroutine linkage
        ENT3  100     M1. Initialize.
        JMP   2F
1H      CMPA  X,3     M3. Compare.
        JGE   *+3                                        (1)
2H      ENT2  0,3     M4. Change m.
        LDA   X,3     (New maximum found)
        DEC3  1       M5. Decrease k.
        J3P   1B      M2. All tested?
EXIT    JMP   *       Return to main program. ▮
```

In a larger program containing this coding as a subroutine, the single instruction "JMP MAX100" would cause register A to be set to the current maximum value of locations X + 1 through X + 100, and the position of the maximum would appear in rI2. Subroutine linkage in this case is achieved by the instructions "MAX100 STJ EXIT" and, later, "EXIT JMP *". Because of the way the J-register operates, the exit instruction will then jump to the location following the place where the original reference to MAX100 was made.

It is not hard to obtain *quantitative* statements about the amount of code saved and the amount of time lost when subroutines are used. Suppose that a piece of coding requires k locations and that it appears in m places in the program. Rewriting this as a subroutine, we need an extra instruction STJ and an exit line for the subroutine, plus a single JMP instruction in each of the m places where the subroutine is called. This gives a total of $m + k + 2$ locations, rather than mk, so the amount saved is

$$(m - 1)(k - 1) - 3. \tag{2}$$

If k is 1 or m is 1 we cannot possibly save any space by using subroutines; this, of course, is obvious. If k is 2, m must be greater than 4 in order to gain, etc.

The amount of time lost is the time taken for the extra JMP, STJ, and JMP instructions, which are not present if the subroutine is not used; therefore if the subroutine is used t times during a run of the program, $4t$ extra cycles of time are required.

These estimates must be taken with a grain of salt, because they were given for an idealized situation. Many subroutines cannot be called simply with a single JMP instruction. Furthermore, if the coding is repeated in many parts of a program, without using a subroutine approach, the coding for each part can take advantage of special characteristics of the particular part of the program in which it lies. With a subroutine, on the other hand, the coding must be written for the most general case, not a specific case, and this will often add several additional instructions.

When a subroutine is written to handle a general case, it is often written in terms of *parameters*, values which govern the subroutine's action, but which are subject to change from one call of the subroutine to another.

The coding in the outside program which transfers control to the subroutine and gets it properly started is known as the "calling sequence." Particular values of parameters, supplied when the subroutine is called, are known as *arguments*. With our MAX100 subroutine, the calling sequence is simply "JMP MAX100", but when arguments must be supplied, a longer calling sequence is generally necessary. For example, Program 1.3.2M is a generalization of MAX100 which finds the maximum of the first n elements of the table. The parameter n appears in index register 1, and we may regard the calling sequence as

```
LD1   =n=
JMP   MAXIMUM.
```

If the calling sequence takes c memory locations, formula (2) for the amount of space saved changes to

$$(m - 1)(k - c) - \text{const} \tag{3}$$

and the time lost for subroutine linkage is slightly increased.

A further correction to the above formulas can be necessary because certain registers might need to be saved and restored. For example, in the `MAX100` subroutine, the programmer must remember that by writing "`JMP MAX100`" he is not only getting the maximum value in register A and its position in register I2; he is also setting register I3 to zero. A subroutine may destroy register contents, and this must be kept in mind. In order to prevent `MAX100` from changing the setting of rI3, it would be necessary to insert "`ST3 TEMP`" just after `MAX100` and also "`LD3 TEMP`" just before `EXIT`; it is an extra two lines of code, plus perhaps another memory cell for `TEMP`, and an extra four machine cycles on every call of the subroutine. (Of course, the subroutine now destroys the contents of memory cell `TEMP`, and the programmer must keep this in mind; it seems we have solved one problem only to have created another. There are two common solutions to this possible dilemma: (a) essentially make `TEMP` part of the subroutine, by changing the instructions to "`ST3 3F`" and "`LD3 3F`" and adding the line "`3H CON 0`" just after `EXIT`; or (b) use a systematic method for the names of temporary storage locations in subroutines, as explained later in this section or as explained in Chapter 8.)

A subroutine may be regarded as an *extension* of the computer's machine language. With the `MAX100` subroutine in memory, we now have a single instruction (namely, "`JMP MAX100`") which is a maximum-finder. It is important to define the effect of each subroutine just as carefully as the machine language operators themselves have been defined, and so the programmer should be sure to write down the characteristics of each subroutine, even though he himself will be the only one to make use of it. In the case of `MAXIMUM` as given in Section 1.3.2, the characteristics are as follows:

Calling sequence:	`JMP MAXIMUM.`	
Entry conditions:	$\text{rI1} = n$; assume $n \geq 1$.	
Exit conditions:	$\text{rA} = \max_{1 \leq k \leq n} \texttt{CONTENTS}\ (\texttt{X} + k)$	(4)
	$= \texttt{CONTENTS}\ (\texttt{X} + (\text{rI2}))$;	
	rI3 = 0; rJ and CI are also affected.	

(We will customarily omit mention of the fact that register J and the comparison indicator are affected by a subroutine; it has been mentioned here only for completeness.) Note that rX and rI1 are unaffected by the action of the subroutine, for otherwise these registers would have been mentioned in the exit conditions.

Now let us consider *multiple entrances* to subroutines. Suppose that we have a program which requires the general subroutine MAXIMUM, but which most frequently wants to use the special case MAX100, with $n = 100$. The two can be combined as follows:

```
MAX100  ENT3  100     First entrance
MAXN    STJ   EXIT    Second entrance
        JMP   2F      Continue as in (1).          (5)
...
EXIT    JMP   *
```

Subroutine (5) is essentially the same as (1), with the first two instructions interchanged; we have used the fact that "ENT3" does not change the setting of the J-register. If we were to add a *third* entrance, MAX50, to this subroutine, we could insert the code

```
MAX50   ENT3  50
        JSJ   MAXN                                   (6)
```

at the beginning. (Recall that "JSJ" means jump without changing register J.)

When the number of parameters is small, it is often desirable to transmit them to a subroutine either by having them in convenient registers (as we have used rI3 to hold the parameter n in MAXN and as we used rI1 to hold the parameter n in MAXIMUM), or by storing them in fixed memory cells. Another way to supply arguments which is often convenient is to simply list them *after* the JMP instruction; the subroutine may refer to its parameters because it knows the J-register setting.

For example, if we wanted to make the calling sequence for MAXN be

```
JMP   MAXN
CON   n                                              (7)
```

then the subroutine could be written

```
MAXN    STJ   *+1
        ENT1  *       rI1 ← rJ.
        LD3   0,1     rI3 ← n.
        JMP   2F      Continue as in (1).            (8)
...
        J3P   1B
        JMP   1,1     Return. ▮
```

On machines such as the 709 for which linkage is ordinarily done by putting the exit location in an index register, the above procedure is particularly convenient. It is also useful when a fairly large number of arguments is to be passed to a subroutine, as well as in conjunction with programs written by compilers

(see Section 12.2.7). The technique of multiple entrances which we used above often fails in this case, however; we could "fake it" by writing

```
MAX100  STJ  1F
        JMP  MAXN
        CON  100
1H      JMP  *
```

but this is not as attractive as (5).

A technique similar to that of listing arguments after the jump is normally used for subroutines with *multiple exits*. Multiple exit means that we want the subroutine to return to one of several different locations, depending on conditions detected by the subroutine. In the strictest sense, the location to which a subroutine exits is a parameter; so if there are several places to which it should exit, depending on the circumstances, these should be supplied as arguments. Our final example of the "maximum" subroutine will have two entrances and two exits. The calling sequence is:

For general n	For $n = 100$
`ENT3` n	
`JMP MAXN`	`JMP MAX100`
Exit here if max ≤ 0 or max $\geq$ rX.	Exit here if max ≤ 0 or max $\geq$ rX.
Exit here if $0 <$ max $<$ rX.	Exit here if $0 <$ max $<$ rX.

(In other words, exit is made to the location *two* past the jump when the maximum value is positive and less than the contents of register X.) The subroutine for these conditions is easily written:

```
MAX100  ENT3  100     Entrance for n = 100
MAXN    STJ   EXIT    Entrance for general n
        JMP   2F      Continue as in (1).
...
        J3P   1B
        JANP  EXIT    Is max positive?                    (9)
        STX   TEMP
        CMPA  TEMP
        JGE   EXIT    Is it less than rX?
        INC3  1       Set rI3 ← 1.
EXIT    JMP   *,3     Return to proper place. ▮
```

In summary, subroutines are often desirable for saving space in a program and reducing its complexity. As always, we don't get something for nothing, and some expense in running time occurs. If the extra time for calling a subroutine is small compared to the total execution time for that subroutine, and if it is fairly long or is referred to quite often, then writing it as a subroutine is far superior to rewriting the code over and over in the program. For typical

uses of subroutines in a larger program, see the examples in Section 1.4.2.1 and throughout the assembler in Chapter 9.

Subroutines may call on other subroutines; in complicated programs it is not unusual to have subroutine calls nested more than 10 deep. The restriction that must be followed when using linkage as described here, however, is that no subroutine may call on any other subroutine which is (directly or indirectly) calling on it. For example,

```
[Main program]  | [Subroutine A]      | [Subroutine B]      | [Subroutine C]
                |                     |                     |
                | A      STJ EXITA    | B      STJ EXITB    | C      STJ EXITC
   ⋮            |         ⋮           |         ⋮           |         ⋮
 JMP A          |        JMP B        |        JMP C        |        JMP A
   ⋮            |         ⋮           |         ⋮           |         ⋮
                | EXITA  JMP *        | EXITB  JMP *        | EXITC  JMP *
                |                     |                     |
```

(10)

If the main program calls on A, which calls B, which calls C, and then C calls on A, the address in EXITA referring to the main program is destroyed, and there is no way to return to the main program. A similar remark applies to all temporary storage cells and registers used by each subroutine. It is possible to make subroutine linkages which will handle the above "recursive" situation properly, and these will be discussed in Chapter 8.

We conclude this section by discussing briefly how we might go about writing a complex and lengthy program. How can we decide what kind of subroutines we will need, and what calling sequences should be used? One successful way to determine this is to use an iterative procedure:

Step 0 (Initial idea). First we vaguely decide upon the general plan of attack in the program.

Step 1 (A rough sketch of the program). We start now by writing the middle portions of the program, assuming that initialization in some form has taken place. (Lacking clairvoyance, we cannot figure out what initialization will be necessary until the rest of the program is written.) With this assumption, we write down the steps the program is to carry out, either in English flow outline form as used in the algorithms of this book, or in flow-chart form, or immediately in some computer language, according to personal preference. Whenever something occurs which seems likely to occur elsewhere or which has already occurred elsewhere, we define a subroutine to do that job. We do not write the subroutine at this point; we continue writing the main program, assuming the subroutine has performed its task. Finally, when the main program has been sketched, we tackle the subroutines in turn, trying to take the most complex subroutines first and then their sub-subroutines, etc. In this manner we will come up with a list of subroutines. The actual function of each subroutine has probably already changed several times, so that the first parts of our sketch will

by now be incorrect; but that is all right, for it is merely a sketch. For each subroutine we now have a reasonably good idea as to how it will be called and how general-purpose it should be. It usually pays to extend the generality of each subroutine a little.

Step 2 (First working program). This step goes in the opposite direction from step 1. We now write in computer language, say MIXAL; we start this time with the lowest level subroutines, and do the main program last. As far as possible, we try never to write any instructions which call a subroutine before the subroutine itself has been coded. (In step 1, we tried the opposite, never considering a subroutine until all of its calls had been written.)

As more and more subroutines are written during this process, our confidence gradually grows, since we are continually extending the power of the machine we are programming. After an individual subroutine is coded, we should immediately prepare a complete description of what it does, and what its calling sequences are, as in (4). It is also important not to overlay temporary storage cells; it may very well be disastrous if every subroutine refers to location TEMP, although when preparing the sketch in step 1, it is convenient not to worry about this problem. An obvious way to overcome overlay worries is to have each subroutine use only its own temp storage, but if this is too wasteful of space, another scheme which does fairly well is to name the cells TEMP1, TEMP2, etc.; the numbering within a subroutine starts with TEMPj, where j is one higher than the greatest number used by any of the sub-subroutines of this subroutine.

Step 3 (Reexamination). The result of step 2 should be very nearly a working program, but it may be possible to improve on it. A good way is to reverse direction again, studying for each subroutine *all* of the calls made on it. It may well be that the subroutine should be enlarged to do some of the more common things which are always done by the outside routine just before or after it uses the subroutine. Perhaps several subroutines should be merged into one; or perhaps a subroutine is called only once (or, if we are fortunate, perhaps one is never called!) and should not be a subroutine at all.

At this point, it is often a good idea to scrap everything and start over again at step 1! This is not intended to be a facetious remark; the time spent in getting this far has not been wasted, for we have learned a great deal about our problem. We will probably know of several improvements that can be made to the organization of the program; there is no reason to be afraid to go back to step 1—it will be much easier to go through steps 1 and 2 again after a program has been done already. Moreover, we will quite probably save as much debugging time later on as it will take to rewrite the program. Some of the best computer programs ever written owe much of their success to the fact that at about this stage all the work was unintentionally lost and the authors had to begin again.

On the other hand, there is probably never a point when a complex computer program cannot be improved somehow, so steps 1 and 2 should not be repeated indefinitely; see the further discussion in Section 9.3.10. When significant improvements can clearly be made, it is well worth the additional time required to start over.

Step 4 (Initialization). Go through the program, allocate storage for it, and figure out exactly what needs to be initialized; then write the initialization part of the program last. (This step is something like writing a textbook: Chapter 1 is usually written last.)

Step 5 (Debug the program). Here we may look at our program in still another direction from the three that were used in steps 1, 2, and 3—we study the program in the order in which the computer will *perform* it.

Debugging is an art that needs much further study, and the way to approach it is highly dependent on the facilities that are available at each computer installation. A good start towards effective debugging is often the preparation of an appropriate "test deck," as explained in Section 9.3.10. The most effective debugging techniques seem to be those which are designed and built into the program itself—many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other half; the first half, which usually consists of fairly straightforward routines, will eventually be thrown away, but the net result is a surprising gain in productivity.

Another good debugging practice is to keep a record of every mistake that is made. Even though this will probably be quite embarrassing, such information is invaluable to anyone doing research on the debugging problem, and it will also help you learn how to reduce the number of future errors.

EXERCISES

1. [*10*] State the characteristics of subroutine (5), just as (4) gives the characteristics of Subroutine 1.3.2M.

2. [*10*] Suggest code to substitute for (6) without using the JSJ instruction.

3. [*M15*] Complete the information in (4) by stating exactly what happens to register J and the comparison indicators as a result of the subroutine; state also what happens if register I1 is not positive.

▶ 4. [*21*] Write a subroutine that generalizes MAXN by finding the maximum value of X[1], X[$1+r$], X[$1+2r$], . . . , X[n], where r and n are parameters. Give a special entrance for the case $r = 1$.

5. [*21*] Suppose that MIX did not have a J-register. Invent a means for subroutine linkage which does not use register J, and give an example of your invention by writing a MAX100 subroutine effectively equivalent to (1). State the characteristics of this subroutine in a fashion similar to (4).

▶ 6. [*26*] Suppose MIX did not have a MOVE operator; write a subroutine entitled MOVE such that the calling sequence

```
JMP  MOVE
NOP  A,I(F)
```

has an effect just the same as "MOVE A,I(F)" if the latter were admissible. The only differences should be the effect on register J and the fact that the time to execute the subroutine will be somewhat longer.

1.4.2. Coroutines

Subroutines are special cases of more general program components, called "coroutines." In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines, which *call on each other*.

To understand the coroutine concept, let us consider another way of thinking about subroutines. The viewpoint adopted in the previous section was that a subroutine merely was an extension of the computer hardware, introduced to save lines of coding. This may be true, but another point of view is possible: We may consider the main program and the subroutine as a *team* of programs, with each member of the team having a certain job to do. The main program, in the course of doing its job, will activate the subprogram; the subprogram performs its own function and then activates the main program. We might stretch our imagination to believe that, from the subroutine's point of view, when it exits *it* is calling the *main* routine; the main routine continues to perform its duty, then "exits" to the subroutine. The subroutine acts, then calls the main routine again.

This somewhat far-fetched philosophy actually takes place with coroutines, when it is impossible to distinguish which is a subroutine of the other. Suppose we have coroutines `A` and `B`; when programming `A`, we may think of `B` as our subroutine, but when programming `B`, we may think of `A` as our subroutine. That is, in coroutine `A`, the instruction "`JMP B`" is used to activate coroutine `B`. In coroutine `B` the instruction "`JMP A`" is used to activate coroutine `A` again. It represents teamwork as in a relay race. Whenever a coroutine is activated, it resumes execution of its program at the point where the action was last suspended.

With `MIX`, such linkage between coroutines `A` and `B` is done by including the following four instructions in the program:

```
A   STJ  BX          B   STJ  AX
AX  JMP  A1          BX  JMP  B1
```
(1)

This requires four machine cycles for transfer of control each way. Initially `AX` and `BX` are set to jump to the starting places of each coroutine, `A1` and `B1`. Suppose we start up coroutine `A` first, at location `A1`. When it executes "`JMP B`" from location `A2`, say, the instruction in location `B` stores rJ in `AX`, which then says "`JMP A2+1`". The instruction in `BX` gets us to location `B1`, and after coroutine `B` begins its execution, it will eventually get to an instruction "`JMP A`" in location `B2`, say. We store rJ in `BX` and jump to location `A2+1`, continuing the execution of coroutine `A` until it again jumps to `B`, which stores J in `AX` and jumps to `B2+1`, etc.

The essential difference between routine-subroutine and coroutine-coroutine linkage, as can be seen by studying the example above, is that a subroutine is always initiated *at its beginning*, i.e., at a fixed place, while the main routine or a coroutine is always initiated *at the place following* where it last terminated.

Coroutines connected with algorithms for input and output arise most naturally in practice. For example, suppose it is the duty of coroutine A to read cards and to perform some transformation on the input, reducing it to a sequence of items. Another coroutine, which we will call B, does further processing of these items, and prints the answers; B will periodically call for the successive input items found by A. Thus, coroutine B jumps to A whenever it wants the next input item, and coroutine A jumps to B whenever an input item has been found. (The reader may say, "Well, B is the main program and A is merely a *subroutine* for doing the input." This, however, becomes less true when the process A is very complicated; indeed, we can imagine A as the main routine and B as a subroutine for doing the output, and the above description remains valid. The usefulness of the coroutine idea emerges midway between these two extremes, when both A and B are complicated and each one calls the other in numerous places. It is rather difficult to find short, simple examples of coroutines which illustrate the importance of the idea; the most useful coroutine applications are generally quite lengthy.

In order to study coroutines in action, let us consider a "contrived" example. Suppose we want to write a program that translates one code into another. The input code to be translated is a sequence of alphameric characters terminated by a period, e.g.,

```
A2B5E3426FG0ZYW3210PQ89R.                    (2)
```

which has been punched onto cards; blank columns appearing on these cards are to be ignored. This input is to be understood as follows, from left to right: If the next character is a digit (i.e., 0, 1, . . . , 9), say n, it indicates $(n + 1)$ repetitions of the following character, whether the following character is a digit or not. A nondigit simply denotes itself. The output of our program is to consist of the sequence indicated in this manner and separated into groups of three characters each (where the last group may have less than three characters). For example, (2) should be translated by our program into

```
ABB  BEE  EEE  E44  446  66F  GZY  W22  220  OPQ  999  999  999  R.   (3)
```

Note that 3426F does not mean 3427 repetitions of the letter F; it means 4 fours and 3 sixes followed by F. Our program is to punch the output onto cards, with sixteen groups of three on each card.

To accomplish this translation, we will write two coroutines and a subroutine. The subroutine, called NEXTCHAR, is designed to successively find nonblank characters of input, and to put the next character into register A:

```
01    * SUBROUTINE FOR CHARACTER INPUT
02    READER    EQU   16                 Unit number of card reader
03    INPUT     ORIG  *+16               Place for input cards
04    NEXTCHAR  STJ   9F                 Entrance to subroutine
05              JXNZ  3F                 Initially rX = 0
06    1H        J6N   2F                 Initially rI6 = 0
07              IN    INPUT(READER)      Read next card.
```

```
08          JBUS  *(READER)        Wait for completion.
09          ENN6  16               Let rI6 point to first word.
10    2H    LDX   INPUT+16,6       Get next word of input.
11          INC6  1                Advance pointer.
12    3H    ENTA  0
13          SLAX  1                Next character → rA.
14    9H    JANZ  *                Skip blanks.
15          JMP   NEXTCHAR+1       ▮
```

This subroutine has the following characteristics:

Calling sequence: `JMP NEXTCHAR`.

Entry conditions: rI6 points to next word, or rI6 = 0 indicating that a new card must be read; rX = characters yet to be used.

Exit conditions: rA = next nonblank character of input; rX, rI6 set for next entry to `NEXTCHAR`.

Our first coroutine, called `IN`, finds the characters of the input code with the proper replication:

```
16    * FIRST COROUTINE
17    2H    INCA  30               Nondigit found
18          JMP   OUT              Send it to OUT coroutine.
19    IN1   JMP   NEXTCHAR         Get character.
20          DECA  30
21          JAN   2B               Is it a letter?
22          CMPA  =10=
23          JGE   2B               Is it a special character?
24          STA   *+1(0:2)         Digit n found
25          ENT5  *                rI5 ← n.
26          JMP   NEXTCHAR         Get next character.
27          JMP   OUT              Send it to OUT coroutine.
28          DEC5  1                Decrease n by 1.
29          J5NN  *-2              Repeat if necessary.
30          JMP   IN1              Begin new cycle. ▮
```

(Recall that in `MIX`'s character code, the digits 0–9 have codes 30–39.) This coroutine has the following characteristics:

Calling sequence: `JMP IN`.

Exit conditions (when jumping to `OUT`): rA = next character of input with proper replication; rI4 unchanged from its value at entry.

Entry conditions (upon return): rA, rX, rI5, rI6 should be unchanged from their values at the last exit.

The other coroutine, called OUT, puts the code into three-digit groups and punches the cards:

```
31      * SECOND COROUTINE
32                ALF                          Constant used for blanking
33      OUTPUT    ORIG  *+16                   Buffer area for answers
34      PUNCH     EQU   17                     Unit number for card punch
35      OUT1      ENT4  -16                    Start new output card.
36                ENT1  OUTPUT
37                MOVE  -1,1(16)               Set output area to blanks.
38      1H        JMP   IN                     Get next translated character.
39                STA   OUTPUT+16,4(1:1)       Store in output.
40                CMPA  PERIOD                 Is it "."?
41                JE    9F
42                JMP   IN                     If not, get another character.
43                STA   OUTPUT+16,4(2:2)       Store it.
44                CMPA  PERIOD                 Is it "."?
45                JE    9F
46                JMP   IN                     If not, get another character.
47                STA   OUTPUT+16,4(3:3)       Store it.
48                CMPA  PERIOD                 Is it "."?
49                JE    9F
50                INC4  1                      Move to next word in output.
51                J4N   1B                     End of card?
52      9H        OUT   OUTPUT(PUNCH)          If so, punch.
53                JBUS  *(PUNCH)               Wait for completion.
54                JNE   OUT1                   Return for more, unless
55                HLT                                "." was sensed.
56      PERIOD    ALF                                 ▮
```

This coroutine has the following characteristics:

Calling sequence:	JMP OUT.
Exit conditions (when jumping to IN):	rA, rX, rI5, rI6 unchanged from their value at entry.
Entry conditions (upon return):	rA = next character of input with proper replication; rI4 unchanged from its value at the last exit.

To complete the program, we need to write the coroutine linkage [cf. (1)] and to provide the proper initialization. Initialization of coroutines tends to be a little tricky, although not really difficult.

```
57      * INITIALIZATION AND LINKAGE
58      START     ENT6  0                      Initialize rI6 for NEXTCHAR.
59                ENTX  0                      Initialize rX for NEXTCHAR.
60                JMP   OUT1                   Start with OUT (cf. exercise 2).
```

```
61    OUT     STJ   INX                    Coroutine linkage
62    OUTX    JMP   OUT1
63    IN      STJ   OUTX
64    INX     JMP   IN1
65            END   START                  ▮
```

This completes the program. The reader should study it carefully, noting in particular how each coroutine can be written independently as though the other coroutine were its subroutine.

The entry and exit conditions for the IN and OUT coroutines mesh perfectly in the above program. In general, we would not be so fortunate, and the coroutine linkage would also include loading and storing appropriate registers. For example, if OUT would destroy the contents of register A, the coroutine linkage would become

```
OUT    STJ   INX
       STA   HOLDA     Store A when leaving IN.
OUTX   JMP   OUT1                                    (4)
IN     STJ   OUTX
       LDA   HOLDA     Restore A when leaving OUT.
INX    JMP   IN1
```

There is an important relation between coroutines and *multiple-pass algorithms*. For example, the translation process we have just described could have been done in two distinct passes: We could first have done just the IN coroutine, applying it to the entire input and writing each character with the proper amount of replication onto magnetic tape. After this was finished, we could rewind the tape and then do just the OUT coroutine, taking the characters from tape in groups of three. This would be called a "two-pass" process. (Intuitively, a "pass" denotes a complete scan of the input. This definition is not precise, and in many algorithms the number of passes taken is not at all clear; but the intuitive concept of "pass" is useful in spite of its vagueness.)

Figure 22(a) illustrates a four-pass process. Quite often we will find that the same process can be done in just one pass, as shown in part (b) of the figure, if we substitute four coroutines A, B, C, D for the respective passes A, B, C, D. Coroutine A will jump to B when pass A would have written an item of output on tape 1; coroutine B will jump to A when pass B would have read an item of input from tape 1, and B will jump to C when pass B would have written an item of output on tape 2; etc.

Conversely, a process done by n coroutines can often be transformed into an n-pass process. Due to this correspondence it is worth while comparing multipass algorithms to one-pass algorithms:

a) *Psychological difference.* A multipass algorithm is generally easier to create and to understand than a one-pass algorithm for the same problem.

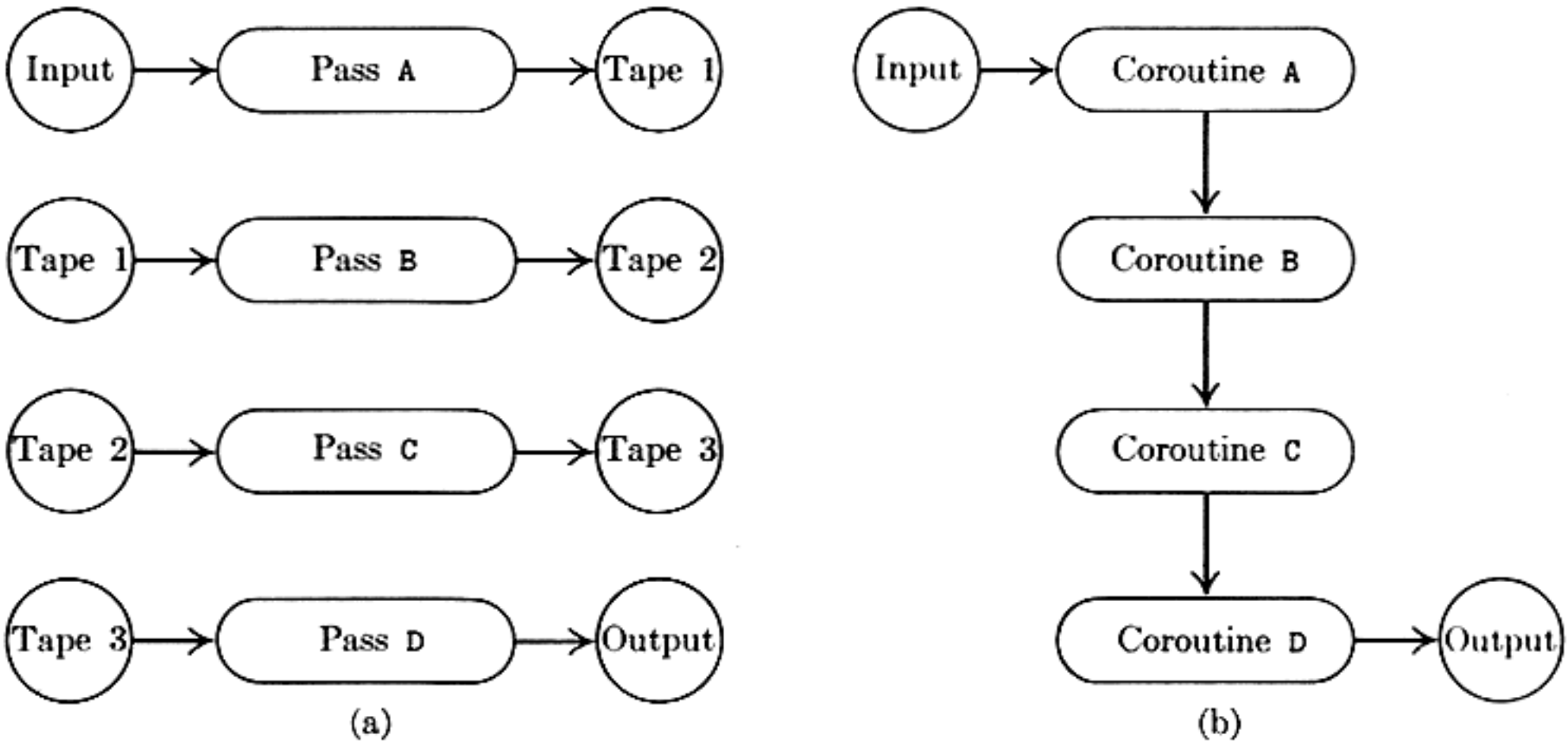


Fig. 22. Passes: (a) a four-pass algorithm, and (b) a one-pass algorithm.

Breaking a process down into a sequence of small steps which happen one after the other is easier to comprehend than considering an involved process in which all of these things go on simultaneously.

Also, if a very large problem is being done and if many people are to cooperate in producing the computer program, a multipass algorithm provides a natural way to divide up the job.

These advantages of a multipass algorithm are present in coroutines as well, since each coroutine can be written essentially separate from the others, and the linkage makes an apparently multipass algorithm into a single-pass process.

b) *Time difference.* The time required to pack, write, read, and unpack intermediate data between passes (e.g., the information in Fig. 22 on tapes) is avoided in a one-pass algorithm. For this reason, a one-pass algorithm will be faster.

c) *Space difference.* The one-pass algorithm requires space to hold all the programs in memory simultaneously, while a multipass algorithm requires space for only one at a time. This may affect the speed, even to a greater extent than indicated in statement (b). For example, many computers have a limited amount of "fast memory" and a larger amount of slower memory; if each pass can fit into the fast memory, the result will be considerably faster than if we use coroutines in a single pass (since the use of coroutines would presumably force most of the program to appear in the slower memory).

Occasionally, there is a need to design algorithms for several computer configurations at once, some of which have larger memory capacity than others. In this case it would be possible to write the program in terms of coroutines, and to let the memory size govern the number of passes: load together as many coroutines as feasible, and supply input or output subroutines for the missing links.

Although this relationship between coroutines and passes is important, we should keep in mind that not all coroutine applications can be split into multipass algorithms. For example, if coroutine B gets input from A and also sends back crucial information to A, it cannot be converted into pass A followed by pass B.

Conversely, it is clear that not all multipass algorithms can be converted to coroutines. Some algorithms are inherently multipass; for example, the second pass may require cumulative information from the first pass (like the total number of occurrences of a certain word in the input). There is an old joke worth noting in this regard:

> *Little old lady, riding a bus. "Little boy, can you tell me how I can get off at Pasadena Street?"*
> *Little boy. "Just watch me, and get off two stops before I do."*

(The joke is that the little boy gives a two-pass algorithm.)

So much for multipass algorithms. We will see further examples of coroutines in numerous places throughout the book, for example, as part of the buffering schemes in Section 1.4.4. Coroutines also play an important role in discrete system simulation; see Section 2.2.5. The important idea of *recursive coroutines* is introduced in Chapter 8, and some interesting applications of this idea may be found in Section 10.6.

EXERCISES

1. [*10*] Explain why short, simple examples of coroutines are hard for the author of a textbook to find.

▶ 2. [*20*] The program in the text starts up the `OUT` coroutine first. What would happen if `IN` were the first to be executed, i.e., if line 60 were changed from "`JMP OUT1`" to "`JMP IN1`"?

3. [*20*] True or false: The three "`CMPA PERIOD`" instructions within `OUT` may all be omitted, and the program would still work.

4. [*20*] Show how coroutine linkage analogous to (1) can be given for several real-life computers you are familiar with.

5. [*15*] Suppose that both coroutines `IN` and `OUT` would want the contents of register A untouched between exit and entry; thus, assume that wherever the instruction "`JMP IN`" occurs within `OUT`, the contents of register A are to be unchanged when control returns to the next line, and make a similar assumption about "`JMP OUT`" within `IN`. What coroutine linkage is needed? [Cf. (4).]

▶ 6. [*22*] Give coroutine linkage analogous to (1) for the case of *three* coroutines, `A`, `B`, `C`, each of which can jump to either of the other two. (Whenever a coroutine is activated, it begins where it last left off.)

▶ 7. [*30*] Write a program which *reverses* the translation done by the program in the text, i.e., it would convert cards punched like (3) into cards punched like (2). The output should be as short a string of characters as possible, so that the zero before the `Z` in (2) would not really be produced from (3).

1.4.3. Interpretive routines.

In this section we will investigate a common type of computer program, the *interpretive routine* (which will be called *interpreter* for short). An interpretive routine is a computer program that performs the instructions of another program, where the other program is written in some machine-like language. By a machine-like language, we mean some way of representing instructions having, say, operation codes, addresses, etc. (This definition, like most definitions of today's computer terms, is not precise, nor should it be; it is impossible to draw the line exactly and to say just which programs are interpreters and which are not.)

Historically, the first interpreters were built around machine-like languages designed especially for simple programming; it was to be a language easier to use than machine language. The rise of programming languages has gradually made this function of interpretive routines obsolete, but interpreters are by no means dying out. On the contrary, their use has been growing, to the extent that effective use of interpretive routines may be regarded as one of the essential characteristics of modern programming. The new applications of interpreters are made chiefly for the following reasons:

a) to represent a fairly complicated sequence of decisions and actions in a compact, efficient manner, and
b) to communicate between passes of a multipass program.

In these cases, special purpose machine-like languages are developed for use in a particular program, and often the machine is the only individual who ever writes "programs" in this language. (Today's expert programmers are also good machine designers, as they not only create the interpretive routine, but also invent the language to be interpreted.)

The interpretive technique has the further advantage of being relatively machine-independent—only the interpreter must be rewritten when changing machines. Furthermore, helpful debugging aids can readily be built in to an interpretive system.

Examples of interpreters of type (a) appear in several places later in this book, e.g., the recursive interpreter in Chapter 8, the "Parsing Machine" interpreter in Section 10.6.4, the `XMIX` interpreter in Section 12.2, and the `TROLL` interpreter in Section 12.3.4.

Another example would be a program in which a great many special cases arise, all similar, but having no really simple pattern. Thus, consider writing a compiler (cf. Chapter 12) in which we would like to generate efficient object programs for adding two quantities together. There might be ten classes of quantities (e.g., constants, simple variables, temp storages, subscripted variables, a quantity in an accumulator or index register, fixed and floating point, etc.) and the combination of all pairs yields 100 different cases. A long program would be required to do the proper thing in each case; the interpretive solution to this problem is to make up a language whose "instructions" fit in one byte. Then

keep a table of 100 "programs" in this language, where each program consists of one to five instructions so it fits in a single word. The idea is merely to pick out the appropriate table entry and to perform the program found there. This technique is simple and efficient.

An example of an interpreter of type (b) appears in the article "Computer-Drawn Flowcharts" by D. E. Knuth, *CACM* **6** (1963), 555–563. In a multipass program, the earlier passes must transmit information to the later passes. This information is often transmitted most efficiently in a somewhat machine-like language, as a set of instructions for the later pass; the later pass is then nothing but a special purpose interpretive routine, and the earlier pass is a special purpose "compiler." This philosophy of multipass operation may be characterized as *telling* the later pass what to do, whenever possible, rather than simply presenting it with a lot of facts and asking it to *figure out* what to do.

Another example of an interpreter of type (b) occurs in connection with compilers for special languages. If the language includes many features which are not easily done on the machine except by subroutine, the resulting object programs will be very long sequences of subroutine calls. This would happen, for example, if the language were concerned primarily with multiple-precision arithmetic. In such a case the object program would be considerably shorter if it were expressed in an interpretive language. An illustration of this approach may be found in Section 12.3, where the TROL language and its interpreter are discussed. See also the book *ALGOL 60 Implementation*, by B. Randell and L. J. Russell (New York: Academic Press, 1964), which describes a compiler to translate from ALGOL 60 into an interpretive language, and which also describes the interpreter for that language; and see "An ALGOL 60 Compiler," by Arthur Evans, Jr., *Ann. Rev. Auto. Programming* **4** (1964), 87–124, for examples of interpretive routines used *within* a compiler.

There is another way to look at a program written in interpretive language—it may be regarded as a series of subroutine calls, one after another. Such a program may in fact be expanded into a long sequence of calls on subroutines, and, conversely, such a sequence can usually be packed into a coded form which is readily interpreted. The advantages of interpretive techniques are the compactness of representation, the machine independence, and the increased diagnostic capability. An interpreter can usually be written so that the amount of time spent in interpretation of the code itself and branching to the appropriate routine is negligible.

1.4.3.1. A MIX simulator. When the language presented to an interpretive routine is the machine language of another computer, the interpreter is often called a simulator.

In the author's opinion, entirely too much programmers' time has been spent in writing such simulators and entirely too much computer time has been wasted in using them. The motivation for simulators is simple: A computer installation buys a new machine and still wants to run programs written for the old machine (rather than rewriting the programs). However, this usually costs

more and gives poorer results than if a special task force of programmers were given temporary employment to do the re-programming. For example, the author once participated in such a re-programming project, and a serious error was discovered in the original program which had been in use for several years; the new program worked at five times the speed of the old, besides giving the right answers for a change! (Not all simulators are bad; for example, it is usually advantageous for a computer manufacturer to simulate a new machine before it has been built, so that software for this machine may be developed as soon as possible. But this is a very specialized application.) An extreme example of the inefficient use of computer simulators is the true story of machine A simulating machine B running a program which simulates machine C! This is the way to make a large, expensive computer give poorer results than its cheaper cousin.

In view of all this, why should a simulator rear its head in this book? There are two reasons:

a) The simulator we will describe below is a good example of a typical interpretive routine; the basic techniques employed in interpreters are illustrated here. It also illustrates the use of subroutines in a moderately long program.

b) We will describe a simulator of the MIX computer, written in (of all things) the MIX language. This will facilitate the writing of MIX simulators for most computers, which are similar; the coding of our program intentionally avoids making heavy use of MIX-oriented features. A MIX simulator will be of advantage as a teaching aid in conjunction with this book and possibly others.

Computer simulators as described in this section should be distinguished from *discrete system simulators*, which are important programs studied in Section 2.2.5.

Now let us turn to the task of writing a MIX simulator. The numbering of MIX's instructions LDA, LD1, . . . , LDX and other similar ones suggests that we keep the simulated contents of these registers in sequential locations, as follows:

```
AREG, I1REG, I2REG, I3REG, I4REG, I5REG, I6REG, XREG, JREG, ZERO.
```

Here ZERO is a "register" filled with zeros at all times. The position of JREG and ZERO is suggested by the operation code numbers of the instructions STJ and STZ.

In keeping with our philosophy of writing the simulator as though it were not done with MIX, we will treat the signs as independent parts of a word. For example, many computers cannot represent the number "minus zero", while MIX definitely can; therefore we will always treat signs specially in this program. The locations AREG, I1REG, . . . , ZERO will always contain the absolute values of the corresponding register contents; another set of locations in our program, called SIGNA, SIGN1, . . . , SIGNZ will contain +1 or −1, depending on whether the sign of the corresponding register is plus or minus.

An interpretive routine generally has a central control section which is called into action between interpreted instructions. In our case, the program transfers to location CYCLE at the end of each simulated instruction.

The control routine does the things common to all instructions, unpacks the instruction into its various parts, and puts the parts into convenient places for later use. The program below sets

rI6 = location of the next instruction;
rI5 = M (address of present instruction, plus indexing);
rI4 = operation code of present instruction;
rI3 = F-field of present instruction;
INST = present instruction.

Program M.

```
01    * MIX SIMULATOR
02              ORIG   3500          (Simulated memory is in locations 0000 up.)
03    BEGIN     STZ    TIME(0:2)
04              STZ    OVTOG         OVTOG is the simulated overflow toggle.
05              STZ    COMPI         COMPI, ±1 or 0, is comparison indicator.
06              ENT6   0             Take first instruction from location zero.
07    CYCLE     LDA    CLOCK         Beginning of control routine
08    TIME      INCA   0             This address is set to execution time
09              STA    CLOCK              of previous instruction, see line 33.
10              LDA    0,6           Instruction to simulate → rA.
11              STA    INST
12              INC6   1             Advance location counter.
13              LDX    INST(1:2)     Get absolute value of address.
14              SLAX   5             Attach sign to address.
15              STA    M
16              LD2    INST(3:3)     Examine index field.
17              J2Z    1F            Is it zero?
18              DEC2   6
19              J2P    INDEXERROR    Illegal index specified?
20              LDA    SIGN6,2       Get sign of index register.
21              LDX    I6REG,2       Get magnitude of index register.
22              SLAX   5             Attach sign.
23              ADD    M             Signed addition for indexing.
24              CMPA   ZERO(1:3)     Is result too large?
25              JNE    ADDRERROR
26              STA    M             Address has been found.
27    1H        LD3    INST(4:4)     F-field → rI3.
28              LD5    M             M → rI5.
29              LD4    INST(5:5)     C-field → rI4.
30              DEC4   63
31              J4P    OPERROR       Is op code ≥ 64?
32              LDA    OPTABLE,4(4:4)  Get execution time from table.
33              STA    TIME(0:2)
34              LD2    OPTABLE,4(0:2)  Get address of proper routine.
35              JNOV   0,2           Jump to operator.
36              JMP    0,2           (Protect against overflows.) ▮
```

The reader's attention is called particularly to lines 34–36: a "switching table" of the 64 operators is part of the simulator, allowing it to jump rapidly

to the correct routine for the current instruction. This is an important time-saving technique (cf. exercise 1.3.2–9).

The 64-word switching table, called OPTABLE, gives also the execution time for the various operators; the following lines indicate the contents of that table:

```
 37              NOP    CYCLE(1)
 38              ADD    ADD(2)
 39              SUB    SUB(2)
 40              MUL    MUL(10)
 41              DIV    DIV(12)
 42              HLT    SPEC(1)
 43              SLA    SHIFT(2)
 44              MOVE   MOVE(1)
 45              LDA    LOAD(2)
 46              LD1    LOAD,1(2)
                 . . .
 51              LD6    LOAD,1(2)
 52              LDX    LOAD(2)
 53              LDAN   LOADN(2)
 54              LD1N   LOADN,1(2)
                 . . .
 60              LDXN   LOADN(2)
 61              STA    STORE(2)
                 . . .
 69              STJ    STORE(2)
 70              STZ    STORE(2)
 71              JBUS   JBUS(1)
 72              IOC    IOC(1)
 73              IN     IN(1)
 74              OUT    OUT(1)
 75              JRED   JRED(1)
 76              JMP    JUMP(1)
 77              JAP    REGJUMP(1)
                 . . .
 84              JXP    REGJUMP(1)
 85              INCA   ADDROP(1)
 86              INC1   ADDROP,1(1)
                 . . .
 92              INCX   ADDROP(1)
 93              CMPA   COMPARE(2)
                 . . .
100    OPTABLE   CMPX   COMPARE(2)
```

Operation code table
typical entry is
"OP address (time)"

(The entries for operators LD*i*, LD*i*N, and INC*i* have an additional ",1" to set the (3:3) field nonzero; this is used below in lines 289–290 to indicate the fact that the size of the quantity within the corresponding index register must be checked after simulating these operations.)

The next part of our simulator program merely lists the locations used to contain the contents of the simulated registers:

```
101   AREG     CON  0     Magnitude of A-register
102   I1REG    CON  0     Magnitude of index registers
103   I2REG    CON  0
104   I3REG    CON  0
105   I4REG    CON  0
106   I5REG    CON  0
107   I6REG    CON  0
108   XREG     CON  0     Magnitude of X-register
109   JREG     CON  0     Magnitude of J-register
110   ZERO     CON  0     Constant zero, for "STZ"
111   SIGNA    CON  1     Sign of A-register
112   SIGN1    CON  1     Sign of index registers
113   SIGN2    CON  1
114   SIGN3    CON  1
115   SIGN4    CON  1
116   SIGN5    CON  1
117   SIGN6    CON  1
118   SIGNX    CON  1     Sign of X-register
119   SIGNJ    CON  1     Sign of J-register
120   SIGNZ    CON  1     Sign stored in "STZ"
121   INST     CON  0     Instruction being simulated
122   COMPI    CON  0     Comparison indicator
123   OVTOG    CON  0     Overflow toggle
124   CLOCK    CON  0     Simulated execution time ▮
```

Now we will consider the various subroutines used by the simulator. First comes the MEMORY subroutine:

Calling sequence: JMP MEMORY.

Entry conditions: rI5 = valid memory address (otherwise subroutine will jump to MEMERROR).

Exit conditions: rX = sign of word in memory location (rI5); rA = magnitude of word in memory location (rI5).

```
125   * SUBROUTINES
126   MEMORY   STJ   9F          Memory fetch subroutine
127            J5N   MEMERROR
128            CMP5  =BEGIN=     Simulated memory is in
129            JGE   MEMERROR        locations 0000 to BEGIN.
130            LDX   0,5
131            ENTA  1
132            SRAX  5           Sign of word → rX.
133            LDA   0,5(1:5)    Magnitude of word → rA.
134   9H       JMP   *           Exit. ▮
```

The FCHECK subroutine processes a partial field specification, making sure it has the form 8L+R with $L \leq R \leq 5$.

Calling sequence: JMP FCHECK.

Entry conditions: rI3 is valid field specification (otherwise subroutine will jump to FERROR).

Exit conditions: rA = rI1 = L, rX = R.

```
135   FCHECK   STJ   9F        Field check subroutine
136            ENTA  0
137            ENTX  0,3       rAX ← field specification.
138            DIV   =8=       Separate into L and R.
139            CMPX  =5=       Is R > 5?
140            JG    FERROR
141            STX   R
142            STA   L
143            LD1   L         rI1 ← L.
144            CMPA  R
145   9H       JLE   *         Exit unless L > R.
146            JMP   FERROR          ▮
```

The last subroutine, GETV, finds the quantity V (i.e., the appropriate field of location M) used in various MIX operators, as defined in Section 1.3.1.

Calling sequence: JMP GETV.

Entry conditions: rI5 = valid memory address; rI3 = valid field.

Exit conditions: rA = magnitude of V; rX = sign of V; rI1 = L; rI2 = −R.

Second entrance: JMP GETAV, used only in comparison operators to extract a field from a register.

```
147   GETAV    STJ   9F        Special entrance, see line 300.
148            JMP   1F
149   GETV     STJ   9F        Subroutine to find V
150            JMP   FCHECK    Process field; L → rI1.
151            JMP   MEMORY    rA = memory magnitude,
                                     rX = sign.
152   1H       J1Z   2F        Is sign part of the field?
153            ENTX  1         If not, set sign positive.
154            SLA   -1,1      Extract off bytes to left
155            SRA   -1,1           of the field.
156   2H       LD2N  R         Shift right into
157            SRA   5,2            proper position.
158   9H       JMP   *         Exit. ▮
```

Now we come to the routines for the individual operators. These routines are given here for completeness, but the reader should study only a few of them

unless he is exceptionally ambitious; those for SUB and JUMP are recommended as typical examples for study. Note how routines for similar operations are neatly combined, and note how the JUMP routine uses another switching table to govern the type of jump.

```
159     * INDIVIDUAL OPERATORS
160     ADD         JMP   GETV        Get value of V in rA, rX.
161                 ENT1  0           Let rI1 indicate the A register.
162                 JMP   INC         Go to "increase" routine.
163     SUB         JMP   GETV        Get value of V in rA, rX.
164                 ENT1  0           Let rI1 indicate the A register.
165                 JMP   DEC         Go to "decrease" routine.
166     *
167     MUL         JMP   GETV        Get value of V in rA, rX.
168                 CMPX  SIGNA       Are signs the same?
169                 ENTX  1
170                 JE    *+2         Set rX to result sign.
171                 ENNX  1
172                 STX   SIGNA       Put it in both simulated registers.
173                 STX   SIGNX
174                 MUL   AREG        Multiply the operands.
175                 JMP   STOREAX     Store the magnitudes.
176     *
177     DIV         LDA   SIGNA       Set sign of remainder.
178                 STA   SIGNX
179                 JMP   GETV        Get value of V in rA, rX.
180                 CMPX  SIGNA       Are signs the same?
181                 ENTX  1
182                 JE    *+2         Set rX to result sign.
183                 ENNX  1
184                 STX   SIGNA       Put it in simulated rA.
185                 STA   TEMP
186                 LDA   AREG        Divide the operands.
187                 LDX   XREG
188                 DIV   TEMP
189     STOREAX     STA   AREG        Store the magnitudes.
190                 STX   XREG
191     OVCHECK     JNOV  CYCLE       Did overflow just occur?
192                 ENTX  1           If so, set simulated
193                 STX   OVTOG             overflow toggle on.
194                 JMP   CYCLE       Return to control routine.
195     *
196     LOADN       JMP   GETV        Get value of V in rA, rX.
197                 ENT1  47,4        rI1 ← C-16; indicates register.
198     LOADN1      STX   TEMP        Negate sign.
199                 LDXN  TEMP
200                 JMP   LOAD1       Change LOADN to LOAD.
201     LOAD        JMP   GETV        Get value of V in rA, rX.
```

```
202              ENT1  55,4          rI1 ← C-8, indicates register.
203   LOAD1      STA   AREG,1        Store magnitude.
204              STX   SIGNA,1       Store sign.
205              JMP   SIZECHECK     Check if magnitude too large.
206   *
207   STORE      JMP   FCHECK        rI1 ← L.
208              JMP   MEMORY        Get contents of memory location.
209              J1P   1F            Is the sign part of the field?
210              ENT1  1             If so, change L to 1
211              LDX   SIGNA+39,4          and "store" sign of register.
212   1H         LD2N  R             rI2 ← —R.
213              SRAX  5,2           Save area to right of field.
214              LDA   AREG+39,4     Insert register in field.
215              SLAX  5,2
216              ENN2  0,1           rI2 ← —L.
217              SRAX  6,2
218              LDA   0,5           Restore area to left of field.
219              SRA   6,2
220              SRAX  -1,1          Attach the sign.
221              STX   0,5           Store in memory.
222              JMP   CYCLE         Return to control routine.
223   *
224   JUMP       DEC3  9             Jump operators
225              J3P   FERROR        Is F too large?
226              LDA   COMPI         Comparison indicator → rA.
227              JMP   JTABLE,3      Jump to appropriate routine.
228   JMP        ST6   JREG          Set simulated J-register.
229              JMP   JSJ
230              JMP   JOV
231              JMP   JNOV
232              JMP   LS
233              JMP   EQ
234              JMP   GR
235              JMP   GE
236              JMP   NE
237   JTABLE     JMP   LE            Jump table
238   JOV        LDX   OVTOG         Check whether to jump on
239              JMP   *+3                 overflow.
240   JNOV       LDX   OVTOG
241              DECX  1             Get complement of overflow toggle.
242              STZ   OVTOG         Shut off overflow toggle.
243              JXNZ  JMP           Jump.
244              JMP   CYCLE         Don't jump.
245   LE         JAZ   JMP           Jump if rA zero or negative.
246   LS         JAN   JMP           Jump if rA negative.
247              JMP   CYCLE         No jump
248   NE         JAN   JMP           Jump if rA negative or positive.
249   GR         JAP   JMP           Jump if rA positive.
```

```
250                 JMP    CYCLE          No jump
251     GE          JAP    JMP            Jump if rA positive or zero.
252     EQ          JAZ    JMP            Jump if rA zero.
253                 JMP    CYCLE          No jump
254     JSJ         JMP    MEMORY         Check for valid memory address.
255                 ENT6   0,5            Simulate a jump.
256                 JMP    CYCLE          Return to main control routine.
257     *
258     REGJUMP     LDA    AREG+23,4      Register jumps
259                 JAZ    *+2            Is register zero?
260                 LDA    SIGNA+23,4     If not, put sign into rA.
261                 DEC3   5
262                 J3NP   JTABLE,3       Change to a conditional JMP unless
263                 JMP    FERROR              F-specification too large.
264     *
265     ADDROP      DEC3   3              Address transfer operators
266                 J3P    FERROR         Is F too large?
267                 ENTX   0,5
268                 JXNZ   *+2            Find sign of M.
269                 LDX    INST
270                 ENTA   1
271                 SRAX   5              rX = sign of M.
272                 LDA    M(1:5)         rA = magnitude of M.
273                 ENT1   15,4           rI1 indicates the register.
274                 JMP    1F,3           Four-way jump.
275                 JMP    INC            Increase.
276                 JMP    DEC            Decrease.
277                 JMP    LOAD1          Enter.
278     1H          JMP    LOADN1         Enter negative.
279     DEC         STX    TEMP           Reverse sign.
280                 LDXN   TEMP           Change to "increase."
281     INC         CMPX   SIGNA,1        Addition routine
282                 JE     1F             Are signs the same?
283                 SUB    AREG,1         No; subtract magnitudes.
284                 JANP   2F             Sign change in register?
285                 STX    SIGNA,1        Change register sign.
286                 JMP    2F
287     1H          ADD    AREG,1         Add magnitudes.
288     2H          STA    AREG,1(1:5)    Store magnitude of result.
289     SIZECHECK   LD1    OPTABLE,4(3:3) Have we just loaded an
290                 J1Z    OVCHECK              index register?
291                 CMPA   ZERO(1:3)      If so, make sure result
292                 JE     CYCLE                fits in two bytes.
293                 JMP    SIZEERROR
294     *
295     COMPARE     JMP    GETV           Get value of V in rA, rX.
296                 SRAX   5              Attach sign.
297                 STX    V
```

```
298              LDA   XREG,4          Get field of appropriate register.
299              LDX   SIGNX,4
300              JMP   GETAV
301              SRAX  5               Attach sign.
302              CMPX  V               Compare (note that -0 = +0).
303              STZ   COMPI           Set comparison indicator to
304              JE    CYCLE                either zero, plus one,
305              ENTA  1                    or minus one.
306              JG    *+2
307              ENNA  1
308              STA   COMPI
309              JMP   CYCLE           Return to control routine
310      *
311              END   BEGIN           ▮
```

The above code adheres to a rather subtle rule that was stated in Section 1.3.1: the instruction "ENTA -0" loads minus zero into register A, as does "ENTA -5,1" when index register 1 contains +5. In general, when M is zero, ENTA loads the sign of the instruction and ENNA loads the opposite sign. The need to specify this condition was overlooked in the presentation in Section 1.3.1. Questions like this usually come to light only when a computer program is being written to follow the rules.

In spite of its length, the above program is incomplete in several respects:

a) It does not recognize floating-point operations.
b) The coding for operation codes 5, 6, and 7 has been left as an exercise.
c) The coding for input-output operators has been left as an exercise.
d) No provision has been made for loading simulated programs (see exercise 4).
e) The error routines

   INDEXERROR, ADDRERROR, OPERROR, MEMERROR, FERROR, SIZEERROR

   have not been included; these are for error conditions which are detected in the simulated program.
f) No provision for diagnostic facilities (e.g., printouts of registers as the program is being executed) has been included.

EXERCISES

1. [*14*] Study all the uses of the `FCHECK` subroutine in the simulator program. Can you suggest a better way to organize the subroutines in this program? (Cf. step 3 in the discussion at the end of Section 1.4.1.)

2. [*20*] Write the `SHIFT` routine, which is missing from the program in the text (operation code 6).

▶ **3.** [*22*] Write the `MOVE` routine, which is missing from the program in the text (operation code 7).

4. [*14*] Change the program in the text so that it begins as though MIX's "GO-button" had been pushed (cf. exercise 1.3.1–26).

▶ 5. [*24*] Determine the time required to simulate the LDA and ENTA operators, compared with the actual time for MIX to execute these operators directly.

6. [*28*] Write programs for the input-output operators JBUS, IOC, IN, OUT, and JRED, which are missing from the program in the text, allowing only units 16 and 18. Assume that a card read or a skip to new page takes $10000u$ and a print takes $7500u$. (*Note:* Experience shows that the JBUS instruction should be simulated by treating "JBUS *" as a special case; otherwise the simulator seems to stop!)

▶ 7. [*32*] Modify the solutions of the previous exercise in such a way that execution of IN or OUT does not cause I/O transmission immediately; the transmission should take place after approximately half of the time required by the simulated devices has elapsed. (This will prevent a frequent student error of improperly using the IN and OUT operators.)

***1.4.3.2. Trace routines.** When a machine is being simulated on itself (as MIX was simulated on MIX in the previous section) we have the special case of a simulator called a *trace* or *monitor* routine. Such programs are occasionally used to help in debugging, since they print out a step-by-step account of how the simulated program behaves.

The program in the preceding section was written as though another computer were simulating MIX. A quite different approach is used for trace programs; we generally let registers represent themselves and let the operators perform themselves.

In a trace program we usually contrive to let the machine execute most of the instructions; the exception is a jump or conditional jump instruction which must not be executed without modification (for the trace program would lose control). Each machine also has its own idiosyncrasies which make tracing more of a challenge; in MIX's case, this is the J-register.

The trace routine given below is initiated when the main program jumps to location ENTER with register J set to the address for *starting* to trace, register X set to the address where tracing should *stop*. The program is interesting and merits careful study.

```
01    * TRACE ROUTINE
02    ENTER   STX   TEST(0:2)      Set exit location.
03            STX   LEAVEX(0:2)
04            STA   AREG           Save contents of rA.
05            STJ   JREG(0:5)      Save contents of rJ.
06            LDA   JREG           Get start location for trace.
07    CYCLE   STA   PREG(0:2)      Store location of next instruction.
08    TEST    DECA  *              Is it the exit location?
09            JAZ   LEAVE
10    PREG    LDA   *              Get next instruction.
11            STA   INST           Make two copies
12            STA   INST1
```

```
13          LDA   INST(5:5)     Get operation code, C.
14          DECA  38
15          JANN  1F            Is C ≥ 38?
16          INCA  6
17          JANZ  2F            Is C ≠ 32?
18          LDA   INST(0:4)     C = 32 (STJ).
19          STA   *+2(0:4)      Changed to STA.
20          LDA   JREG
21          STA   *
22          JMP   INCP
23    2H    DECA  2
24          JANZ  2F            C ≠ 34?
25          JMP   3F            C = 34 (JBUS).
26    1H    DECA  9             Test for jump instructions.
27          JAP   2F            C ≥ 48?
28    3H    LDA   8F(0:3)       Jump instruction detected;
29          STA   INST(0:3)     its address is changed to "JUMP".
30    2H    LDA   AREG          Restore register A.
31    *                         All registers except J now have proper
32    *                         values with respect to the external program.
33    INST  NOP   *             The instruction is executed.
34          STA   AREG          Store register A again.
35    INCP  LDA   PREG(0:2)     Move to next instruction.
36          INCA  1
37          JMP   CYCLE
38    8H    JSJ   JUMP          Constant for lines 28, 39
39    JUMP  LDA   8B(4:5)       A jump has occurred.
40          SUB   INST(4:5)     Was it JSJ?
41          JAZ   *+4
42          LDA   PREG(0:2)     If not, update simulated
43          INCA  1                   J-register.
44          STA   JREG
45          LDA   INST1(0:3)    Compute indexed address.
46          STA   *+1(0:3)
47          ENTA  *
48          JMP   CYCLE         Move to this instruction.
49    LEAVE LDA   AREG          Restore A-register.
50    LEAVEX JMP  *             Stop tracing.
51    AREG  CON   0             External rA contents
52    JREG  CON   0             External rJ contents
53    INST1 CON   0             Copy of instruction ▮
```

The following things should be noted about trace routines in general and this one in particular:

1) We have presented only the most interesting part of a trace program, the part that retains control while executing another program. For a trace to be useful, there must also be a routine for writing out the contents of registers,

and this has not been included. Such a routine distracts from the more subtle features of a trace program, although it certainly is important; the necessary modifications are left as an exercise (see exercise 2).

2) Space is generally more important than time, i.e., the program should be written to be as short as possible. This is done so the trace routine can coexist with large programs, and the running time is consumed by output anyway.

3) Care was taken to avoid destroying the contents of most registers; in fact, the program uses only MIX's A-register. Neither the comparison indicator nor the overflow toggle are affected by the trace routine. (The less things we use, the less things we need to restore.)

4) When a jump to location JUMP occurs, it is not necessary to "STA AREG", since rA cannot have changed.

5) After leaving the trace routine, the J-register is not reset properly. Exercise 1 shows how to remedy this.

6) The program being traced is subject to only three restrictions: (a) It must not store anything into the locations used by the trace program. (b) It must not use the output device on which tracing information is being recorded (for example, JBUS would give an improper indication). (c) The program is executed at a different rate of speed when tracing.

EXERCISES

1. [*22*] Modify the trace routine of the text so it restores register J when leaving. (You may assume register J is not zero.)

2. [*26*] Modify the trace routine of the text so that before executing each program step it writes the following information on tape unit 0. Use a "buffer swapping" method (see Section 1.4.4) for speed.

Word 1: location.
Word 2: instruction.
Word 3: register A (before execution).
Word 4: register X (before execution).
Words 5–10: registers I1–I6 (before execution).
Word 11: register J.
Word 12: 1 if overflow is on, 0 otherwise.
Word 13: +1 if comparison is greater, 0 if equal, −1 if less.
Words 14–100: not used.

3. [*10*] The previous exercise suggests having the trace program write its output onto tape. Discuss why this would be preferable to printing directly.

▶ **4.** [*25*] What would happen if the trace routine were tracing *itself*? Specifically, consider the behavior if the two instructions

```
ENTX  0
JMP   *+1
```

were placed just before ENTER.

5. [*28*] In a manner similar to that used to solve the previous exercise, consider the situation in which two copies of the trace routine are placed in different places in memory, and each is set up to trace the other. What would happen?

▶ **6.** [*40*] Design a trace routine which is capable of tracing itself, in the sense of exercise 4; i.e., it should print out the steps of its own program at slower speed, and that program will be tracing itself at still slower speed, ad infinitum until memory capacity is exceeded.

1.4.4. Input and Output

Perhaps the most outstanding differences between one computer and the next are the facilities available for doing input and output, and the computer instructions which govern these peripheral devices. We cannot hope to discuss in a single book all of the problems and techniques that arise in this area, so we will confine ourselves to a study of typical input-output methods which apply to most computers. The input-output operators of MIX represent a compromise between the widely varying facilities available in actual machines; to give an example of how to think about input-output, let us discuss in this section the problem of getting the best MIX input-output.

Many computer users feel that input and output are not actually part of "real programming," they are merely things that (unfortunately) must be done in order to get information in and out of the machine. For this reason, the input and output facilities of a computer are usually not learned until after all other features have been examined, and it frequently happens that only a small fraction of the programmers of a particular machine ever know much about the details of input and output. This attitude is present to a certain extent in this book, where we generally give only a light treatment to input-output topics. Hopefully some day another book, covering the principles of executive systems as this book covers other aspects of software systems, will be written, giving a really adequate treatment of input-output control routines.

A brief digression about terminology is perhaps appropriate here. Although contemporary dictionaries seem to regard the words "input" and "output" only as nouns (e.g., "What kind of input are we getting?"), it is now customary to use them grammatically as adjectives (e.g., "Don't drop the input tape.") and as transitive verbs (e.g., "Why did the program output this garbage?"). The combined term "input-output" is most frequently referred to by the abbreviation "I/O". Inputting is often called *reading*, and outputting is, similarly, called *writing*. The stuff that is input or output is generally known as "data"—this word is, strictly speaking, a plural form of the word "datum," but it is used collectively as if it were singular (e.g., "The data has not been read."). This completes today's English lesson.

Suppose now that we wish to read from magnetic tape. The IN operator of MIX, as defined in Section 1.3.1, merely *initiates* the input process, and the computer continues to execute further instructions while the input is taking place. Thus the instruction "IN 1000(5)" will begin to read 100 words from

tape unit number 5 into memory cells 1000–1099, but the ensuing program must not refer to these memory cells until later. The input will be complete only after (a) another I/O operation (IN, OUT, or IOC) referring to unit 5 has been initiated, or (b) the conditional jump instructions JBUS(5) or JRED(5) indicate that unit 5 is no longer "busy."

The simplest way to read a tape record into locations 1000–1099 and to have the information present is therefore the sequence of two instructions

```
IN     1000(5)
JBUS   *(5)                                    (1)
```

We have used this rudimentary method in the program of Section 1.4.2 (see lines 07–08 and 52–53). The method is generally wasteful of computer time, however, because a very large amount of potentially useful calculating time, say $1000u$ or even $10000u$, is consumed by the repeated execution of the "JBUS" instruction. The program's running speed can be as much as doubled if this additional time is utilized for calculation. (See exercises 4 and 5.)

One way to avoid wasting this computation time is to have two areas of memory used for the input; we would read into one area, and while this is going on, the program would compute from the data in the other area. For example, suppose the program begins with the instruction:

```
IN     2000(5)      Begin reading first record.      (2)
```

Subsequently, whenever a tape record is desired we may now give the following five commands:

```
ENT1   1000         Prepare for MOVE operator.
JBUS   *(5)         Wait until unit 5 is ready.
MOVE   2000(50)     (2000–2049) → (1000–1049).       (3)
MOVE   2050(50)     (2050–2099) → (1050–1099).
IN     2000(5)      Begin reading next record.
```

These have the same overall effect as (1).

This program begins to read a tape record into locations 2000–2099 before the preceding record has been examined. This is called "reading ahead" or *anticipated input*—it is done on faith that the record will eventually be needed. In fact, however, we might learn (by examining the contents of the record moved to 1000–1099) that no more input is really required. For example, consider the analogous situation in the coroutine program of Section 1.4.2, where the input was coming from punched cards instead of tape: a "." appearing anywhere in the card meant that it was the final card of the deck. Such a situation would make anticipated input impossible, unless we would assume that either (a) a blank card or special trailer card of some other sort must follow the input deck, or (b) an identifying mark (e.g. ".") must appear in column 80

of the final card of the deck. Some means for properly terminating the input at the end of the program must always be provided whenever input is anticipated.

The technique of overlapping computation time and I/O time is known as *buffering*. The rudimentary method (1) is called "unbuffered" input. The area of memory 2000–2099 used to hold the anticipated input in (3), as well as the area 1000–1099 to which the input was moved, is called a "buffer." Webster's New World Dictionary defines "buffer" as "any person or thing that serves to lessen shock," and the term is appropriate because buffering tends to keep I/O devices running smoothly. (Computer engineers often use the word "buffer" in another sense, to denote a part of the I/O device which stores information during the transmission, but in this book "buffer" will signify an area of *memory* used by a programmer to hold I/O data.)

The sequence (3) is not always superior to (1), although the exceptions are rare. Let us compare the execution times; suppose T is the time required to input 100 words, and suppose C is the computation time which intervenes between input requests. Method (1) requires a time of essentially $T + C$ per tape record, while method (3) takes essentially $\max(C, T) + 202u$. (The quantity $202u$ is the time required by the two MOVE instructions.) One way to look at this running time is to consider so-called "critical path time," in this case, the amount of time the I/O unit is idle between uses. Method (1) keeps the unit idle for C units of time, while method (3) keeps it idle for 202 units (assuming $C < T$).

The relatively slow MOVE commands of (3) are undesirable, particularly because they take up critical path time when the tape unit must be inactive. An almost obvious improvement of the method allows us to avoid these MOVE instructions: the outside program can be revised so that it refers alternately to locations 1000–1099 and 2000–2099; while we are reading into one buffer area, we can be computing with the information in the other. This is the important technique known as *buffer swapping*. The location of the current buffer of interest will be kept in an index register (or, if no index registers are available, in a memory location). We have already seen an example of buffer swapping applied to output in Algorithm 1.3.2P (see steps P9–P11) and the accompanying program.

As an example of buffer swapping on input, suppose that we have a computer application in which each tape record consists of 100 separate one-word items. The following program is a subroutine which gets the next word of input, and which reads in a new record if the current one is exhausted.

```
01   WORDIN  STJ   1F            Store exit location.
02           INC6  1             Advance to next word.
03   2H      LDA   0,6           Is it the end of the
04           CMPA  =SENTINEL=          buffer?
05   1H      JNE   *             If not, exit.                (4)
06           IN    -100,6(U)     Refill this buffer.
07           LD6   1,6           Get address of other
08           JMP   2B                  buffer and return. ▌
```

In this program, index register 6 is used to address the last word of input; we assume that the outside program does not affect this register. The symbol U refers to a tape unit, and the symbol SENTINEL refers to a value which is known (from characteristics of the program) to be *absent* from all tape records. The subroutine is accompanied by the following layout of buffers:

```
09    INBUF1  ORIG  *+100      First buffer
10            CON   SENTINEL   'Sentinel' at end of buffer
11            CON   *+1        Address of other buffer
12    INBUF2  ORIG  *+100      Second buffer
13            CON   SENTINEL   'Sentinel' at end of buffer
14            CON   INBUF1     Address of other buffer ▮
```

Several things about this program should be noted:

1) The "sentinel" constant appears as the 101st word of each buffer, and it makes a convenient test for the end of the buffer. In many applications, however, this technique will not be reliable, since any word may appear on tape. If we were doing card input, a similar technique (with the 17th word of the buffer equal to a sentinel) can always be used; in this case, any negative word can serve as a sentinel, since input from cards always gives nonnegative words.

2) Each buffer contains the address of the other buffer (see lines 07, 11, and 14). This "linking together" facilitates the swapping process.

3) No "JBUS" instruction was necessary, since the next input was initiated before any word of the previous record was accessed. If the quantities C and T refer as before to computation time and tape time, the execution time per tape record is now $\max(C, T)$; it is therefore possible to keep the tape going at full speed if $C < T$. (*Note:* MIX is an idealized computer in this regard, however, since no I/O errors must be treated by the program. On most computers some instructions to test the successful completion of the previous operation would be necessary just before the "IN" instruction here.)

4) To make this subroutine work properly, it will be necessary to get things started out right when the program begins. Details are left to the reader (see exercise 6).

5) The WORDIN subroutine makes the tape unit appear to have a record length of 1 rather than 100 as far as the rest of the program is concerned. The idea of having several program-oriented records filling a single actual tape record is called "blocking of records."

The techniques which we have illustrated for input apply, with minor changes, to output as well (see exercises 2 and 3).

Multiple buffers. Buffer swapping is just the special case $N = 2$ of a general method involving N buffers. In some applications it is desirable to have more than two buffers; for example, consider the following type of algorithm:

Step 1. Read in 5 records in rapid succession.

Step 2. Perform a fairly long calculation based on this data.
Step 3. Return to step 1.

Here five or six buffers would be desirable, so that the next batch of five records could be read during step 2. This tendency for I/O activity to be "bunched" makes multiple buffering an improvement over buffer swapping. Multiple buffering is also desirable for those devices (such as certain UNIVAC card readers) on which the IN instruction initiates the input process but several additional INs (initiating the input process for further records) may be given before the first record is available; then it is necessary to have multiple buffers in order to have any chance of running the input device at a reasonable speed.

Suppose we have N buffers for some input or output process using a single I/O device; we will think of them as if they were arranged in a circle as shown in Fig. 23. So far as this I/O unit is concerned, the program external to the buffering process will be assumed to have the following form:

```
    ⋮
ASSIGN
    ⋮
RELEASE
    ⋮
ASSIGN
    ⋮
RELEASE
    ⋮
```

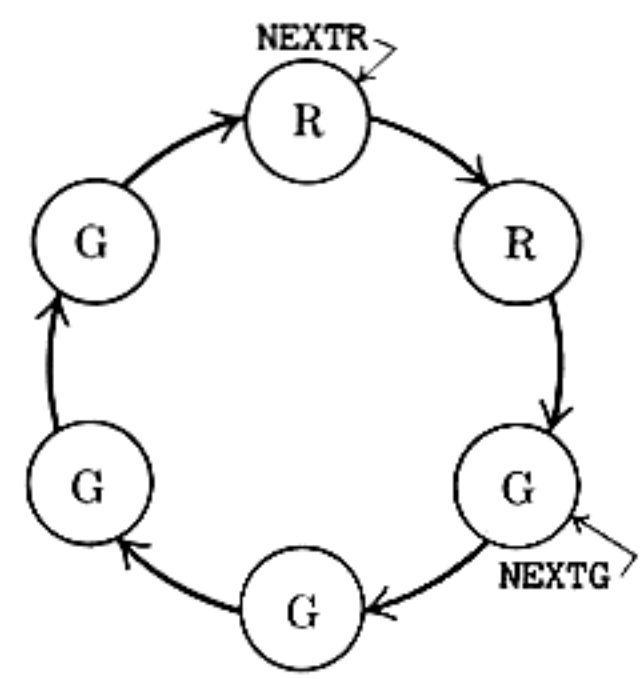


Fig. 23. A circle of buffers ($N = 6$).

i.e., alternate actions of "ASSIGN" and "RELEASE", separated by other computation which does not affect the buffer manipulations for this device.

ASSIGN means that the program acquires the address of the next buffer area, i.e., this address is assigned as the value of some program variable.

RELEASE means the program is done with the current buffer area.

Between ASSIGN and RELEASE the program is communicating with one of the buffers, called the *current* buffer area; between RELEASE and ASSIGN, the program makes no reference to any buffer area.

Conceivably, ASSIGN could immediately follow RELEASE, and discussions of buffering have often been based on this assumption. However, if RELEASE is done as soon as possible, the buffering process has more freedom and will be more effective; by separating the two essentially different functions of ASSIGN and RELEASE we will find the buffering technique is simpler to understand, and our discussion will be meaningful even if $N = 1$.

To be more explicit, let us consider the cases of input and output separately. For input, suppose we are dealing with a card reader. The action ASSIGN means the program needs the information from a new card; we would like to set an index register to the memory address at which the next card image is located. The action RELEASE occurs when the information in the current card image is no longer needed—it has somewhere been digested by the program, perhaps copied to another part of memory, etc. The current buffer area may therefore be filled with further anticipated input.

For output, consider the case of a printer. The action ASSIGN occurs when a free buffer area is needed, into which a line image is to be placed for printing. We wish to set an index register equal to the memory address of such an area. The action RELEASE occurs when this line image has been fully set up in the buffer area, in a form ready to be printed.

Example. To print the contents of locations 0800–0823, we might write

```
JMP    ASSIGNP      (Sets rI5 to buffer location)
ENT1   0,5                                          (5)
MOVE   800(24)      Move 24 words into buffer.
JMP    RELEASEP
```

where ASSIGNP and RELEASEP represent subroutines to do the two buffering functions for the printer.

In an optimal situation (from the standpoint of the computer), the ASSIGN operation will require virtually no execution time. This would mean, on input, that each card image has been anticipated, so it is available when the program is ready for it; and on output, it would mean that there always is a free place in memory to record the line image. No time will be spent waiting for the I/O device.

To help describe the buffering algorithm, and to make it more colorful, we will say buffer areas are either "green," "yellow," or "red" (shown as G, Y, and R in Fig. 24).

Green means that the area is ready to be ASSIGNed; this means it has been filled with anticipated information [in an input situation], or that it is a free area [in an output situation].

Yellow means that the area has been ASSIGNed, not RELEASEd; this means it is the current buffer, and the program is communicating with it.

Red means that the area has been RELEASEd; thus it is a free area (in an input situation) or it has been filled with information (in an output situation).

Figure 23 shows two "pointers" associated with the circle of buffers. These are, conceptually, index registers in the program. NEXTG and NEXTR point to the "next green" and "next red" buffer, respectively. A third pointer, CURRENT (shown in Fig. 24), indicates the yellow buffer when one is present.

Although the algorithm applies equally well to output, we will first consider the case of input from a card reader for definiteness. Suppose a program has

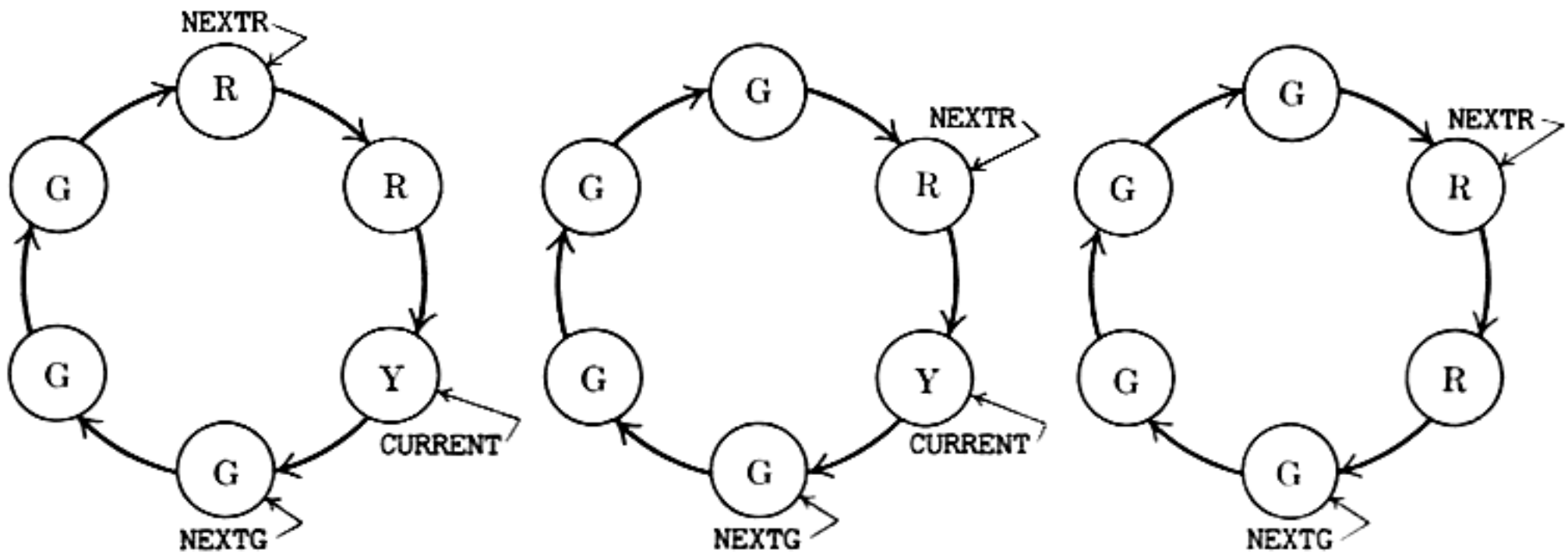


Fig. 24. Buffer transitions, (a) after ASSIGN, (b) after I/O complete, and (c) after RELEASE.

reached the state shown in Fig. 23. This means that four card images have been anticipated by the buffering process, and they reside in the green buffers. At this moment, two things are happening *simultaneously:* (a) The program is computing, following a RELEASE operation; (b) a card is being read into the buffer indicated by NEXTR. This state of affairs will continue until the input cycle is completed (the unit will then go from "busy" to "ready"), or until the program does an ASSIGN operation. Suppose the latter occurs: then the buffer indicated by NEXTG changes to yellow (it is assigned as the current buffer), NEXTG moves clockwise, and we arrive at the position shown in Fig. 24(a). If now the input is completed, another anticipated record is present so the buffer changes from red to green and NEXTR moves over as shown in Fig. 24(b). If the RELEASE operation follows next, we obtain Fig. 24(c).

For an example concerning output, see Fig. 27 on page 223. That illustration shows the "colors" of buffer areas as a function of time, in a program that opens with four quick outputs, then produces four at a slow pace, and finally issues two in rapid succession as the program ends. Three buffers appear in that example.

The pointers NEXTR and NEXTG proceed merrily around the circle, each at an independent rate of speed, moving clockwise. It is a race between the program (which turns buffers from green to red) and the I/O buffering process (which turns them from red to green). Two situations of conflict can occur:

a) if NEXTG tries to pass NEXTR, the program has gotten ahead of the I/O device and it must wait until the device is ready.
b) if NEXTR tries to pass NEXTG, the I/O device has gotten ahead of the program and we must shut it down until the next RELEASE is given.

Both of these situations are depicted in Fig. 27. (See exercise 9.)

Fortunately, in spite of the rather lengthy explanation just given of the ideas behind a circle of buffers, the actual algorithms for handling the situation are

very simple. In the following description,

$$\begin{aligned} N &= \text{total number of buffers;} \\ n &= \text{current number of red buffers.} \end{aligned} \tag{6}$$

The variable n is used in the algorithm below to avoid interference between NEXTG and NEXTR.

Algorithm A (ASSIGN *action*). This algorithm includes the steps implied by ASSIGN within a computational program, as described above.

A1. [Wait for $n < N$.] If $n = N$, stall the program until $n < N$. (If $n = N$, no buffers are ready to be assigned; but Algorithm B below, which runs in parallel to this one, will eventually succeed in producing a green buffer.)

A2. [CURRENT ← NEXTG.] Set CURRENT ← NEXTG (thereby assigning the current buffer).

A3. [Advance NEXTG.] Advance NEXTG to the next clockwise buffer. ▮

Algorithm R (RELEASE *action*). This algorithm includes the steps implied by RELEASE within a computational program, as described above.

R1. [Increase n.] Increase n by one. ▮

Algorithm B (*Buffer control*). This algorithm performs the actual initiation of I/O operators in the machine; it is to be executed "simultaneously" with the main program, in the sense described below.

B1. [Compute.] Let the main program compute for a short period of time; step B2 will be executed after a certain time delay, at a time when the I/O device is ready for another operation.

B2. [$n = 0$?] If $n = 0$, go to B1. (Thus, if no buffers are red, no I/O action can be performed.)

B3. [Initiate I/O.] Initiate transmission between the buffer area designated by NEXTR and the I/O device.

B4. [Compute.] Let the main program run for a period of time; then go to step B5 when the I/O operation is completed.

B5. [Advance NEXTR.] Advance NEXTR to the next clockwise buffer.

B6. [Decrease n.] Decrease n by one, and go to B2. ▮

In these algorithms, we have two independent processes which are going on "simultaneously": the buffering control program and the computation program. These are, in fact, *coroutines*, which we will call CONTROL and COMPUTE. Coroutine CONTROL jumps to COMPUTE in steps B1 and B4; coroutine COMPUTE jumps to CONTROL by interspersing "jump ready" instructions at sporadic intervals in its program.

Coding this algorithm for MIX is extremely simple. For convenience, assume that the buffers are linked so that the word *preceding* each one is the address of the next; i.e., if $N = 3$,

$$\texttt{CONTENTS(BUF1-1)} = \texttt{BUF2},$$
$$\texttt{CONTENTS(BUF2-1)} = \texttt{BUF3}, \quad \text{and} \quad \texttt{CONTENTS(BUF3-1)} = \texttt{BUF1}.$$

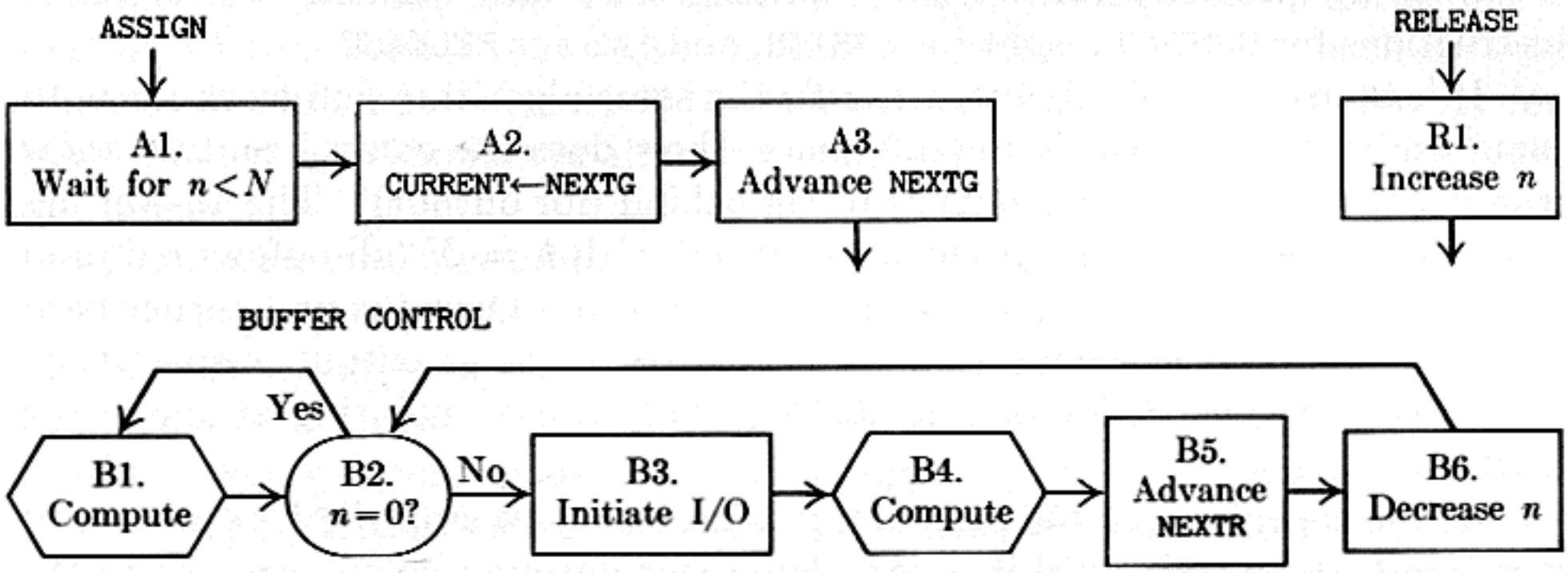


Fig. 25. Algorithms for multiple buffering.

Program A (ASSIGN, *a subroutine within the* COMPUTE *coroutine*). rI4 ≡ CURRENT; rI6 ≡ n; calling sequence is JMP ASSIGN; on exit, rX contains NEXTG.

```
ASSIGN  STJ   9F            Subroutine linkage
1H      JRED  CONTROL(U)    A1. Wait for n < N.
        CMP6  =N=
        JE    1B
        LD4   NEXTG         A2. CURRENT ← NEXTG.
        LDX   -1,4          A3. Advance NEXTG.
        STX   NEXTG
9H      JMP   *             Exit. ▮
```

Program R (RELEASE, *code used within the* COMPUTE *coroutine*). rI6 ≡ n. This short code is to be inserted wherever RELEASE is desired.

```
        INC6  1             R1. Increase n.
        JRED  CONTROL(U)    Possible jump to CONTROL coroutine ▮
```

Program B (*The* CONTROL *coroutine*). rI6 ≡ n, rI5 ≡ NEXTR.

```
CONT1   JMP   COMPUTE       B1. Compute.
1H      J6Z   *-1           B2. n = 0?
        IN    0,5(U)        B3. Initiate I/O.
        JMP   COMPUTE       B4. Compute.
        LD5   -1,5          B5. Advance NEXTR.
        DEC6  1             B6. Decrease n.
        JMP   1B                ▮
```

Besides the above code, we also have the usual coroutine linkage

```
CONTROL   STJ  COMPUTEX        COMPUTE   STJ  CONTROLX
CONTROLX  JMP  CONT1           COMPUTEX  JMP  COMP1
```

and the instruction "JRED CONTROL(U)" is to be placed within COMPUTE about once in every fifty instructions.

Thus the programs for multiple buffering essentially amount to only seven instructions for CONTROL, eight for ASSIGN, and two for RELEASE.

It is perhaps remarkable that *exactly* the same algorithm will work for both input and output. What is the difference—how does the control routine know whether to anticipate (for input) or to lag behind (for output)? The answer lies in the initial conditions: for input we start out with $n = N$ (all buffers red) and for output we start out with $n = 0$ (all buffers green). Once the process has been started properly, it continues to behave as either input or output, respectively. The other initial condition is that NEXTR = NEXTG, both pointing at one of the buffers.

At the conclusion of the program, it is necessary to stop the I/O process (if it is input) or to wait until it is completed (for output); details are left to the reader (see exercises 12 and 13).

It is important to ask what is the best value of N to use. Certainly as N gets larger, the speed of the program will not decrease, but it will not increase indefinitely either and so we come to a point of diminishing returns. Let us refer again to the quantities C and T, representing computation time between I/O operators and the I/O time itself. More precisely, let C be the amount of time between successive ASSIGNs, and let T be the amount of time needed to transmit one record. If C is always *greater* than T, then $N = 2$ is adequate, for it is not hard to see that with two buffers we keep the computer busy at all times. If C is always *less* than T, then again $N = 2$ is adequate, for we keep the I/O device busy at all times. Larger values of N are therefore useful only when C varies between small values and large values; one plus the average number of consecutive small values may be right for N, if the large values of C are significantly longer than T. (However, the advantage of buffering is virtually nullified if all input occurs at the beginning of the program and if all output occurs at the end.) If the time between ASSIGN and RELEASE is always quite small, the value of N may be decreased by 1 throughout the above discussion, with little effect on running time.

The above approach to buffering can be adapted in many ways, and we will mention a few of these briefly. So far we have assumed only one I/O device was being used; in practice, of course, several will be in use at the same time.

There are several ways to approach the subject of multiple units. In the simplest case, we can have a separate circle of buffers for each device. There will be values of n, N, NEXTR, NEXTG, and CURRENT, and a different CONTROL coroutine for each unit. This will give efficient buffering action simultaneously on each I/O device.

It is also possible to "pool" buffer areas which are of the same size, i.e., to have two or more devices sharing buffers from a common list. This would be handled by using the linked memory techniques of Chapter 2: all red input buffers and green output buffers would be linked together. It becomes necessary to distinguish between input and output in this case, and to rewrite the algorithms without using n amd N. The algorithm may get irrevocably stuck if all buffers in the pool are filled with anticipated input, so a check should be made that at all times there is at least one buffer (preferably one for each device) which is not input-green; only if the COMPUTE routine is stalled at step A1 for some input device should we allow input into the final buffer of the pool from this device.

Some machines have additional constraints on the use of input-output units, so that it is impossible to be transmitting data from certain pairs of devices at the same time. (For example, several units might be attached to the computer by means of a single "channel.") This constraint also affects our buffering routine; when we must choose which I/O unit to initiate first, how is the choice to be made? This is called "forecasting." The best forecasting rule for the general case would seem to give preference to the unit whose buffer circle has the largest value of n/N, assuming the number of buffers in the circles has been wisely chosen.

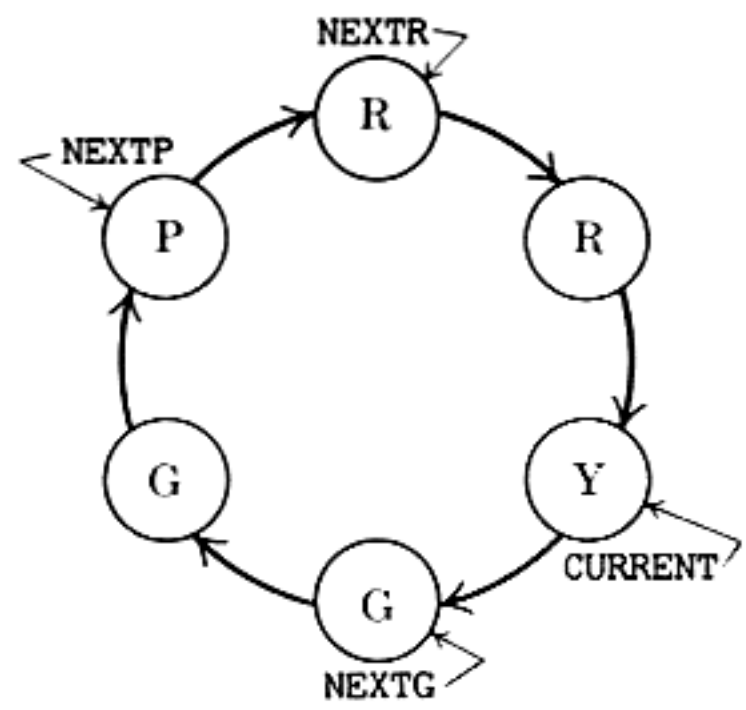


Fig. 26. Input and output from the same circle.

To conclude this discussion, we will mention a useful method for doing input and output from the same buffer circle, under certain conditions. In Fig. 26 we have added another color of buffer (purple). In this situation, green buffers represent anticipated *input;* the program ASSIGNs and a green buffer becomes yellow, then upon RELEASE it turns red and represents a record to be *output.* The input and output processes follow around the circle independently as before, except now we turn red buffers to purple after the output is done, and convert purple to green on input. It is necessary to ensure that none of the pointers NEXTG, NEXTR, NEXTP passes another. At the instant shown in Fig. 26, the program is computing between ASSIGN and RELEASE, using the yellow buffer; simultaneously, input is going into the buffer indicated by NEXTP; and output is coming from the buffer indicated by NEXTR.

EXERCISES

1. [*05*] Would sequence (3) still be correct if the MOVE instructions were placed before the JBUS instruction instead of after it? What if the MOVE instructions were placed after the IN command?

2. [*10*] The instructions

```
OUT    1000(6)
JBUS   *(6)
```

may be used to output a tape record in an unbuffered fashion, just as the instructions (1) did this for input. Give a method analogous to (2) and (3) which buffers this output, by using MOVE instructions and an auxiliary buffer in locations 2000–2099.

▸ **3.** [*22*] Write a buffer-swapping output subroutine analogous to (4). The subroutine, called WORDOUT, should store the word in rA as the next word of output, and if a buffer is full it should write 100 words onto tape unit V. Index register 5 should be used to refer to the current buffer position. Before storing any words into a buffer, it should be cleared to zeros. Show the layout of buffer areas and explain what instructions (if any) are necessary at the beginning and end of the program to ensure that the first and last records are properly written.

4. [*M20*] Show that if a program refers to a single I/O device, it is possible to double the running speed by buffering the I/O, in favorable circumstances, but it is not possible to improve the running speed over the amount of time taken by unbuffered I/O by more than a factor of two.

▸ **5.** [*M21*] Generalize the situation of the preceding exercise to the case when the program refers to n I/O devices instead of just one.

6. [*12*] What instructions should be placed at the beginning of a program so that the WORDIN subroutine (4) gets off to the right start? (For example, index register 6 must be set to *something*.)

7. [*22*] Write a subroutine called WORDIN which is essentially like (4) except that it does not make use of a "sentinel."

8. [*11*] The text describes a hypothetical input situation which leads from Fig. 23 through parts (a), (b), and (c) of Fig. 24. Interpret the same situation given that output to the printer is being done, instead of input from cards. (For example, what things are happening at the time shown in Fig. 23?)

▸ **9.** [*21*] A program which leads to the buffer contents shown in Fig. 27 may be characterized by the following list of times:

A, 1000, R, 1000, A, 1000, R, 1000, A, 1000, R, 1000, A, 1000, R, 1000,
A, 7000, R, 5000, A, 7000, R, 5000, A, 7000, R, 5000, A, 7000, R, 5000,
A, 1000, R, 1000, A, 2000, R, 1000.

This list means "assign, compute for 1000u, release, compute for 1000u, assign, . . . , compute for 2000u, release, compute for 1000u." The computation times given do not include any intervals during which the computer might have to wait for the output device to catch up (as at the fourth "assign" in Fig. 27). The output device operates at a speed of 7500u per record.

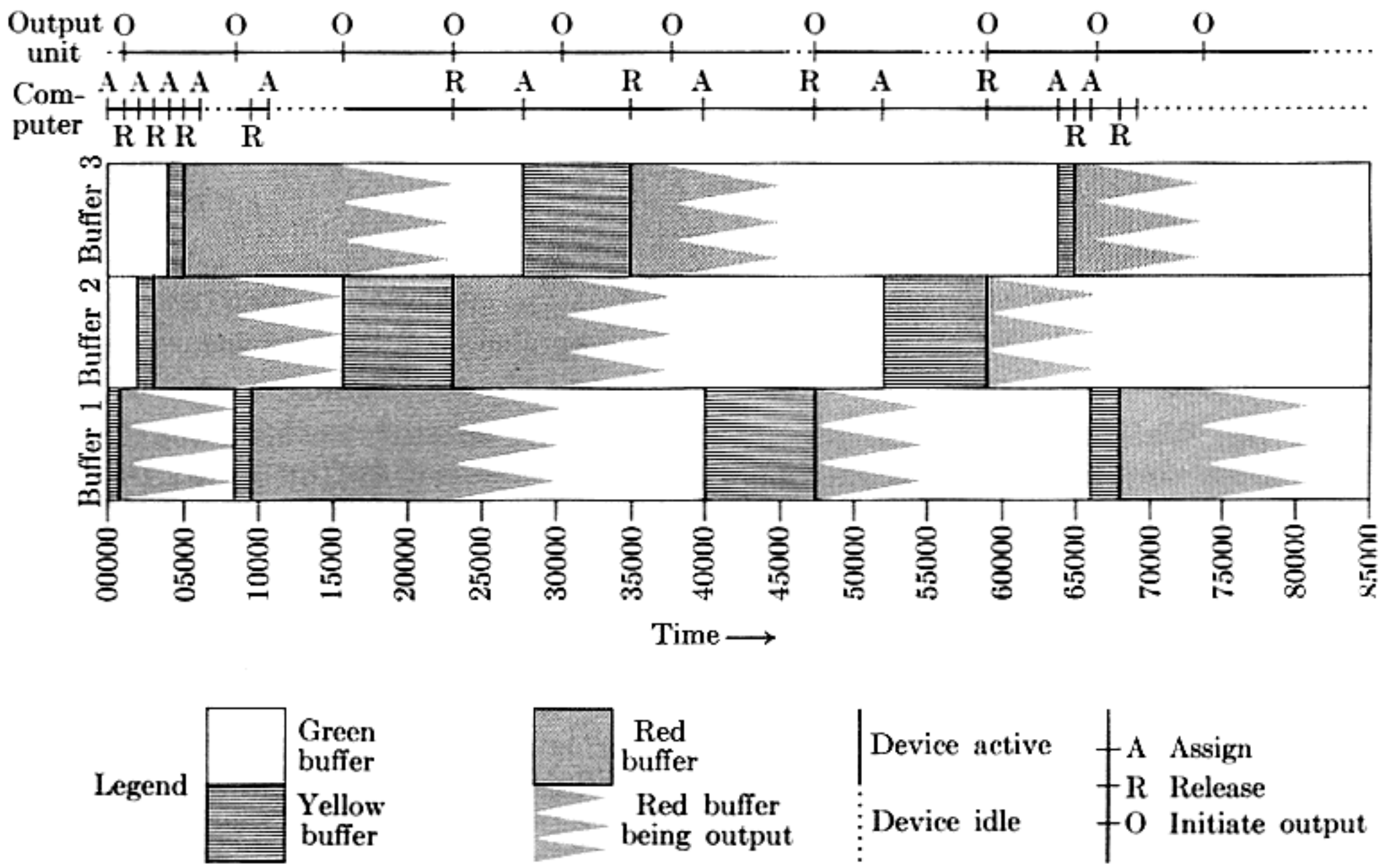


Fig. 27. Output with three buffers (cf. exercise 9).

The following chart specifies the actions shown in Fig. 27 as time passes:

Time	Action	Time	Action
0	ASSIGN(BUF1)	38500	OUT BUF3
1000	RELEASE, OUT BUF1	40000	ASSIGN(BUF1)
2000	ASSIGN(BUF2)	46000	Output stops.
3000	RELEASE	47000	RELEASE, OUT BUF1
4000	ASSIGN(BUF3)	52000	ASSIGN(BUF2)
5000	RELEASE	54500	Output stops.
6000	ASSIGN (wait)	59000	RELEASE, OUT BUF2
8500	BUF1 assigned, OUT BUF2	64000	ASSIGN(BUF3)
9500	RELEASE	65000	RELEASE
10500	ASSIGN (wait)	66000	ASSIGN(BUF1)
16000	BUF2 assigned, OUT BUF3	66500	OUT BUF3
23000	RELEASE	68000	RELEASE
23500	OUT BUF1	69000	Computation stops.
28000	ASSIGN(BUF3)	74000	OUT BUF1
31000	OUT BUF2	81500	Output stops.
35000	RELEASE		

The total time required was therefore 81500u; the computer was idle from 6000–8500, 10500–16000, and 69000–81500, or 20500u altogether; the output unit was idle from 0–1000, 46000–47000, and 54500–59000, or 6500u.

Make a "time-action" chart like the above for the same program, assuming there are only *two* buffers.

10. [*21*] Repeat exercise 9, except with *four* buffers.

11. [*21*] Repeat exercise 9, except with just *one* buffer.

12. [*24*] Suppose that the multiple buffering algorithm in the text is being used for card input, and suppose the input is to terminate as soon as a card with "." in column 80 has been read. Show how the CONTROL coroutine (i.e., Algorithm B and Program B) should be changed so that input is shut off in this way.

13. [*20*] What instructions should be included at the end of the COMPUTE coroutine in the text, if the buffering algorithms are being applied to output, to ensure that all information has been output from the buffers?

▶ **14.** [*20*] What if the computational program does not alternate between ASSIGN and RELEASE, but instead gives the sequence of actions . . . ASSIGN . . . ASSIGN . . . RELEASE . . . RELEASE. What effect does this have on the algorithms described in the text? Is it possibly useful?

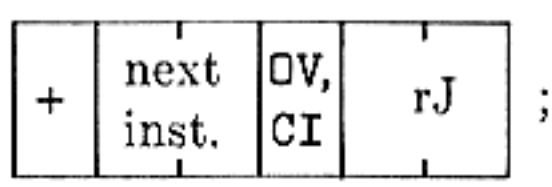
▶ **15.** [*22*] Write a program that copies 100 records from tape unit 0 to tape unit 1, using just three buffers. The program should be as fast as possible.

16. [*29*] Formulate the "green-yellow-red-purple" algorithm suggested by Fig. 26, in the manner of the algorithms for multiple buffering given in the text, using three coroutines (one to control the input device, one for the output device, and the computation coroutine).

17. [*40*] Adapt the multiple-buffer algorithm to pooled buffers; build in methods which keep the process from slowing down, due to too much anticipated input. Try to make the algorithm as elegant as possible. Compare your method to nonpooling methods, applied to real-life problems.

▶ **18.** [*30*] A modification of MIX is planned which introduces "interrupt capability." This would be done as explained below; the exercise is to modify Algorithms and Programs A, R, and B of the text so that they use these interrupt facilities instead of the "JRED" instructions.

The new MIX features include an additional 3999 memory cells, locations —3999 through —0001. The machine has two internal "states," *normal state* and *control state*. In normal state, locations —3999 through —0001 are not admissible memory locations and the MIX computer behaves as usual. When an "interrupt" occurs, due to conditions explained later, locations —0009 through —0001 are set equal to the contents of MIX's registers: rA in —0009; rX in —0008; rI1 through rI6 in —0007 through —0002; and rJ, the overflow toggle, the comparison indicator, and the location of the next instruction all are stored in —0001 as



control state is entered, and the machine jumps to a location depending on the type of interrupt.

Location —0010 acts as a "clock": every 1000*u* of time, the number appearing in this location is decreased by one, and if the result is zero an interrupt to location —0011 occurs.

The new MIX instruction "INT" (C = 5, F = 7) works as follows: (a) In normal state, an interrupt occurs to location —0012. (Thus a programmer may force an interrupt, to communicate with a control routine; the address of INT has no effect, although the control routine may use it for information to distinguish between types of interrupt.) (b) In control state, all MIX registers are loaded from locations —0009 to —0001, the computer goes into normal state, and it resumes execution. The execution time for INT is $2u$ in each case.

An IN, OUT, or IOC instruction given in *control* state will cause an interrupt to occur as soon as the I/O operation is completed. The interrupt goes to location —(0020 + unit number).

No interrupts occur while in control state; any interrupt conditions are "saved" until after the next INT operation, and interrupt will occur after one instruction of the normal state program has been performed.

▶ **19.** [*37*] Some computers do not have the ability to perform input-output simultaneously with computation; the I/O operators cause the computer to wait until transmission is complete. These computers have no equivalent of MIX's JBUS or JRED operators, since the units are always "ready"; and there is no interrupt capability as given in the previous exercise. However, there is sometimes the ability to do I/O operations on two different units at once, by giving an "IN–IN" or "IN–OUT" or "OUT–OUT" instruction that causes *two* operations to occur (and the computer waits until *both* are finished).

Multiple-buffering techniques can be used to advantage in this situation, in order to double up I/O operations as frequently as possible. For example, if an output is to be done, anticipated input could simultaneously be read into a buffer.

Develop algorithms for this situation, assuming that a program uses two input tapes and one output tape. There should be three circles of buffers, with N_1 buffers in the first circle, N_2 in the second, and N_3 in the third. Give algorithms for assigning and releasing on each unit, which are as similar to Algorithms A, R, and B of this section as possible. Test your algorithms using computer simulation.

1.4.5. History and Bibliography

Most of the fundamental techniques described in Section 1.4 have been independently developed by a number of different people, and the exact history of the ideas will probably never be known. An attempt has been made to record here the most important contributions to the history.

Subroutines were the first labor-saving devices invented for programmers. Perhaps they were envisioned already in the 19th century by Charles Babbage in connection with his Analytical Engine (but only to the extent that the idea would apply to the punched-card control of the Jacquard loom, since that served as the control of his machine). A subroutine for computing $\sin x$ was written in August, 1944, for the Mark I calculator under the direction of Howard Aiken. This machine was controlled by programs on external paper tapes, instead of programs stored in the memory. H. H. Goldstine and J. von Neumann gave detailed methods for relocating machine instructions and doing subroutine linkage in 1946; see von Neumann, *Collected Works*, Vol. 5, pp. 215–235. In

England, A. M. Turing independently designed computer hardware to facilitate subroutine linkage. [See *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery* (Cambridge, Mass.: Harvard University, 1949), 83–90.] The use and construction of a very versatile subroutine library is the principal topic of the first computer programming text, *The Preparation of Programs for an Electronic Digital Computer*, by M. V. Wilkes, D. J. Wheeler, and S. Gill, 1st ed. (Reading, Mass.: Addison-Wesley, 1951).

The word "coroutine" was coined by M. E. Conway in 1958, after he had developed the concept, and he first applied it to the construction of an assembly program. Coroutines were independently studied by J. Erdwinn and J. Merner, at about the same time; they wrote a paper entitled "Bilateral linkage," which was not then considered sufficiently interesting to merit publication, and unfortunately no copies of this paper seem to exist today. The first published explanation of the coroutine concept appeared much later in Conway's article "Design of a Separable Transition-Diagram Compiler," *CACM* **6** (1963), 396–408.

The first interpretive routine may be said to be the "Universal Turing Machine," a Turing machine capable of simulating any other Turing machines (see Chapter 11). These are not actual machines, they are theoretical tools used in proving some problems "unsolvable." Interpretive routines in the conventional sense were spontaneously developed slightly before 1950 at several places, chiefly to provide a convenient means of doing floating-point arithmetic. Turing took a part in this development also; interpretive systems for the Pilot ACE computer were written under his direction. The most notable early interpreters were certain routines for the Whirlwind I (by C. W. Adams and others) and for the Illiac (by D. J. Wheeler and others). For references to the state of interpreters in the early fifties, see the article "Interpretative Sub-routines," by J. M. Bennett, D. G. Prinz, and M. L. Woods, *Proc. ACM Nat. Conf.* (1952), 81–87; see also various papers in the *Proceedings of the Symposium on Automatic Programming for Digital Computers* (1954), published by the Office of Naval Research, Washington, D.C.

The most extensively used early interpretive system was probably John Backus's "IBM 701 Speedcoding system" [see *JACM* **1** (1954), 4–6]. This system was slightly modified and skillfully written for the IBM 650 by V. M. Wolontis and others of the Bell Telephone Laboratories; their routine, called the "Bell Interpretive System," was extremely popular. The IPL interpretive systems, designed in 1956 by A. Newell and H. Simon for applications to quite different problems (see Section 2.6), have also seen extensive use as a programming tool. Modern uses of interpreters, as mentioned in the introduction to Section 1.4.3, are often mentioned in passing in the computer literature; see the references listed in that section for articles which discuss interpretive routines in somewhat more detail.

The first tracing routine was developed by Stanley Gill in 1950; see his interesting article in *Proceedings of the Royal Society of London*, series A, **206** (May, 1951), 538–554. The text by Wilkes, Wheeler, and Gill mentioned above

includes listings of several trace routines. Perhaps the most interesting of these is subroutine C-10 by D. J. Wheeler, which includes a provision for suppressing the trace upon entry to a library subroutine, executing the subroutine at full speed, then continuing the trace. Published information about trace routines is quite rare in the general computer literature, primarily because the methods are inherently oriented to a particular machine. The only other reference known to the author is H. V. Meek, "An Experimental Monitoring Routine for the IBM 705," *Proc. Western Joint Computer Conf.* (1956), 68–70, which discusses a trace routine for a machine on which the problem is particularly difficult.

The development of buffering techniques waited until computers became capable of doing simultaneous operations and suitably controlling the interactions. Buffering was first done by computer hardware, in a manner analogous to the code 1.4.4(3), where an internal "buffer area" inaccessible to the programmer plays the role of locations 2000–2099, and where the sequence 1.4.4(3) was performed when an input command was given. For a good survey of the prevailing philosophy towards I/O in 1952, see the Proceedings of the Eastern Joint Computer Conference held in that year.

The real invention of buffering occurred in the minds of computer designers who first realized that they could let the I/O device communicate with core memory at the same time the program is running, and who built in suitable instructions to make efficient programmatic buffering possible. (The author attempted to locate patent numbers of these inventions, but found the search too formidable.) The first published reference to buffering techniques in the sense we have described gives a highly sophisticated approach; see O. Mock and C. J. Swift, "Programmed Input-Output Buffering," *Proc. ACM Nat. Conf.* (1958) paper 19, and *JACM* **6** (1959), 145–151. (The reader is cautioned that these articles contain a good deal of local jargon which may take some time to understand, but neighboring articles in *JACM* **6** will help.) An interrupt system which enabled buffering of input and output was independently developed by E. W. Dijkstra of the Netherlands in 1957 and 1958. His doctoral thesis, "Communication with an Automatic Computer" (1958, now out of print) mentions buffering techniques, which in this case involved very long circles of buffers since the routines were primarily concerned with paper tape and typewriter I/O; each buffer contained either a single character or a single number. The paper "Input-Output Buffering and FORTRAN," by David E. Ferguson, *JACM* **7** (1960), 1–9, describes buffer circles and gives a detailed description of simple buffering with many units at once.

CHAPTER TWO

INFORMATION STRUCTURES

I think that I shall never see
A poem lovely as a tree.

—JOYCE KILMER (1913)

Yea, from the table of my memory
I'll wipe away all trivial fond records.

— Hamlet (Act I, Sc. 5, Line 98)

2.1. INTRODUCTION

Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important *structural relationships* between the data elements.

In its simplest form, a table might be a linear list of elements, when its relevant structural properties include the answers to such questions as: which element is first in the list? which is last? which elements precede and follow a given one? There is a lot to be said about structure even in this apparently simple case (see Section 2.2).

In more complicated situations, the table might be a two-dimensional array (i.e., a matrix or grid, having both a row and a column structure), or it might be an n-dimensional array for higher values of n; it might be a tree structure, representing hierarchical or branching relationships; or it might be a complex multilinked structure with a great many interconnections, such as we may find in a human brain.

In order to use a computer properly, it is important to acquire a good understanding of the structural relationships present within data, and of the techniques for representing and manipulating such structure within a computer.

The present chapter summarizes the most important facts about information structures: the static and dynamic properties of different kinds of structure; means for storage allocation and representation of structured data; and efficient algorithms for creating, altering, accessing, and destroying structural information. In the course of this study, we will also work out several important examples which illustrate the application of these methods to a wide variety of problems. The examples include topological sorting, polynomial arithmetic,

discrete system simulation, operations on sparse matrices, algebraic formula manipulation, and applications to the writing of compilers and operating systems. Our concern will be almost entirely with structure as represented *inside* a computer; the conversion from external to internal representations is the subject of Chapters 9 and 10.

Much of the material we will discuss is often called "List processing," since a number of programming systems (e.g., IPL-V, LISP, and SLIP) have been designed to facilitate working with certain general kinds of structures called *Lists*. (When the word "list" is capitalized in this chapter, it is being used in a technical sense to denote a particular type of structure that is studied in detail in Section 2.3.5.) Although List-processing systems are useful in a large number of situations, they impose constraints on the programmer that are often unnecessary; it is usually better to use the methods of this chapter directly in one's own programs, tailoring the data format and the processing algorithms to the particular application. Too many people unfortunately still feel that List-processing techniques are quite complicated (so that it is necessary to use someone else's carefully written interpretive system or set of subroutines), and that List-processing must be done only in a certain fixed way. We will see that there is nothing magic, mysterious, or difficult about the methods for dealing with complex structures; these techniques are an important part of every programmer's repertoire, and he can use them easily whether he is writing a program in assembly language or in a compiler language like FORTRAN or ALGOL.

We will illustrate methods of dealing with information structures in terms of the MIX computer. A reader who does not care to look through detailed MIX programs should at least study the ways in which structural information is represented in MIX's memory.

It is important to define at this point several terms and notations which we will be using frequently from now on. The information in a table consists of a set of *nodes* (called "records," "entities," or "beads" by some authors); we will occasionally say "item" or "element" instead of "node." Each node consists of one or more consecutive words of the computer memory, divided into named parts called *fields*. In the simplest case, a node is just one word of memory, and it has just one field comprising that whole word. As a more interesting example, suppose the elements of our table are intended to represent playing cards; we might have two-word nodes broken into five fields, TAG, SUIT, RANK, NEXT, and TITLE:

+	TAG	SUIT	RANK	NEXT
+	TITLE			

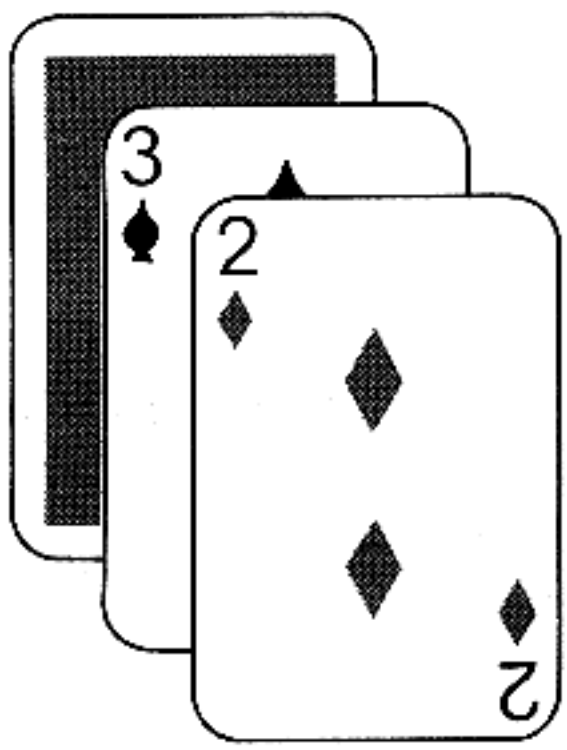
(1)

(This format reflects the contents of two MIX words. Recall that a MIX word consists of five bytes plus a sign; see Section 1.3.1. In this example we assume the signs are + in each word.) The *address* of a node, also called a *link*, *pointer*, or *reference* to that node, is the memory location of its first word. The address

is often taken relative to some "base" location, but in this chapter for simplicity we will take the address to be an absolute memory location.

The contents of any field within a node may represent numbers, alphabetic characters, links, or anything else the programmer may desire. In connection with the example above, let us suppose we wish to represent a pile of cards that might appear in a game of solitaire: TAG = 1 means the card is face down, TAG = 0 means it is face up; SUIT = 1, 2, 3, or 4 for clubs, diamonds, hearts, or spades, respectively; RANK = 1, 2, . . . , 13 for ace, deuce, . . . , king; NEXT is a *link* to the card *below* this one in the pile; and TITLE is the five-character alphabetic name of this card, for use in printouts. A typical pile might look like this:

Actual cards



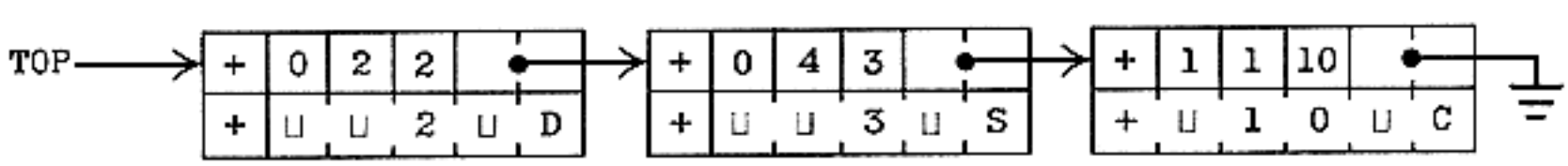
Computer representation

100:	+	1	1	10	Λ	
101:	+	⊔	1	0	⊔	C
386:	+	0	4	3	100	
387:	+	⊔	⊔	3	⊔	S
242:	+	0	2	2	386	
243:	+	⊔	⊔	2	⊔	D

(2)

The memory locations in the computer representation are shown here as 100, 386, and 242; these could have been any other numbers as far as this example is concerned, since each card links to the next. Note the special link "Λ" in the node 100; *we use the Greek letter lambda to denote the null link*, i.e., the link to no node. The null link Λ appears in node 100 since the 10 of clubs is the bottom card of the pile. Within the machine, Λ is represented by some easily recognizable value which cannot be the address of a node. We will generally assume no node appears in location 0, and, consequently, Λ will almost always be represented as the link value 0 in MIX programs.

The introduction of links to other elements of data is an extremely important idea in computer programming; this is the key to the representation of complex structures. When displaying computer representations of nodes it is usually convenient to represent links by arrows, so that our above example would appear thus:



(3)

The actual locations 242, 386, and 100 (which are irrelevant anyway) no longer

appear in the representation (3). A null link can be shown as "grounded" in electrical circuit notation. We have added "TOP" in (3); this stands for a *link variable*, often called a pointer variable, i.e., a variable within the computer program whose value is a link. All references to nodes in a program are made directly through link variables (or link constants), or indirectly through link fields in other nodes.

Now we come to the most important part of the notation, the means of referring to fields within nodes. This is done simply by giving the name of the field followed by a link to the desired node in parentheses; for example in (1), (2), (3) we have

$$\texttt{TITLE(TOP)} = \text{"}\sqcup\sqcup\texttt{2}\sqcup\texttt{D"}; \qquad \texttt{SUIT(TOP)} = 2; \qquad \texttt{RANK(100)} = 10; \qquad \texttt{RANK(NEXT(TOP))} = 3. \tag{4}$$

The reader should study these examples carefully, since such field notations will be used in many algorithms of this chapter and the following chapters. To make the ideas clearer, we will now state a simple algorithm for placing a new card on top of the pile, assuming NEWCARD is a link variable whose value is a link to the new card:

A1. Set NEXT(NEWCARD) ← TOP. (This sets the appropriate link in the new card node.)

A2. Set TOP ← NEWCARD. (This keeps TOP pointing to the top of the pile.)

A3. Set TAG(TOP) ← 0. (This marks the card as "face up.") ▮

Another example is the following algorithm, which counts the number of cards currently in the pile:

B1. Set N ← 0, X ← TOP. (N is an integer variable, X is a link variable.)

B2. If X = Λ, stop; N is the number of cards in the pile.

B3. Set N ← N + 1, X ← NEXT(X), and go back to step B2. ▮

Note that we use symbolic names for two quite different things in these algorithms: as names of *variables* (TOP, NEWCARD, N, X) and as names of *fields* (TAG, NEXT). These quantities must not be confused. If F is a field name and L ≠ Λ is a link, then F(L) is a variable; but F itself is not a variable—it does not possess a value unless it is qualified by a nonnull link.

Two further notations are used, to convert between addresses and the values stored there:

a) CONTENTS always denotes a full-word field of a one-word node; hence CONTENTS(1000) denotes the value stored in memory location 1000, i.e., it is a variable having this value. If V is a link variable, CONTENTS(V) is the value pointed to by V (not the value V itself).

b) If V is a variable, LOC(V) denotes the memory location of V. Consequently, if V is a variable whose value is stored in a full word of memory, we have CONTENTS(LOC(V)) = V.

It is easy to transform this notation into MIXAL assembly language code. The values of link variables are put into index registers, and the partial-field capability of MIX is used to refer to the desired field. For example, Algorithm A above could be written thus:

```
NEXT  EQU  4:5          Definition of the NEXT
TAG   EQU  1:1              and TAG fields for the assembler
      LD1  NEWCARD      A1. rI1←NEWCARD.
      LDA  TOP              rA←TOP.                          (5)
      STA  0,1(NEXT)        NEXT(rI1)←rA.
      ST1  TOP          A2. TOP←rI1.
      STZ  0,1(TAG)     A3. TAG(rI1)←0. ▮
```

The ease and efficiency with which these operations can be carried out in a computer is the primary reason for the importance of the "linked memory" concept.

Sometimes we have a single variable which denotes a whole node (i.e., a set of fields instead of just one field). Thus we might write

$$\texttt{CARD} \leftarrow \texttt{NODE(TOP)}, \tag{6}$$

where NODE is a field specification just like CONTENTS, except that it refers to an entire node, and where CARD is a variable which assumes values like those in (1). If there are c words in a node, the notation (6) is an abbreviation for the c assignments

$$\texttt{CONTENTS(LOC(CARD)} + j) \leftarrow \texttt{CONTENTS(TOP} + j), \qquad 0 \le j < c. \tag{7}$$

EXERCISES

1. [*04*] In (3), what is the value of
   a) `TAG(NEXT(TOP))`; b) `NEXT(NEXT(NEXT(TOP)))`?

▶ 2. [*10*] The text points out that in many cases `CONTENTS(LOC(V)) = V`. Under what conditions do we have `LOC(CONTENTS(V)) = V`?

3. [*11*] Give an algorithm which essentially undoes the effect of Algorithm A, i.e., it removes the top card of the pile (if the pile is not empty) and sets `NEWCARD` to the address of this card.

4. [*18*] Give an algorithm analogous to Algorithm A, except that it puts the new card *face down* at the *bottom* of the pile. (The pile may be empty.)

▶ 5. [*21*] Give an algorithm which essentially undoes the effect of exercise 4, i.e., assuming that the pile is not empty and its bottom card is face down, it removes this bottom card and makes `NEWCARD` link to it. (This algorithm is sometimes called "cheating" in solitaire games.)

6. [*06*] In the playing card example, suppose that `CARD` is the name of a variable whose value is an entire node. The operation `CARD ← NODE(TOP)` sets the fields of

CARD respectively equal to those of the top of the pile. After this operation, which of the following notations stands for the suit of the top card? (a) SUIT(CARD); (b) SUIT(LOC(CARD)); (c) SUIT(CONTENTS(CARD)); (d) SUIT(TOP)?

7. [*04*] In the text's example MIX program, (5), the link variable TOP is stored in the MIX computer word whose assembly language name is TOP. Assuming the field structure (1), which of the following sequences of code brings the quantity NEXT(TOP) into register A? Explain why the other sequence is incorrect.

```
a) LDA   TOP(4:5)

b) LD1   TOP
   LDA   0,1(4:5)
```

▶ **8.** [*18*] Write a MIX program corresponding to Algorithm B.

9. [*23*] Write a MIX program which prints out the alphabetic names of the current contents of the card pile, starting at the top card, with one card per line, and with parentheses around cards that are face down.

2.2. LINEAR LISTS

2.2.1. Stacks, Queues, and Deques

Usually there is much more structural information present in the data than we actually want to represent directly in a computer. In each "playing card" node of the preceding section, for example, we have a NEXT field to specify what card is beneath it in the pile, but there is no direct way to find what card, if any, is *above* a given card, or to find which pile a given card is in. Of course, there is much information possessed by any *real* deck of playing cards which has been totally suppressed from the computer representation: the details of the design on the back of the cards, the relation of the cards to other objects in the room where the game is being played, the molecules which compose the cards, etc. It is conceivable that such structural information would be relevant in certain computer applications, but obviously we never want to store *all* of the structure present in every situation. Indeed, for most card-playing situations we would not need all of the facts retained in our earlier example; thus the TAG field, which tells whether a card is face up or face down, will often be unnecessary.

It is therefore clear that we must decide in each case how much structure to represent in our tables, and how accessible to make each piece of information. To make this decision, we need to know what operations are to be performed on the data. For each problem considered in this chapter, *we therefore consider not only the data structure but also the class of operations to be done on the data;* the design of computer representations depends on the desired function of the data as well as on its intrinsic properties. Such an emphasis on "function" as well as "form" is basic to design problems in general.

In order to illustrate this point further, let us consider a simple example which arises in computer hardware design: a computer memory is often classified as a "random access memory," i.e., MIX's main memory; or as a "read only memory," i.e., one which is to contain essentially constant information; or a "secondary bulk memory," like MIX's disk units, which cannot be accessed at high speed although large quantities of information can be stored; or an "associative memory," more properly called a "content-addressed memory," i.e., one for which information is addressed by values stored with it rather than by its location; and so on. Note that the intended function of each kind of memory is so important that it enters into the name of the particular memory type; all of these devices are "memory" units, but the purposes to which they are put profoundly influence their design and their cost.

A *linear list* is a set of $n \geq 0$ nodes X[1], X[2], . . . , X[n] whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that, if $n > 0$, X[1] is the first node; when $1 < k < n$, the kth node X[k] is preceded by X[$k - 1$] and followed by X[$k + 1$]; and X[n] is the last node.

The operations we might want to perform on linear lists include, for example, the following.

i) Gain access to the kth node of the list to examine and/or change the contents of its fields.
ii) Insert a new node just before the kth node.
iii) Delete the kth node.
iv) Combine two or more linear lists into a single list.
v) Split a linear list into two or more lists.
vi) Make a copy of a linear list.
vii) Determine the number of nodes in a list.
viii) Sort the nodes of the list into ascending order based on certain fields of the nodes.
ix) Search the list for the occurrence of a node with a particular value in some field.

In operations (i), (ii), and (iii) the special cases $k = 1$ and $k = n$ are of principal importance since the first and last items of a linear list may be easier to get at than a general element is. We will not discuss operations (viii) and (ix) in this chapter, since these topics are the subjects of Chapters 5 and 6, respectively.

A computer application rarely calls for all nine of the above operations in their full generality, so we find there are many ways to represent linear lists depending on the class of operations which are to be done most frequently. It appears to be impossible to design a single representation method for linear lists in which all of these operations are efficient; for example, the ability to gain access to the kth node of a long list for random k is comparatively hard to do if at the same time we are inserting and deleting items in the middle of the list. Therefore we distinguish between types of linear lists depending on the principal operations to be performed, just as we have noted that computer memories are distinguished by their intended applications.

Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names:

A *stack* is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list.

A *queue* is a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end.

A *deque* ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list.

A deque is therefore more general than a stack or a queue; it has some properties in common with a deck of cards, and it is pronounced the same way. We also distinguish *output-restricted* or *input-restricted* deques, in which deletions or insertions, respectively, are allowed to take place at only one end.

In some disciplines the word "queue" has been used in a much broader sense to describe any kind of list that is subject to insertions and deletions; the special cases identified above are then called various "queuing disciplines." Only the

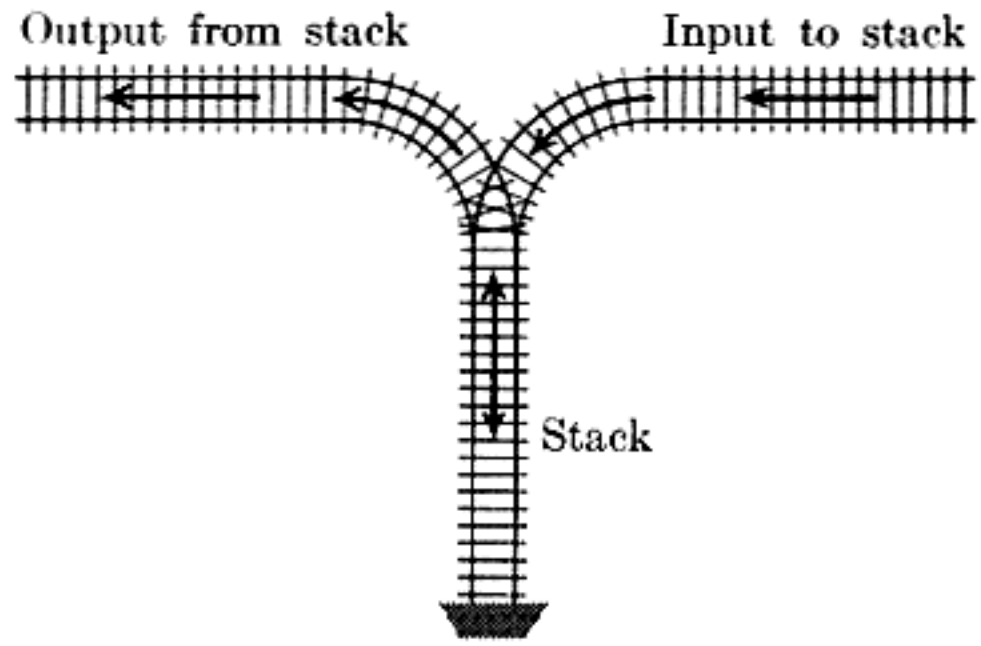


Fig. 1. A stack represented as a railway switching network.

restricted use of the term "queue" is intended in this book, however, by analogy with orderly queues of people waiting in line for service.

Sometimes it helps to understand the mechanism of a stack in terms of an analogy from the switching of railroad cars, as suggested by E. W. Dijkstra (see Fig. 1). A corresponding picture for deques is shown in Fig. 2.

With a stack we always remove the "youngest" item currently in the list, i.e., the one which has been inserted more recently than any other. With a queue just the opposite is true: the "oldest" item is always removed; the nodes leave the list in the same order as they entered it.

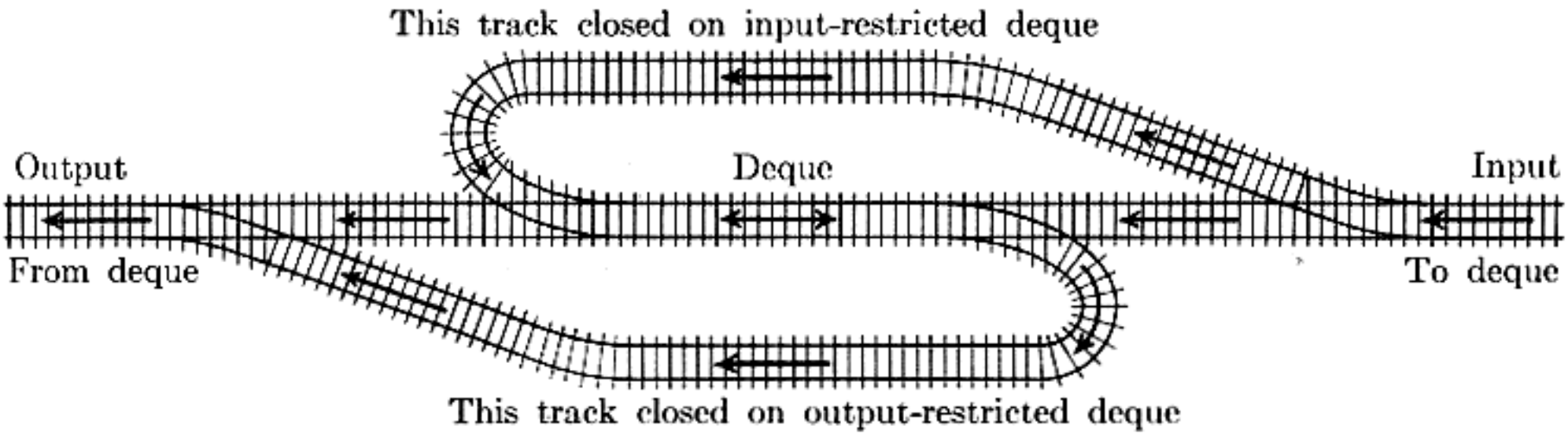


Fig. 2. A deque represented as a railway switching network.

Many people who realized the importance of stacks and queues independently have given other names to these structures: stacks have been called push-down lists, reversion storages, cellars, nesting stores, piles, last-in-first-out ("LIFO") lists, and even yo-yo lists! Queues are sometimes called circular stores or first-in-first-out ("FIFO") lists. The terms LIFO and FIFO have been used for many years by accountants, as names of methods for pricing inventories. Still another term, "shelf," has been applied to output-restricted deques, and input-restricted deques have been called "scrolls" or "rolls." This multiplicity of other names is interesting in itself since it is evidence for the importance of the concepts. The words stack and queue are gradually becoming standard terminology; and of all the other words listed above, only "push-down list" is still reasonably common, particularly in connection with automata theory.

Stacks arise quite frequently in practice. One simple example is the situation where we go through a set of data and keep a list of exceptional conditions or

things to do later; when the original set is processed, we come back to this list to do the subsequent processing, removing its entries until it becomes empty. (For example, see the "saddle point" problem, exercise 1.3.2–10.) Either a stack or a queue serves this purpose, and a stack is generally more convenient. We all have "stacks" in our minds when we are solving problems: One problem leads to another and this leads to another; we stack up the problems and subproblems and remove them as they are solved. Similarly, the process of entering and leaving subroutines during the execution of a computer program has a stack-like behavior. Stacks are particularly useful for the processing of languages with a nested structure, like programming languages, arithmetic expressions, and the literary German "Schachtelsätze." In general, stacks most frequently occur in connection with explicitly or implicitly recursive algorithms, and we will discuss this connection thoroughly in Chapter 8.

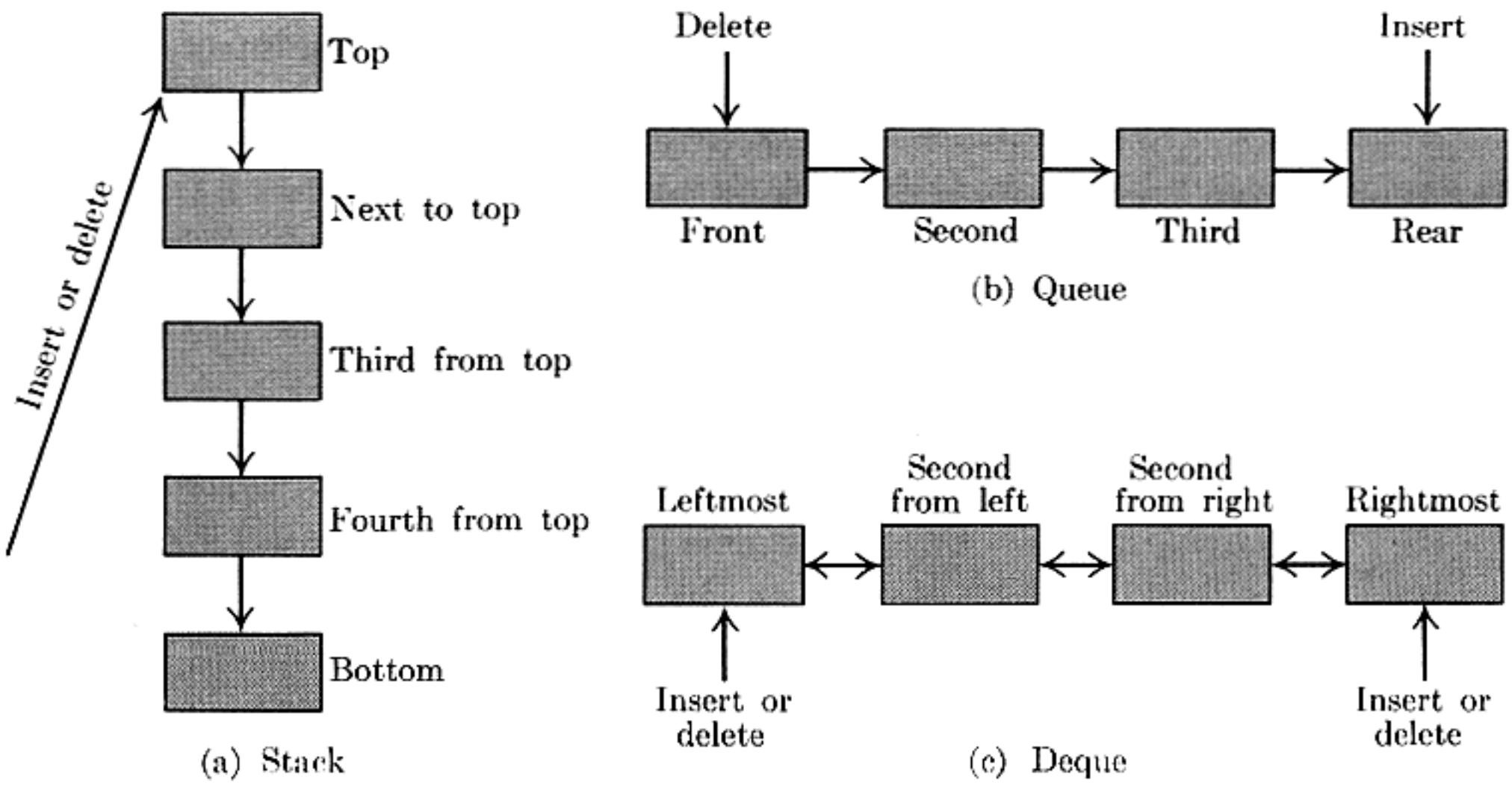


Fig. 3. Special linear lists.

Special terminology is generally used in algorithms referring to these structures: We put an item onto the *top* of a stack, or take off the top item (see Fig. 3a). The *bottom* of the stack is the least accessible item, and it will not be removed until all other items have been deleted. (People often say they *push down* an item onto a stack, and *pop up* the stack when the top item is deleted. This terminology comes from an analogy with the stack of plates often found in cafeterias, or with stacks of cards in some punched-card devices. The brevity of the words "push" and "pop" has its advantages, but these terms falsely imply a motion of the whole list within computer memory. Nothing is physically pushed down; items are added onto the top, as in haystacks or stacks of boxes.) With queues, we speak of the *front* and the *rear* of the queue; things enter at the rear and are removed when they ultimately reach the front position (see Fig. 3b). When referring to deques, we speak of the *left* and *right* ends (Fig. 3c). The concepts of top, bottom, front, and rear are sometimes applied to deques being

used as stacks or queues, with no standard conventions as to whether top, front, and rear are to appear at the left or the right.

Thus we find it easy to use a rich variety of descriptive words from English in our algorithms: "up-down" terminology for stacks, "left-right" terminology for deques, and "waiting in line" terminology for queues.

A little bit of additional notation has proved to be convenient for dealing with stacks and queues: we write

$$A \Leftarrow x \tag{1}$$

(when A is a stack) to mean that the value x is inserted on top of stack A, or (when A is a queue) to mean that x is inserted at the rear of the queue. Similarly, the notation

$$x \Leftarrow A \tag{2}$$

is used to mean that the variable x is set equal to the value at the top of stack A or at the front of queue A, and this value is deleted from A. Notation (2) is meaningless when A is empty, i.e., when A contains no values.

When A is a nonempty stack, we may write

$$\text{top}\ (A) \tag{3}$$

to denote its top element.

EXERCISES

1. [*06*] An input-restricted deque is a linear list in which items may be inserted at one end but removed from either end; clearly an input-restricted deque can operate either as a stack or as a queue, if we consistently remove all items from one of the two ends. Can an output-restricted deque also be operated either as a stack or a queue?

▶ **2.** [*15*] Imagine four railroad cars positioned on the input side of the track in Fig. 1, numbered 1, 2, 3, and 4, respectively. Suppose we perform the following sequence of operations (which is compatible with the direction of the arrows in the diagram and does not require cars to "jump over" other cars): (a) move car 1 into the stack; (b) move car 2 into the stack; (c) move car 2 into the output; (d) move car 3 into the stack; (e) move car 4 into the stack; (f) move car 4 into the output; (g) move car 3 into the output; (h) move car 1 into the output.

As a result of these operations the original order of the cars, 1234, has been changed into 2431. *It is the purpose of this exercise and the following exercises to examine what permutations are obtainable in such a manner from stacks, queues, or deques.*

If there are six railroad cars numbered 123456, can they be permuted into the order 325641? Can they be permuted into the order 154623? (In case it is possible, show how to do it.)

3. [*25*] The operations (a) through (h) in the previous exercise can be much more concisely described by the code SSXSSXXX, where S stands for "move a car from the input into the stack," and X stands for "move a car from the stack into the output."

Some sequences of S's and X's specify meaningless operations, since there may be no cars available on the specified track; for example, the sequence SXXSSXXS cannot be carried out.

Let us call a sequence of S's and X's *admissible* if it contains n S's and n X's, and if it specifies no operations that cannot be performed. Formulate a rule by which it is easy to distinguish between admissible and inadmissible sequences; show furthermore that no two different admissible sequences give the same output permutation.

4. [*M34*] Find a simple formula for a_n, the number of permutations on n elements that can be obtained with a stack like that in exercise 2.

▶ **5.** [*M28*] Show that it is possible to obtain the permutation $p_1 p_2 \ldots p_n$ from $1\ 2 \ldots n$ using a stack if and only if there are no indices $i < j < k$ such that $p_j < p_k < p_i$.

6. [*00*] Consider the problem of exercise 2, with a queue substituted for a stack. What permutations of $1\ 2 \ldots n$ can be obtained with the use of a queue?

▶ **7.** [*25*] Consider the problem of exercise 2, with a deque substituted for a stack. (a) Find a permutation of 1 2 3 4 which can be obtained with an input-restricted deque, but which cannot be obtained with an output-restricted deque. (b) Find a permutation of 1 2 3 4 which can be obtained with an output-restricted deque but not with an input-restricted deque. [As a consequence of (a) and (b), there is a definite difference between input-restricted and output-restricted deques.] (c) Find a permutation of 1 2 3 4 which cannot be obtained with either an input-restricted or an output-restricted deque.

8. [*22*] Are there any permutations of $1\ 2 \ldots n$ which cannot be obtained with the use of a deque that is neither input- nor output-restricted?

9. [*M20*] Let b_n be the number of permutations on n elements obtainable by the use of an input-restricted deque. (Note that $b_4 = 22$, as shown in exercise 7.) Show that b_n is also the number of permutations on n elements obtainable with an *output*-restricted deque.

10. [*M25*] (See exercise 3.) Let S, Q, and X denote respectively the operations of inserting an element at the left, inserting an element at the right, and emitting an element from the left, of an output-restricted deque. For example, the sequence QQXSXSXX will transform the input sequence 1 2 3 4 into 1 3 4 2. The sequence SXQSXSXX gives the same transformation.

Find a way to define the concept of an *admissible* sequence of the symbols S, Q, and X in such a way that (a) each admissible sequence performs a meaningful sequence of operations that defines a permutation of n elements; and that (b) each permutation of n elements that is attainable with an output-restricted deque corresponds to precisely one admissible sequence.

▶ **11.** [*M40*] As a consequence of exercises 9 and 10, the number b_n is the number of admissible sequences of length $2n$. Find a "closed form" for the generating function $\sum_{n\geq 0} b_n z^n$.

12. [*HM34*] Compute the asymptotic values of the quantities a_n and b_n in exercises 4 and 11.

13. [*M49*] How many permutations of n elements are obtainable with the use of a general deque?

2.2.2. Sequential Allocation

The simplest and most natural way to keep a linear list inside a computer is to put the list items in sequential locations, one node after the other. We thus will have

$$\texttt{LOC(X[}j+1\texttt{])} = \texttt{LOC(X[}j\texttt{])} + c,$$

where c is the number of words per node. (Usually $c = 1$. When $c > 1$, it is sometimes more convenient to split a single list into c "parallel" lists, so that the kth word of node X[j] is stored a fixed distance from the location of the first word of X[j]. We will continually assume, however, that adjacent groups of c words form a single node.) In general,

$$\texttt{LOC(X[}j\texttt{])} = \texttt{L}_0 + cj, \tag{1}$$

where $\texttt{L}_0$ is a constant called the *base address*, the location of an artificially assumed node X[0].

This technique for representing a linear list is so obvious and well-known that there seems to be no need to dwell on it at any length. But we will be seeing many other "more sophisticated" methods of representation later on in this chapter, and it is a good idea to examine the simple case first to see just how far we can go with it. It is important to understand the limitations as well as the power of the use of sequential allocation.

Sequential allocation is quite convenient for dealing with a *stack*. We simply have a variable T called the *stack pointer*. When the stack is empty, we let T = 0. To place a new element Y on top of the stack, we set

$$\texttt{T} \leftarrow \texttt{T} + 1; \qquad \texttt{X[T]} \leftarrow \texttt{Y}. \tag{2}$$

And when the stack is not empty, we can set Y equal to the top node and delete that node by reversing the actions of (2):

$$\texttt{Y} \leftarrow \texttt{X[T]}; \qquad \texttt{T} \leftarrow \texttt{T} - 1. \tag{3}$$

(Inside a computer it is usually most efficient to maintain the value cT instead of T, because of (1). Such modifications are easily made, so we will continue our discussion as though $c = 1$.)

The representation of a *queue* or a more general *deque* is a little trickier. An obvious solution is to keep two pointers, say F and R (for the front and rear of the queue), with F = R = 0 when the queue is empty. Then inserting an element at the rear of the queue would be

$$\texttt{R} \leftarrow \texttt{R} + 1; \qquad \texttt{X[R]} \leftarrow \texttt{Y}; \tag{4}$$

removing the front node (F points just below the front) would be

$$\texttt{F} \leftarrow \texttt{F} + 1; \qquad \texttt{Y} \leftarrow \texttt{X[F]}; \qquad \text{if} \quad \texttt{F} = \texttt{R}, \quad \text{then} \quad \text{set} \quad \texttt{F} \leftarrow \texttt{R} \leftarrow 0. \tag{5}$$

But note what can happen: If R always stays ahead of F (so there is always at least one node in the queue) the table entries used are X[1], X[2], . . . , X[1000], . . . , ad infinitum, and this is terribly wasteful of storage space. The simple method (4), (5) is therefore to be used only in the situation when F is known to catch up to R quite regularly (for example, if all deletions come in spurts, which empty the queue).

To circumvent the problem of the queue overrunning memory, we can set aside M nodes X[1], . . . , X[M] arranged implicitly in a circle with X[1] following X[M]. Then the above processes (4), (5) become (for R = F = M initially)

$$\text{if } R = M \text{ then } R \leftarrow 1, \text{ otherwise } R \leftarrow R + 1;\ X[R] \leftarrow Y. \tag{6}$$

$$\text{if } F = M \text{ then } F \leftarrow 1, \text{ otherwise } F \leftarrow F + 1;\ Y \leftarrow X[F]. \tag{7}$$

This circular queuing action is much like that which we have already seen in the discussion of input-output buffering (Section 1.4.4).

The above discussion has been very unrealistic in that we have tacitly assumed nothing could go wrong. When we deleted a node from a stack or queue, we assumed that there was at least one node present. When we inserted a node onto a stack or queue, we assumed there was room for it in memory. But clearly the method (6), (7) allows at most M nodes in the entire queue, and methods (2), (3), (4), (5) allow T and R to reach only a certain maximum amount within any given computer program. The following specifications show how the above actions must be rewritten for the common case where we do not assume that these restrictions are automatically satisfied:

$$X \Leftarrow Y \text{ (insert into stack):} \quad T \leftarrow T + 1;\ \text{if } T > M, \text{ then OVERFLOW};\ X[T] \leftarrow Y. \tag{2a}$$

$$Y \Leftarrow X \text{ (delete from stack):} \quad \text{if } T = 0, \text{ then UNDERFLOW};\ Y \leftarrow X[T];\ T \leftarrow T - 1. \tag{3a}$$

$$X \Leftarrow Y \text{ (insert into queue):} \quad \begin{cases} \text{if } R = M, \text{ then } R \leftarrow 1, \\ \qquad \text{otherwise } R \leftarrow R + 1; \\ \text{if } R = F, \text{ then OVERFLOW}; X[R] \leftarrow Y. \end{cases} \tag{6a}$$

$$Y \Leftarrow X \text{ (delete from queue):} \quad \begin{cases} \text{if } R = F, \text{ then UNDERFLOW}; \\ \text{if } F = M, \text{ then } F \leftarrow 1, \\ \qquad \text{otherwise } F \leftarrow F + 1;\ Y \leftarrow X[F]. \end{cases} \tag{7a}$$

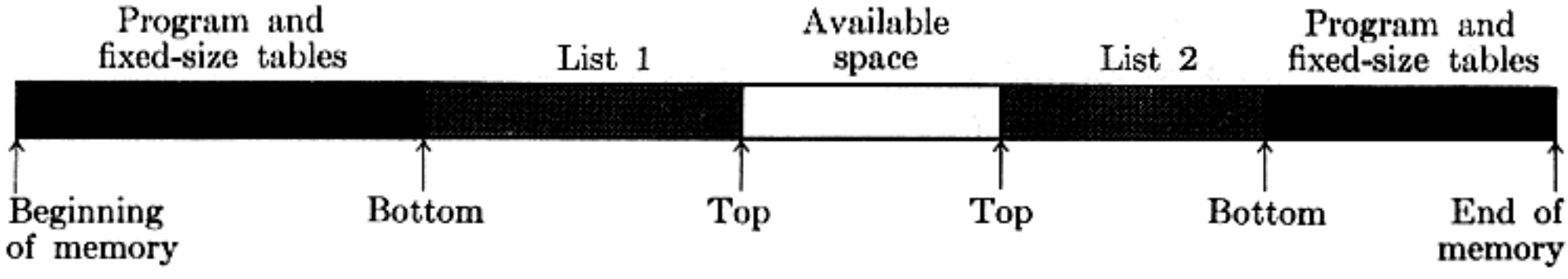
Here we assume that X[1], . . . , X[M] is the total amount of space allowed for the list; OVERFLOW and UNDERFLOW mean an excess or deficiency of items.

The next question is, "What do we do when UNDERFLOW or OVERFLOW occurs?" In the case of UNDERFLOW, we have tried to remove a nonexistent item; this is usually a meaningful condition—not an error situation—which can be used to govern the flow of a program, e.g., we might want to delete items repeatedly

until UNDERFLOW occurs. An OVERFLOW situation, however, is usually an error; it means the table is full already, yet there is still more information that ought to be put in. The usual policy in case of OVERFLOW is to report reluctantly that the program cannot go on because its storage capacity has been exceeded, and the program terminates.

Of course we would hate to give up in an OVERFLOW situation when only one list has gotten too large, while other lists of the same program may very well have plenty of room remaining. In the above discussion we were primarily thinking of a program with only one list. However, we frequently encounter programs which involve several stacks, each of which has a dynamically varying size. In such a situation we would hate to impose a maximum size on each stack, for usually the size is somewhat unpredictable; and even if a maximum size has been determined for each stack, we will rarely find *all* stacks simultaneously filling their maximum capacity.

When there are just two variable size lists, they can coexist together very nicely if we let the lists grow toward each other:



Here list 1 expands to the right, and list 2 (stored in reverse order) expands to the left. OVERFLOW will not occur unless the total size of both lists exhausts all memory space. The lists may independently expand and contract so that the effective maximum size of each one could be significantly more than half of the available space. The above layout of memory space is used very frequently.

The reader may easily convince himself, however, that *there is no way* to store three or more variable-size sequential lists in memory so that (a) OVERFLOW will occur only when the total size of all lists exceeds the total space, and (b) each list has a fixed location for its "bottom" element. When there are, say, ten or more variable size lists—and this is not unusual—the storage allocation problem becomes very significant. If we wish to satisfy condition (a), we must give up condition (b); that is, we must allow the "bottom" elements of the lists to change their positions. This means the location L_0 of Eq. 1 is *not constant* any longer; no reference to the table may be made to an absolute memory address, all references must be relative to the base address L_0. In the case of MIX, the coding to bring a one-word node into register A is changed

```
                                  LD1  I
from   LD1  I          to, e.g.,  LDA  BASE(0:2)          (8)
       LDA  L0, 1                 STA  *+1(0:2)
                                  LDA  *,1
```

where BASE contains

L_0	0	0	0

.

This relative addressing is evidently slower to do than when the base was fixed, although we find it would be only slightly slower if MIX had an "indirect addressing" feature (see exercise 3).

An important special case occurs when each of the variable size lists is a stack. Then, since only the top element of each stack is relevant at any time, we can proceed almost as efficiently as before. Suppose that we have n stacks; the insertion and deletion algorithms above become the following, if BASE[i] and TOP[i] are link variables for the ith stack:

Insertion: TOP[i] $\leftarrow$ TOP[i] $+ c$; if TOP[i] $>$ BASE[$i+1$], then OVERFLOW; otherwise set NODE(TOP[i]) $\leftarrow$ Y. (9)

Deletion: if TOP[i] $=$ BASE[i], then UNDERFLOW; otherwise set Y $\leftarrow$ NODE(TOP[i]), TOP[i] $\leftarrow$ TOP[i] $- c$. (10)

Here BASE[$i+1$] is the beginning location of the $(i+1)$st stack. The condition TOP[i] $=$ BASE[i] means stack i is empty.

In the above situation, OVERFLOW is no longer such a crisis as it was before; we can "repack memory," making room for the table that overflowed by taking some away from tables that aren't yet filled. A number of possible ways to do this suggest themselves, and since these repacking algorithms are very important in connection with sequential allocation of linear lists, we will now consider this problem in detail. We will start by giving the simplest of these methods, and will then consider some of the alternatives.

Assume that there are n stacks, and the values BASE[i] and TOP[i] are to be manipulated as in (9), (10). These stacks are all to share the common memory area consisting of all locations L with $L_0 \le L < L_\infty$. (Here L_0 and L_∞ are constants which specify the total number of words available for use; $L_\infty - L_0$ is a multiple of c.) We might start out with all stacks empty, and BASE[i] $=$ TOP[i] $= L_0 - c$, for all i. We also set BASE[$n+1$] $\equiv L_\infty - c$ so that (9) will work properly for $i = n$. Now whenever a particular stack, except stack n, gets more items in it than it ever had before, OVERFLOW will occur.

When stack i overflows, there are three possibilities:

a) We find the smallest k for which $i < k \le n$ and TOP[k] $<$ BASE[$k+1$], if any such k exist. Now move things *up* one notch:

Set CONTENTS(L $+ c$) $\leftarrow$ CONTENTS(L), for TOP[k] $+ c >$ L $\ge$ BASE[$i+1$] $+ c$.

(Note that this should be done for decreasing, not increasing, values of L to avoid losing information. It is possible that TOP[k] $=$ BASE[$i+1$], in which case nothing needs to be moved.)

Set BASE[j] $\leftarrow$ BASE[j] $+ c$, TOP[j] $\leftarrow$ TOP[j] $+ c$, for $i < j \le k$.

b) No k can be found as in (a), but we find the largest k for which $1 \le k < i$ and $\texttt{TOP}[k] < \texttt{BASE}[k+1]$. Now move things *down* one notch:

$$\text{Set} \quad \texttt{CONTENTS}(\texttt{L}) \leftarrow \texttt{CONTENTS}(\texttt{L}+c), \qquad \text{for} \quad \texttt{BASE}[k+1] \le \texttt{L} < \texttt{TOP}[i].$$

(Note that this should be done for increasing values of L.)

$$\text{Set} \quad \texttt{BASE}[j] \leftarrow \texttt{BASE}[j] - c, \qquad \texttt{TOP}[j] \leftarrow \texttt{TOP}[j] - c, \qquad \text{for} \quad k < j \le i.$$

c) We have $\texttt{TOP}[k] = \texttt{BASE}[k+1]$ for all $k \ne i$. Then obviously we cannot find room for the new stack entry, and we must give up.

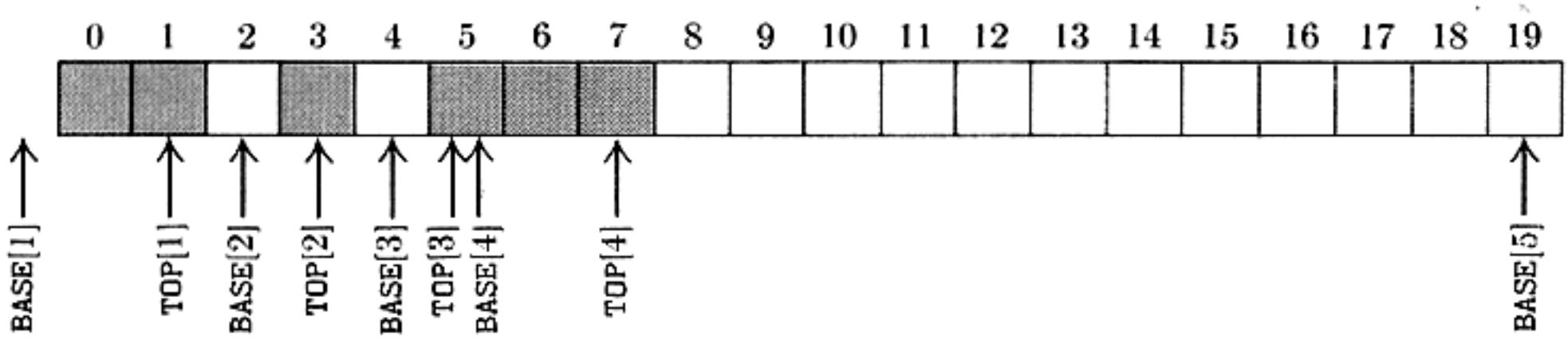


Fig. 4. Example of memory configuration after several insertions and deletions.

Figure 4 illustrates the configuration of memory for the case $n = 4$, $c = 1$, $\mathrm{L}_0 = 0$, $\mathrm{L}_\infty = 20$, after the successive actions

$$\mathrm{I}_1^* \ \mathrm{I}_1^* \ \mathrm{I}_4 \ \mathrm{I}_2^* \ \mathrm{D}_1 \ \mathrm{I}_3^* \ \mathrm{I}_1 \ \mathrm{I}_1^* \ \mathrm{I}_2^* \ \mathrm{I}_4 \ \mathrm{D}_2 \ \mathrm{D}_1.$$

(Here I_j and D_j refer to insertion and deletion in stack j, and an asterisk refers to an occurrence of OVERFLOW, assuming that no space is initially allocated to stacks 1, 2, and 3.)

It is clear that many of the first stack overflows which occur with this method could be eliminated if we choose our initial conditions wisely, instead of allocating all space initially to the nth stack as suggested above. For example, if we expect each stack to be of the same size, we could start out with

$$\texttt{BASE}[j] = \texttt{TOP}[j] = c\left\lfloor \left(\frac{j-1}{n}\right)\left(\frac{\mathrm{L}_\infty - \mathrm{L}_0}{c}\right)\right\rfloor + \mathrm{L}_0 - c. \tag{11}$$

Operating experience with a particular program may suggest better starting values; however, no matter how well the initial allocation is set up, it can save at most a fixed number of overflows, and the effect is noticeable only in the early stages of a program run.

Another possible improvement in the above method would be to make room for more than one new entry each time memory is repacked. The shifting of tables in memory is a time-consuming operation, and we can gain speed by shifting up 2 or 3 at once instead of shifting by 1 several times.

This idea has been exploited by J. Garwick, who suggests a complete repacking of memory when overflow occurs, based on the change in size of each

stack since the last repacking. This algorithm uses an additional array, called OLDTOP[i], $1 \le i \le n$, which retains the value that TOP[i] had just after the previous time memory was allocated. Initially, the tables are set as before, with OLDTOP[i] = TOP[i]. The new algorithm proceeds as follows:

Algorithm G (*Reallocate sequential tables*). Assume that OVERFLOW has occurred in stack i, according to (9). After Algorithm G has been performed, either we will find the memory capacity exceeded or the memory will have been rearranged so that the action NODE(TOP[i]) $\leftarrow$ Y may be done. (Note that TOP[i] has already been increased in (9) before Algorithm G takes place.)

G1. [Initialize.] Set SUM $\leftarrow$ $L_\infty - L_0$, INC $\leftarrow$ 0. Then do step G2 for $1 \le j \le n$. (The effect will be to make SUM equal to the total amount of memory space left, and INC equal to the total amount of increases in table sizes since the last allocation.) After this has been done, go on to step G3.

G2. [Gather statistics.] Set SUM $\leftarrow$ SUM $-$ (TOP[j] $-$ BASE[j]). If TOP[j] $>$ OLDTOP[j], set D[j] $\leftarrow$ TOP[j] $-$ OLDTOP[j] and INC $\leftarrow$ INC $+$ D[j]; otherwise set D[j] $\leftarrow$ 0.

G3. [Is memory full?] If SUM < 0, we cannot proceed.

G4. [Compute allocation factors.] Set $\alpha = (0.1/n)(\text{SUM}/c)$, $\beta = (0.9/\text{INC})(\text{SUM}/c)$. (Here α and β are fractions, not integers, which are to be computed to reasonable accuracy. In the succeeding step, each list will get 10 percent more available space than it now has; the other 90 per cent of this space will be divided proportionately to the amount of increase in table size since the previous allocation.)

G5. [Compute new base addresses.] Set NEWBASE[1] $\leftarrow$ BASE[1]; then for $j = 2, 3, \ldots, n$ set

$$\text{NEWBASE}[j] \leftarrow \text{NEWBASE}[j-1] + \text{TOP}[j-1] - \text{BASE}[j-1] + c\lfloor\alpha\rfloor + c\lfloor \text{D}[j-1]\beta\rfloor.$$

G6. [Repack.] Perform Algorithm R below, and then set OLDTOP[j] $\leftarrow$ TOP[j] for $1 \le j \le n$. ∎

It is readily seen that since $\lfloor\alpha\rfloor \le \alpha$ and $\lfloor \text{D}[j-1]\beta\rfloor \le \text{D}[j-1]\beta$, the reallocation formulas of step G5 will never give anomalous results. Perhaps the most interesting part of this whole algorithm is the general repacking process, which we now describe. Repacking is not trivial, since some portions of memory shift up and others shift down; it is obviously important not to overwrite any of the good information in memory while it is being moved.

Algorithm R (*Relocate sequential tables*). For $1 \le j \le n$ the information specified by BASE[j] and TOP[j] in accord with the conventions stated above is moved to new positions specified by NEWBASE[j], and the values of BASE[j] and TOP[j] are suitably adjusted.

R1. [Initialize.] Set $j \leftarrow 1$. (Note that stack 1 never needs to be moved, so for efficiency the programmer should put the largest stack first if he knows which one will be largest.)

R2. [Find start of shift.] Increase j in steps of 1 until finding either
a) NEWBASE[j] $<$ BASE[j]: go to R3; or
b) NEWBASE[j] $>$ BASE[j]: go to R4; or
c) $j > n$: the algorithm terminates.

R3. [Shift down.] Set $\delta \leftarrow$ BASE[j] $-$ NEWBASE[j]. Set CONTENTS(L $- \delta$) $\leftarrow$ CONTENTS(L), for L $=$ BASE[j] $+ c$, BASE[j] $+ c + 1, \ldots,$ TOP[j] $+ c - 1$. (Note that it is possible for BASE[j] to equal TOP[j], in which case no action is required.) Set BASE[j] $\leftarrow$ NEWBASE[j], TOP[j] $\leftarrow$ TOP[j] $- \delta$. Go to R2.

R4. [Find top of shift.] Find the smallest $k \geq j$ for which NEWBASE[$k + 1$] $\leq$ BASE[$k + 1$]. (*Note:* NEWBASE[$n + 1$] should equal BASE[$n + 1$], so that such a k will always exist.) Then do step R5 for $t = k, k - 1, \ldots, j$; finally set $j \leftarrow k$ and go to R2.

R5. [Shift up.] Set $\delta \leftarrow$ NEWBASE[t] $-$ BASE[t]. Set CONTENTS(L $+ \delta$) $\leftarrow$ CONTENTS(L), for L $=$ TOP[t] $+ c - 1$, TOP[t] $+ c - 2, \ldots,$ BASE[t] $+ c$. (Note that as in step R3, no action may be needed here.) Set BASE[t] $\leftarrow$ NEWBASE[t], TOP[t] $\leftarrow$ TOP[t] $+ \delta$. ∎

In Algorithms G and R we have purposely made it possible to have

$$\text{OLDTOP}[j] \equiv \text{D}[j - 1] \equiv \text{NEWBASE}[j]$$

for $1 \leq j \leq n + 1$, that is, these three tables can share common memory locations since their values are never needed at conflicting times. It will be necessary to perform step G2 for *decreasing* values of j and to set NEWBASE[$n + 1$] $\leftarrow$ BASE[$n + 1$] in step G5 when using this overlap.

We have described these repacking algorithms for stacks, but it is clear that they can be adapted to any relatively addressed tables in which the current information is contained between BASE[j] and TOP[j]. Other pointers (for example, FRONT[j], REAR[j]) could also be attached to the lists, making them serve as a queue or deque. See exercise 8 which considers the case of a queue in detail.

The mathematical analysis of dynamic storage-allocation algorithms is extremely difficult. Some interesting results appear in the exercises below, although they only begin to scratch the surface as far as the general behavior is concerned. Perhaps some of the mathematical results of "queuing theory" can be applied to this study, although the orientation of that theory seems to be different.

As an example of the theory which *can* be derived, suppose we consider the case when the tables grow only by insertion (deletions and subsequent insertions that cancel their effect are ignored), and let us assume further that each table is

expected to fill at the same rate. This situation can be modeled by imagining a sequence of m insertion operations $a_1, a_2, \ldots, a_m$, where each a_i is an integer between 1 and n (representing an insertion on top of stack a_i). For example, the sequence 1, 1, 2, 2, 1 means two insertions to stack 1, followed by two to stack 2, followed by another onto stack 1. We can regard each of the n^m possible specifications $a_1, a_2, \ldots, a_m$ as equally likely, and then we can ask for the average number of times it is necessary to move a word from one location to another during the repacking operations as the entire table is built. For the first algorithm, starting with all available space given to the nth stack, we find that the average number of move operations required is

$$\frac{1}{2}\left(1-\frac{1}{n}\right)\binom{m}{2}. \tag{12}$$

Thus, as we might expect, the number of moves is essentially proportional to the *square* of the number of items in the tables. If we don't start with all the space given to the nth stack, but rather give each of the first $n-1$ stacks t cells to expand in before overflow occurs, we get

$$\frac{1}{2n^t}\sum_{0\le k\le m-t-2}\binom{t-1+k}{k}\binom{m-t-k}{2}\left(1-\frac{1}{n}\right)^{k+1}, \quad \text{for} \quad t\ge 0. \tag{13}$$

This is better than (12), but again it is essentially proportional to m^2. If we put $n=\infty$ in the sum, (13) reduces to

$$\frac{1}{2n^t}\binom{m}{t+2},$$

which is an upper bound that is not too informative; a lower bound is given by the $k=0$ term, namely

$$\frac{1}{2n^t}\left(1-\frac{1}{n}\right)\binom{m-t}{2}.$$

The moral of the story seems to be that a very large number of moves will be made if a reasonably large number of items are put in the tables. This is the price we must pay for the ability to pack a large number of sequential tables together tightly. No theory has been developed to analyze the characteristics of Algorithm G, and it is unlikely that any simple model will be able to describe the characteristics of real-life tables in such an environment anyway.

Experience shows that when memory is only half loaded (i.e., the available space equals half the total space), we need very little rearranging of the tables with Algorithm G; the important thing is perhaps that the algorithm behaves well in the half-full case and that it at least delivers the right answers in the almost-full case.

But let us think about the almost-full case more carefully: When the tables nearly fill memory, Algorithm R takes rather long to perform its job, and to

make matters worse OVERFLOW is much more frequent just before memory space is used up. There are very few programs that will come *close* to filling memory without soon thereafter completely overflowing it; and those that do overflow memory will probably waste enormous amounts of time in Algorithms G and R just before memory is overrun. Unfortunately, an undebugged program frequently will overflow memory capacity. To avoid wasting all this time, a possible suggestion would be to stop Algorithm G in step G3 if SUM is less than $S_{\min}$, where the latter is chosen by the programmer to prevent excessive repacking. When there are many variable-size sequential tables, we should *not* expect to make use of 100 percent of the memory space before storage is exceeded.

EXERCISES

▶ **1.** [*15*] In the queue operations given by (6a), (7a), how many items can be in the queue at one time without OVERFLOW occurring?

▶ **2.** [*22*] Generalize the method of (6a), (7a) to apply to any deque with less than M elements. In other words, give specifications for the other two operations, "delete from rear" and "insert at front."

3. [*21*] Suppose that MIX is extended as follows: The I-field of each instruction is to have the form $8I_1 + I_2$, where $0 \le I_1 < 8$, $0 \le I_2 < 8$. In assembly language one writes "OP ADDRESS, I_1:I_2" or (as presently) "OP ADDRESS, I_2" if $I_1 = 0$. The meaning is to perform first the "address modification" I_1 on ADDRESS, then to perform the "address modification" I_2 on the resulting address, and finally to perform the OP with the new address. The address modifications are defined as follows:

0: M = A
1: M = A + (rI1)
2: M = A + (rI2)
. . .
6: M = A + (rI6)
7: M = resulting address defined from the "ADDRESS,I_1:I_2" fields found in location A. The case $I_1 = I_2 = 7$ in location A is not allowed.

Here A denotes the address before the operation, and M denotes the resulting address after the address modification. In all cases the result is undefined if the value of M does not fit in two bytes plus sign. The execution time is increased by one unit for each "indirect-addressing" (modification 7) operation performed.

As a nontrivial example, suppose that location 1000 contains "NOP 1000,1:7"; location 1001 contains "NOP 1000,2"; and index registers 1 and 2 respectively contain 1 and 2. Then the command "LDA 1000,7:2" is equivalent to "LDA 1004", because

```
1000,7:2 = (1000,1:7),2 = (1001,7),2 = (1000,2),2 = 1002,2 = 1004.
```

a) Using this indirect addressing feature (if necessary), show how to simplify the coding on the right-hand side of (8) so that two instructions are saved per reference to the table. How much faster is your code than (8)?

b) Suppose there are several tables whose base addresses are stored in locations BASE, BASE + 1, BASE + 2, . . . ; how can the indirect addressing feature be used to bring the Ith element of the Jth table into register A in one instruction, assuming that I is in rI1 and J is in rI2?

c) What is the effect of the instruction "ENT4 X,7", assuming the (3:3)-field in location X is zero?

4. [*23*] Assume that MIX has been extended as in exercise 3. Show how to give a *single instruction* (plus auxiliary constants) for each of the following actions:

i) To loop indefinitely because indirect addressing never terminates.

ii) To bring into register A the value CONTENTS(CONTENTS(X)), where X and CONTENTS(X) are link variables whose values are stored in the (0:2)-field of their memory locations.

iii) To bring into register A the value CONTENTS(CONTENTS(CONTENTS(X))), under assumptions like those in (ii).

iv) To bring into register A the contents of location (rI1) + (rI2) + · · · + (rI6).

v) To quadruple the current value of rI6.

▸ **5.** [*35*] The extension of MIX suggested in exercise 3 has an unfortunate restriction that "7:7" is not allowed in an indirectly addressed location.

a) Show that without this restriction it would be necessary for the MIX hardware to be capable of maintaining a long internal stack of three-bit items. (This would be prohibitively expensive hardware, even for a mythical computer like MIX.)

b) Show how such a stack is not needed under the present restriction; in other words, design an algorithm with which the hardware of a computer could perform the desired address modifications without much additional register capacity.

c) Give a milder restriction than that of exercise 3 on the use of 7:7 which alleviates the difficulties of exercise 4(iii), yet which can be cheaply implemented in computer hardware.

6. [*10*] Starting with the memory configuration shown in Fig. 4, determine which of the following sequences of operations causes overflow or underflow. (a) I_1; (b) I_2; (c) I_3; (d) $I_4I_4I_4I_4I_4$; (e) $D_2D_2I_2I_2I_2$.

7. [*12*] Step G4 of Algorithm G indicates a division by the quantity INC. Can INC ever be zero at that point in the algorithm?

▸ **8.** [*25*] Explain how to modify (9), (10) and the repacking algorithms for the case that one or more of the lists is a queue being handled circularly as in (6a), (7a).

▸ **9.** [*M27*] Using the mathematical model described near the end of the section, prove that Eq. (12) is the expected number of moves. (Note that the sequence 1, 1, 4, 2, 3, 1, 2, 4, 2, 1 specifies 0 + 0 + 0 + 1 + 1 + 3 + 2 + 0 + 3 + 6 = 16 moves.)

10. [*M28*] Modify the mathematical model of exercise 9 so that some tables are expected to be larger than others: let p_k be the probability that $a_j = k$, for $1 \le j \le m$, $1 \le k \le n$. Thus $p_1 + p_2 + \cdots + p_n = 1$; the previous exercise considered the special case $p_k = 1/n$ for all k. Determine the expected number of moves, as in Eq. (12), for this more general case. It is possible to rearrange the relative order of the n lists so that lists which are expected to be longer are put to the right (or to the left) of lists expected to be shorter; what is the best relative order for the n lists to minimize the expected number of moves, based on $p_1, p_2, \ldots, p_n$?

11. [*M30*] Generalize the argument of exercise 9 to show that if each stack starts out with t available spaces (so that the first t insertions in any stack cause no movement), then the expected number of moves to make m entries is given by Eq. (13). Note that if $t = 2$, the sequence in exercise 9 specifies $0 + 0 + 0 + 0 + 0 + 3 + 0 + 0 + 3 + 6 = 12$ moves.

12. [*M28*] The advantage of having two tables coexist in memory by growing towards each other, rather than by having them kept in separate independently bounded areas, may be quantitatively estimated (to a certain extent) as follows. Use the model of exercise 9 with $n = 2$; for each of the 2^m equally probable sequences $a_1, a_2, \ldots, a_m$, let there be k_1 1's and k_2 2's. (Here k_1 and k_2 are the respective sizes of the two tables after the memory is full. We are able to run the algorithm with $m = k_1 + k_2$ locations when the tables are adjacent, instead of $2 \max(k_1, k_2)$ locations to get the same effect with separate tables.)

What is the average value of $\max(k_1, k_2)$?

13. [*M50*] The value $\max(k_1, k_2)$ investigated in exercise 12 will be even greater if larger fluctuations in the tables are introduced by allowing random *deletions* as well as random insertions. Suppose we alter the model so that with probability p the sequence value a_j is interpreted as a deletion instead of an insertion; the process continues until $k_1 + k_2$ (the total number of table locations in use) equals m. A deletion from an empty list causes no effect.

For example if $m = 4$, it can be shown that when the above process stops, we get the probability distribution:

the value of (k_1, k_2),	$(4, 0)$ or $(0, 4)$,	$(3, 1)$ or $(1, 3)$,	$(2, 2)$,
occurs with probability:	$\dfrac{1}{16 - 12p + 4p^2}$,	$\dfrac{1}{4}$,	$\dfrac{6 - 6p + 2p^2}{16 - 12p + 4p^2}$.

Thus as p increases, the difference between k_1 and k_2 tends to increase. It is not difficult to show that in the limit as p approaches unity, the distribution of k_1 becomes essentially uniform, and the limiting value of $\max(k_1, k_2)$ is exactly $\frac{3}{4}m$, when m is even. This behavior is quite different from that in the previous exercise (when $p = 0$); however, it may not be extremely significant, since when p approaches unity, the amount of time taken to terminate the process rapidly approaches infinity. The problem posed in this exercise is to examine the dependence of $\max(k_1, k_2)$ on p and m, and to determine asymptotic formulas for fixed p (like $p = \frac{1}{3}$) as m approaches infinity.

14. [*HM40*] Generalize the result of exercise 12 to arbitrary $n \geq 2$, by showing that, when n is fixed and m approaches infinity, the quantity

$$\frac{m!}{n^m} \sum_{\substack{k_1 + \cdots + k_n = m \\ k_1, \ldots, k_n \geq 0}} \frac{\max(k_1, \ldots, k_n)}{k_1! \cdots k_n!}$$

has the asymptotic form $(m/n) + c_n\sqrt{m} + O(1)$. Determine the constants c_2, c_3, c_4, and c_5.

15. [*HM40*] Survey the field of queuing theory to see what the results of that theory tell us about storage allocation algorithms.

16. [*40*] Using a Monte Carlo method, simulate the behavior of Algorithm G under varying distributions of insertions and deletions. What do your experiments imply about the efficiency of Algorithm G? Compare its performance with the algorithm given earlier that shifts up and down one node at a time.

17. [*20*] The text illustrates how two stacks can be located so they grow towards each other, thereby making efficient use of a common memory area. Can two *queues*, or a stack and a queue, make use of a common memory area with the same degree of efficiency?

2.2.3. Linked Allocation

Instead of keeping a linear list in sequential memory locations, we can make use of a much more flexible scheme in which each node contains a link to the next node of the list.

Sequential allocation:

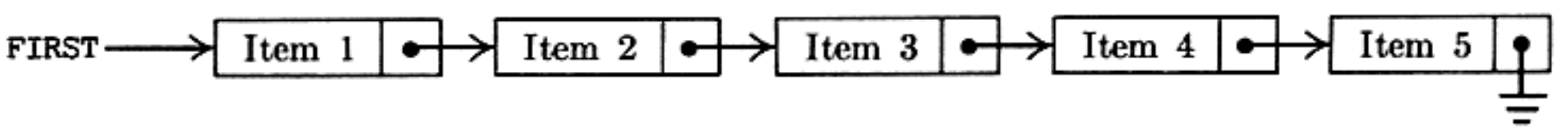
Address	Contents
$L_0 + c$:	Item 1
$L_0 + 2c$:	Item 2
$L_0 + 3c$:	Item 3
$L_0 + 4c$:	Item 4
$L_0 + 5c$:	Item 5

Linked allocation:

Address	Contents	
A:	Item 1	B
B:	Item 2	C
C:	Item 3	D
D:	Item 4	E
E:	Item 5	Λ

Here A, B, C, D, and E are arbitrary locations in the memory, and Λ is the null link (see Section 2.1). The program which uses this table in the case of sequential allocation would have an additional variable or constant whose value indicates that the table is five items in length, or else this information would be specified by a "sentinel" code within item 5 or in the following location. A program for linked allocation would have a link variable or constant that points to A, and from A all the other items of the list can be found.

Recall from Section 2.1 that links are often shown simply by arrows, since the actual memory locations occupied are usually irrelevant. The linked table above might therefore be shown as follows:



(1)

Here FIRST is a link variable pointing to the first node of the list.

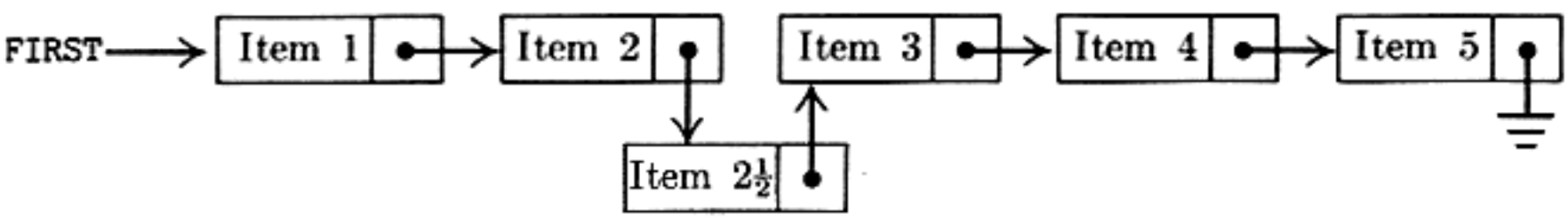
There are several obvious comparisons we can make between these two basic forms of storage:

1) Linked allocation takes up additional memory space for the links. This can be the dominating factor in some situations. However, we frequently find

that the information in a node does not take up a whole word anyway, so there is already space for a link field present. Also, it is possible in many applications to combine several items into one node so that there is only one link for several items of information (see exercise 2.5–2). But even more importantly, there is often an implicit *gain* in storage by the linked memory approach, since tables can overlap, sharing common parts; and in many cases, sequential allocation will not be as efficient as linked allocation unless a rather large number of additional memory locations are left vacant anyway. For example, the discussion at the end of the previous section shows how the systems described there are necessarily inefficient when memory is densely loaded.

2) It is easy to delete an item from within a linked list. For example, to delete item 3 we need only change the link associated with item 2. But with sequential allocation such a deletion generally implies moving a large part of the list up into different locations.

3) It is easy to insert an item into the midst of a list when the linked scheme is being used. For example, to insert an item $2\frac{1}{2}$ into (1) we need change only two links:



(2)

By comparison, this operation would be extremely time-consuming in a long sequential table.

4) References to random parts of the list are much faster in the sequential case. To gain access to the kth item in the list, when k is a variable, takes a fixed time in the sequential case, but it takes k iterations to march down to the right place in the linked case. Thus the usefulness of linked memory is predicated on the fact that in the large majority of applications we want to walk through lists sequentially, not randomly; if items in the middle or at the bottom of the list are needed, we try to keep an additional link variable or list of link variables pointing to the proper places.

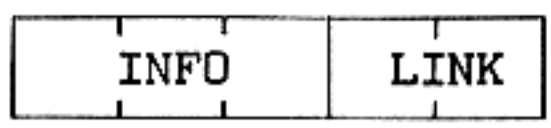
5) The linked scheme makes it easier to join two lists together or to break one apart.

6) The linked scheme lends itself immediately to more intricate structures than simple linear lists. We can have a variable number of variable size lists; any node of the list may be a starting point for another list; the nodes may simultaneously be linked together in several orders corresponding to different lists; and so on.

7) Simple operations, like proceeding sequentially through a list, are slightly faster for sequential lists on many computers. For MIX, the comparison is between "`INC1 1`" and "`LD1 0,1(LINK)`", which is only one cycle different, but many machines do not enjoy the property of being able to load an index register from an indexed location.

Thus we see that the linking technique, which frees us from any constraints imposed by the consecutive nature of computer memory, gives us a good deal more efficiency in some operations, while we lost some capabilities in other cases. It is usually clear which allocation technique will be most appropriate in a given situation, and often both methods are used in different lists of the same program.

In the next few examples we will assume for convenience that a node has one word and that it is broken into the two fields INFO and LINK:



(3)

The use of linked allocation generally implies the existence of some mechanism for finding empty space available for a new node, when we wish to insert some newly created information onto a list. This is usually done by having a special list called the *list of available space*. We will call it the AVAIL list (or, the AVAIL stack, since it is usually treated in a last-in-first-out manner). The set of all nodes not currently in use is linked together in a list just like any other list; the link variable AVAIL refers to the top element of this list. Thus, if we want to set link variable X to the address of a new node, and to reserve that node for future use, we can proceed as follows:

$$\mathtt{X} \leftarrow \mathtt{AVAIL}, \qquad \mathtt{AVAIL} \leftarrow \mathtt{LINK(AVAIL)}. \tag{4}$$

This effectively removes the top of the AVAIL stack and makes X point to the node just removed. *Operation* (4) *occurs so often that we have a special notation for it:* "X $\Leftarrow$ AVAIL" will mean X is set to point to a new node.

When a node is deleted and no longer needed, process (4) can be reversed:

$$\mathtt{LINK(X)} \leftarrow \mathtt{AVAIL}, \qquad \mathtt{AVAIL} \leftarrow \mathtt{X}. \tag{5}$$

This operation puts the node addressed by X back onto the list of raw material; we denote (5) by "AVAIL $\Leftarrow$ X".

Several important things have been omitted from the above discussion of the AVAIL stack. We did not say how to set it up at the beginning of a program; clearly this can be done by (a) linking together all nodes which are to be used for linked memory, (b) setting AVAIL to the address of the first of these nodes, and (c) making the last node link to Λ. The set of all nodes which can be allocated is called the *storage pool*.

A more important omission in our discussion was the test for overflow: we neglected to check in (4) if all available memory space has been taken. The operation X $\Leftarrow$ AVAIL should really be defined as follows:

$$\text{if } \mathtt{AVAIL} = \Lambda, \text{ then } \mathtt{OVERFLOW}; \text{ otherwise } \mathtt{X} \leftarrow \mathtt{AVAIL}, \ \mathtt{AVAIL} \leftarrow \mathtt{LINK(AVAIL)}. \tag{6}$$

The possibility of overflow must always be considered. Here OVERFLOW generally

means that we terminate the program with regrets; or else we can go into a "garbage collection" routine which attempts to find more available space. Garbage collection is discussed in Section 2.3.5.

There is another important technique for handling the AVAIL stack: We often do not know in advance how much memory space is to be used for the storage pool. There may be a sequential table of variable size which is to coexist in memory with the linked tables; in such a case we do not want the linked memory area to take any more space than is absolutely necessary. So suppose that we wish to place the linked memory area in ascending locations beginning with L_0, and that this area is never to extend past the value of variable SEQMIN (which represents the current lower bound of other tables). Then we can proceed as follows, using a new variable POOLMAX:

a) Initially set AVAIL $\leftarrow \Lambda$ and POOLMAX $\leftarrow L_0 - c$, where c is the node size.
b) The operation X $\Leftarrow$ AVAIL becomes the following:

"If AVAIL $\neq \Lambda$, then X $\leftarrow$ AVAIL, AVAIL $\leftarrow$ LINK(AVAIL).
Otherwise set POOLMAX $\leftarrow$ POOLMAX $+ c$; (7)
now if POOLMAX $\geq$ SEQMIN, then OVERFLOW; otherwise set X $\leftarrow$ POOLMAX."

c) When other parts of the program attempt to decrease the value of SEQMIN, they should sound the OVERFLOW alarm if SEQMIN $\leq$ POOLMAX.
d) The operation AVAIL $\Leftarrow$ X is unchanged from (5).

This idea actually represents little more than the previous method with a special recovery procedure substituted for the OVERFLOW situation in (6). The net effect is to keep the storage pool as small as possible. Many people like to use this idea even when *all* lists occupy the storage pool area (so that SEQMIN is constant), since it avoids the rather time-consuming operation of initially linking all available cells together and it sometimes facilitates debugging.

We now see that it is quite easy to maintain a "pool" of available nodes, in such a way that free nodes can be efficiently found and later returned. These methods give us a source of raw material to use in linked tables. Our discussion was predicated on the implicit assumption that all nodes have a fixed size, c; the cases which arise when different sizes of nodes are present are very important, but we will defer that discussion until Section 2.5. Now we will consider a few of the most common list operations in the special case where stacks and queues are involved.

A stack is the simplest kind of linked list. Figure 5 shows a typical stack, with a pointer T to the top of the stack. When the stack is empty, this pointer will have the value Λ.

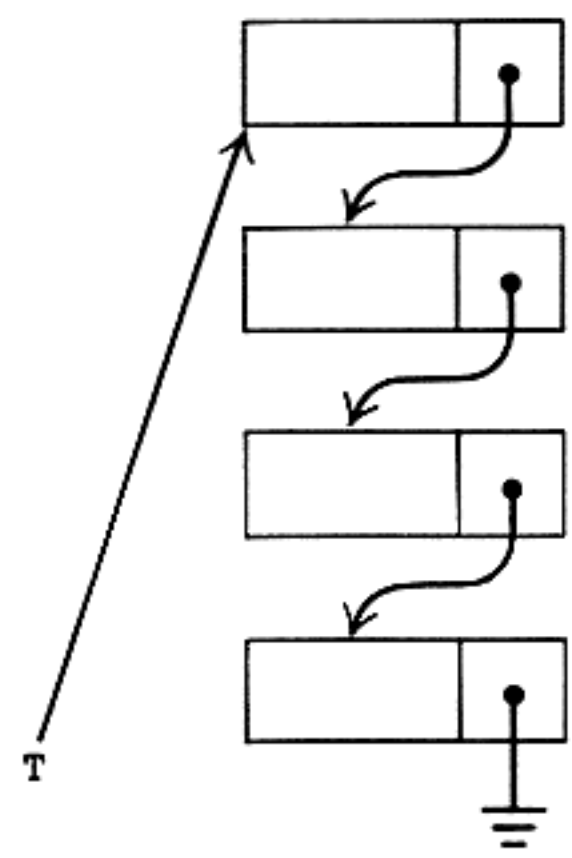


Fig. 5. A linked stack.

It is clear how to insert ("push down") the information Y onto the top of the stack, using an auxiliary pointer variable P:

$$\texttt{P} \Leftarrow \texttt{AVAIL}, \qquad \texttt{INFO(P)} \leftarrow \texttt{Y}, \qquad \texttt{LINK(P)} \leftarrow \texttt{T}, \qquad \texttt{T} \leftarrow \texttt{P}. \tag{8}$$

Conversely, to set Y equal to the information at the top of the stack and to "pop up" the stack:

$$\begin{aligned} &\text{If } \texttt{T} = \Lambda, \text{ then } \texttt{UNDERFLOW};\\ &\qquad \text{otherwise set } \texttt{P} \leftarrow \texttt{T},\ \texttt{T} \leftarrow \texttt{LINK(P)},\ \texttt{Y} \leftarrow \texttt{INFO(P)},\ \texttt{AVAIL} \Leftarrow \texttt{P}. \end{aligned} \tag{9}$$

These operations should be compared with the analogous mechanisms for sequentially allocated stacks, (2a) and (3a) in Section 2.2.2. The reader should study (8) and (9) carefully, since they are extremely important operations.

Before looking at the case of queues, let us see how these operations can be expressed conveniently in programs for MIX. A program for insertion, with P ≡ rI1, can be written as follows:

```
INFO  EQU  0:3          (Definition of INFO field)
LINK  EQU  4:5          (Definition of LINK field)
      LD1  AVAIL        P ← AVAIL.
      J1Z  OVERFLOW     Is AVAIL = Λ?          } P ⇐ AVAIL
      LDA  0,1(LINK)
      STA  AVAIL        AVAIL ← LINK(P).
      LDA  Y                                                    (10)
      STA  0,1(INFO)    INFO(P) ← Y.
      LDA  T
      STA  0,1(LINK)    LINK(P) ← T.
      ST1  T            T ← P.
```

This takes 17 cycles, compared to 12 cycles for the comparable operation with a sequential table (although OVERFLOW in the sequential case would in many cases take considerably longer). In this program, as in others to follow in this chapter, OVERFLOW denotes either an ending routine or a *subroutine* which finds more space and returns to location (rJ) − 2.

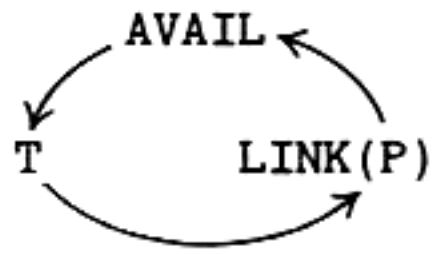
A program for deletion is equally simple:

```
LD1  T            P ← T.
J1Z  UNDERFLOW    Is T = Λ?
LDA  0,1(LINK)
STA  T            T ← LINK(P).
LDA  0,1(INFO)                                                  (11)
STA  Y            Y ← INFO(P).
LDA  AVAIL                          }
STA  0,1(LINK)    LINK(P) ← AVAIL.  } AVAIL ⇐ P
ST1  AVAIL        AVAIL ← P.        }
```

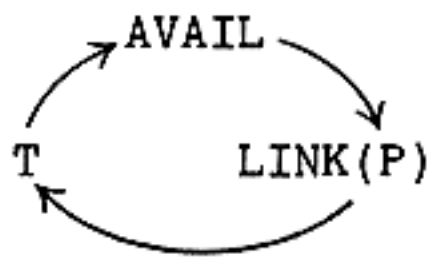
It is worth while to observe that each of these operations involves a cyclic permutation of three links. For example, in the insertion operation let P be the value of AVAIL before the insertion; if P $\neq \Lambda$, we find that after the operation

> the value of AVAIL has become the previous value of LINK(P),
> the value of LINK(P) has become the previous value of T; and
> the value of T has become the previous value of AVAIL.

So the insertion process (except for setting INFO(P) $\leftarrow$ Y) is the cyclic permutation



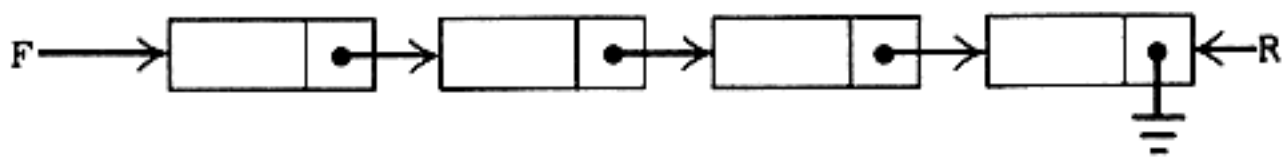
Similarly in the case of deletion, where P has the value of T before the operation and we assume that P $\neq \Lambda$, we have Y $\leftarrow$ INFO(P) and



In these diagrams the fact that the permutation is cyclic is not really a relevant issue, since *any* permutation on three elements that moves every element is cyclic. The important point is rather that precisely three links are permuted in these operations.

The above insertion and deletion algorithms have been described for stacks, but they apply much more generally to insertion and deletion in *any* linear list. Insertion, for example, is performed just before the node pointed to by link variable T. The insertion of item $2\frac{1}{2}$ above [see (2)] would be done by using operation (8) with T = LINK(LINK(FIRST)).

Linked allocation applies in a particularly convenient way to queues. In this case it is easy to see that the links should run from the front of the queue towards the rear, so that when a node is removed from the front, the new front node is directly specified. We will make use of pointers F and R, to the front and rear:



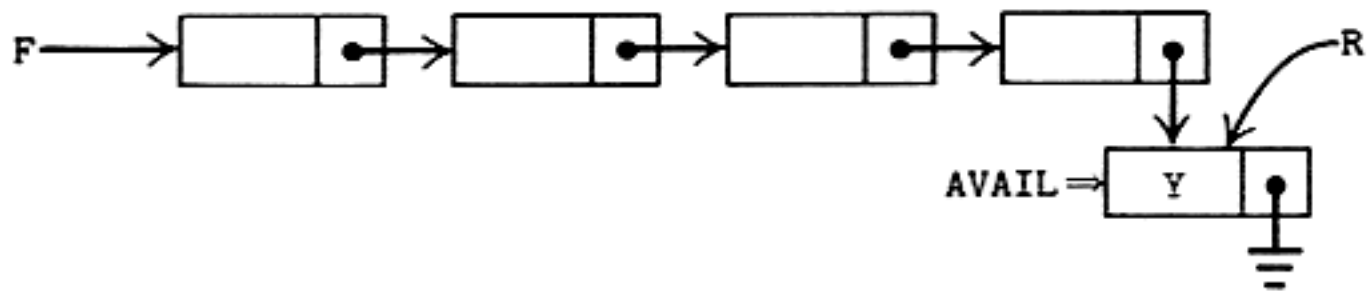
(12)

Except for R, this diagram is abstractly identical to Fig. 5.

Whenever the layout of a list is designed, it is important to specify all conditions carefully, particularly for the case when the list is empty. A failure to do things properly for the case of empty lists is one of the most common programming errors met in connection with linked allocation; the other common error is forgetting to set all of the links when the structure is being manipulated. In order to avoid the first type of error, always examine the "boundary conditions" carefully. To avoid making the second type of error, it is helpful to draw

"before and after" diagrams and to compare them to see which links need to be changed.

Let us illustrate the remarks of the preceding paragraph by applying them to the case of queues. First consider the insertion operation: if (12) is the situation before insertion, the picture after insertion at the rear of the queue should be

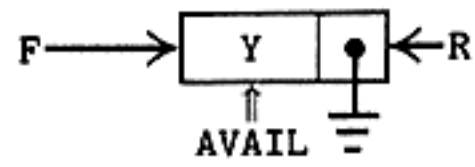


(13)

(The notation used here implies that a new node is obtained from the AVAIL list.) Comparing (12) and (13) shows us how to proceed when inserting the information Y at the rear of the queue:

$$\mathtt{P} \Leftarrow \mathtt{AVAIL}, \quad \mathtt{INFO(P)} \leftarrow \mathtt{Y}, \quad \mathtt{LINK(P)} \leftarrow \Lambda, \quad \mathtt{LINK(R)} \leftarrow \mathtt{P}, \quad \mathtt{R} \leftarrow \mathtt{P}. \tag{14}$$

Let us now consider the "boundary" situation when the queue is empty: in this case the situation before insertion is yet to be determined, and the situation "after" is

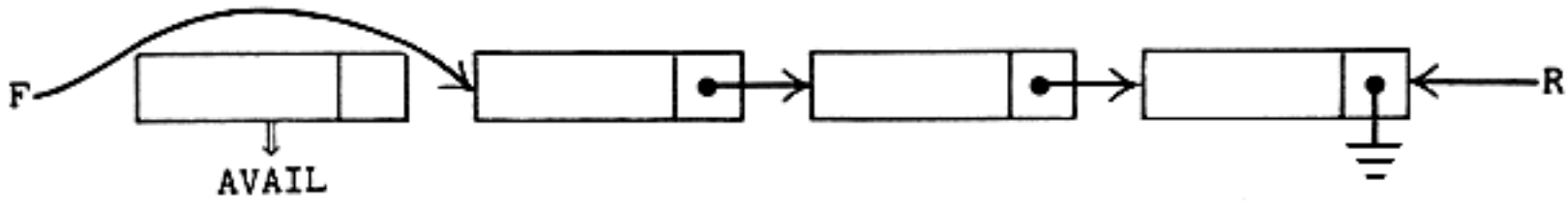


(15)

It is desirable to have operations (14) apply in this case also, even if insertion into an empty queue means that we must change *both* F and R, not only R. We find that (14) will work properly if R = LOC(F) when the queue is empty, *assuming that* F = LINK(LOC(F)); the value of variable F must be stored in the LINK field of its location if this idea is to work. In order to make the testing for an empty queue as efficient as possible, we will let F = Λ in this case. Our policy is therefore that

an empty queue is represented by F = Λ *and* R = LOC(F).

The deletion operation for queues is derived in a similar fashion. If (12) is the situation before deletion, the situation afterward is



(16)

For the boundary conditions we must make sure the deletion operation works when the queue is empty either before or after the operation. These considerations lead us to the following way to do a deletion in general:

If F = Λ, then UNDERFLOW;
otherwise set P ← F, F ← LINK(P), Y ← INFO(P), AVAIL ⇐ P, (17)
and if F = Λ, then set R ← LOC(F).

Note that R must be changed when the queue becomes empty; this is precisely the type of "boundary condition" we should always be watching for.

The above suggestions are not the only way to represent queues in a linearly-linked fashion; we will give other methods later in this chapter. Indeed, none of the operations above are meant to be prescribed as the only way to do something; they are intended as examples of the basic means of operating with linked lists. The reader who has had only a little previous experience with such techniques will find it helpful to reread the present section up to this point before going on.

So far in this chapter we have discussed methods of performing certain operations on tables, but our discussions have always been "abstract" in the sense that we never exhibited actual programs in which the particular techniques were useful. A person is not motivated to study abstractions of a problem until he has seen enough special instances of the problem to arouse his interest. The operations discussed so far (manipulation of variable size lists of information by insertion and deletion, and the use of tables as stacks or queues) are of such wide application, it is hoped that the reader will have encountered them often enough in his own programs that he is already willing to grant their importance. But now we will leave the realm of the abstract as we begin to study a series of significant practical examples of the techniques of this chapter.

Our first example is a problem called *topological sorting*, which is an important process needed in connection with network problems, with so-called PERT charts, and even with linguistics; in fact, it is of potential use whenever we have a problem involving a *partial ordering*. A "partial ordering" of a set S is a relation between the objects of S, which we may denote by the symbol "$\preceq$", satisfying the following properties for any objects x, y, and z (not necessarily distinct) in S:

i) If $x \preceq y$ and $y \preceq z$, then $x \preceq z$. (Transitivity.)
ii) If $x \preceq y$ and $y \preceq x$, then $x = y$. (Antisymmetry.)
iii) $x \preceq x$. (Reflexivity.)

The notation $x \preceq y$ may be read "x precedes or equals y." If $x \preceq y$ and $x \neq y$, we write $x \prec y$ and say "x precedes y." It is easy to see from (i), (ii), and (iii) that we always have

i′) If $x \prec y$ and $y \prec z$, then $x \prec z$. (Transitivity.)
ii′) If $x \prec y$, then $y \not\prec x$. (Asymmetry.)
iii′) $x \not\prec x$. (Irreflexivity.)

The relation denoted by $y \not\prec x$ means "y does not precede x." If we start with a relation $\prec$ satisfying properties (i′), (ii′), and (iii′), we can reverse the above process and define $x \preceq y$ if $x \prec y$ or $x = y$; then properties (i), (ii), and (iii) are true. Therefore we may regard either properties (i), (ii), (iii) or properties

(i′), (ii′), (iii′) as the definition of partial order. [Note that property (ii′) is actually a consequence of (i′) and (iii′).]

Partial orderings occur quite frequently in everyday life as well as in mathematics. As examples from mathematics we can mention the relation $x \leq y$ between real numbers x and y; the relation $x \subseteq y$ between sets of objects; the relation $x \backslash y$ (x divides y) between positive integers. In the case of PERT networks, S is a set of jobs that must be done, and the relation "$x \prec y$" means "x must be done before y."

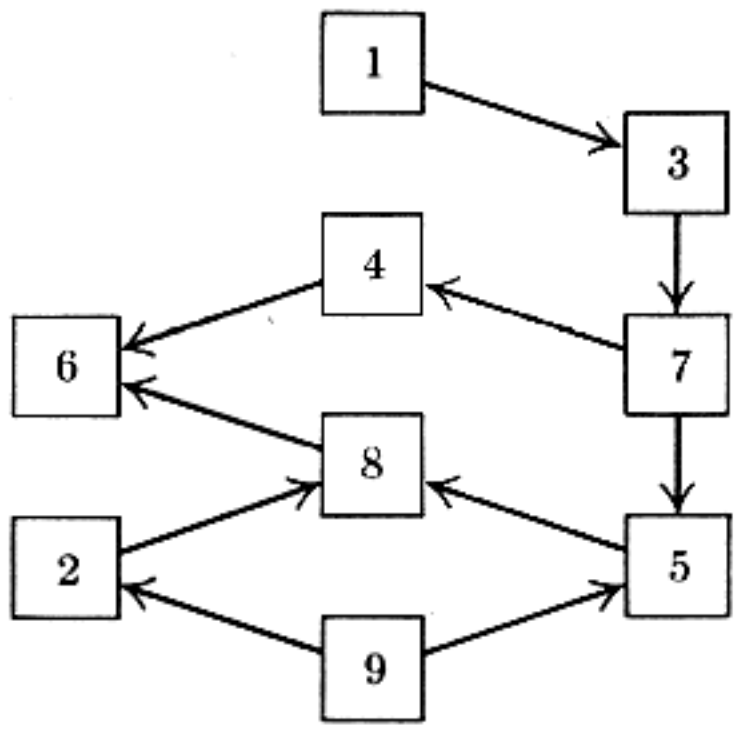


Fig. 6. A partial ordering.

We will naturally assume that S is a finite set, since we want to work with it inside a computer. A partial ordering on a finite set can always be illustrated by drawing a diagram such as Fig. 6, in which the objects are represented by small boxes and the relation is represented by arrows between these boxes; $x \prec y$ means there is a path from the box labeled x to box y which follows the direction of the arrows. Property (ii) of partial ordering means there are *no closed loops* (no paths that close on themselves) in the diagram. If an arrow were drawn from 4 to 1 in Fig. 6, we would no longer have a partial ordering.

The problem of topological sorting is to "embed the partial order in a linear order," i.e., to arrange the objects into a linear sequence $a_1, a_2, \ldots, a_n$ such that whenever $a_j \prec a_k$, we have $j < k$. Graphically, this means that the boxes are to be rearranged into a line so that all arrows go towards the right (see Fig. 7). It is not immediately obvious that such a rearrangement is possible in every case, although such a rearrangement certainly could not be done if any loops were present. Therefore the algorithm we will give is interesting not only because it does a useful operation, but also because it proves that this operation is *possible* for every partial ordering.

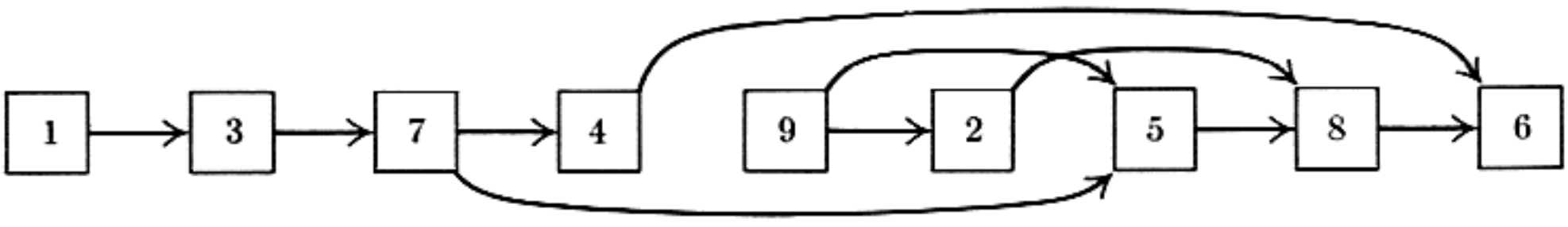


Fig. 7. The ordering relation of Fig. 6 after topological sorting.

As an example of topological sorting, imagine a large glossary containing definitions of technical terms. We can write $w_2 \prec w_1$ if the definition of word w_1 depends directly or indirectly on that of word w_2. This relation is a partial ordering provided that there are no "circular" definitions. The problem of topological sorting in this case is to *find a way to arrange the words in the glossary so that no term is used before it has been defined.* Analogous problems arise in writing programs to process the declarations in certain assembly and compiler languages; they also arise in writing a user's manual describing a computer language or in writing textbooks about information structures.

There is a very simple way to do topological sorting: We start by taking an object which is not preceded by any other object in the ordering. This object may be placed first in the output. Now we remove this object from the set S; the resulting set is again partially ordered, and the process can be repeated until the whole set has been sorted. For example, in Fig. 6 we could start by removing 1 or 9; after 1 has been removed, 3 can be taken, and so on. The only way in which this algorithm could fail would be if there were a nonempty partially ordered set in which every element was preceded by another; for in such a case the algorithm would find nothing to do. But if every element is preceded by another, we could construct an arbitrarily long sequence $b_1, b_2, b_3, \ldots$ in which $b_{j+1} \prec b_j$; since S is finite, we must have $b_j = b_k$ for some $j < k$, but this implies that $b_k \preceq b_{j+1}$, contradicting (ii).

In order to implement this process efficiently by computer, we need to be ready to perform the actions described above, i.e., to locate objects which are not preceded by any others, and to remove them from the set. Our implementation is also influenced by the desired input and output characteristics. The most general program would accept alphabetic names for the objects and would allow for huge numbers of objects to be sorted—more than could possibly fit in the computer memory at once. These complications would obscure the main points we are trying to make here, however; the handling of alphabetic data can be done efficiently by using the methods of Chapter 6, and the handling of large networks is left as an interesting project for the reader.

Therefore we will assume that the objects to be sorted are numbered from 1 to n in any order. The input to the program will be on tape unit 1: each tape record contains 50 pairs of numbers; the pair (j, k) means object j precedes object k. The first pair, however, is $(0, n)$, where n is the number of objects. The pair $(0, 0)$ terminates the input. We shall assume that n plus the number of relation pairs will fit comfortably in memory; and we shall assume that it is not necessary to check the input for validity. The output is to be the numbers of the objects in sorted order, followed by the number 0, on tape unit 2.

As an example of the input, we might have the pairs

$$\begin{array}{c} 9 \prec 2, \quad 3 \prec 7, \quad 7 \prec 5, \quad 5 \prec 8, \\ 8 \prec 6, \quad 4 \prec 6, \quad 1 \prec 3, \\ 7 \prec 4, \quad 9 \prec 5, \quad 2 \prec 8. \end{array} \tag{18}$$

It is not necessary to give any more pairs than are needed to characterize the desired partial ordering. Thus additional relations like $9 \prec 8$ (which can be deduced from $9 \prec 5$ and $5 \prec 8$) may be omitted from or added to the input without harm. In general, it is only necessary to give the pairs corresponding to arrows on a diagram such as Fig. 6.

The algorithm which follows uses a sequential table X[1], X[2], . . . , X[n], and each node X[k] has the form

+	0	COUNT[k]	TOP[k]

.

Here COUNT[k] is the *number of direct predecessors* of object k (i.e., the number of pairs $j \prec k$ which have appeared in the input), and TOP[k] is a link to the beginning of the *list of direct successors* of object k. The latter list contains entries in the format

+	0	SUC	NEXT

,

where SUC is a direct successor of k and NEXT is the next item of the list. As an example of these conventions, Fig. 8 shows the schematic contents of memory corresponding to the input (18).

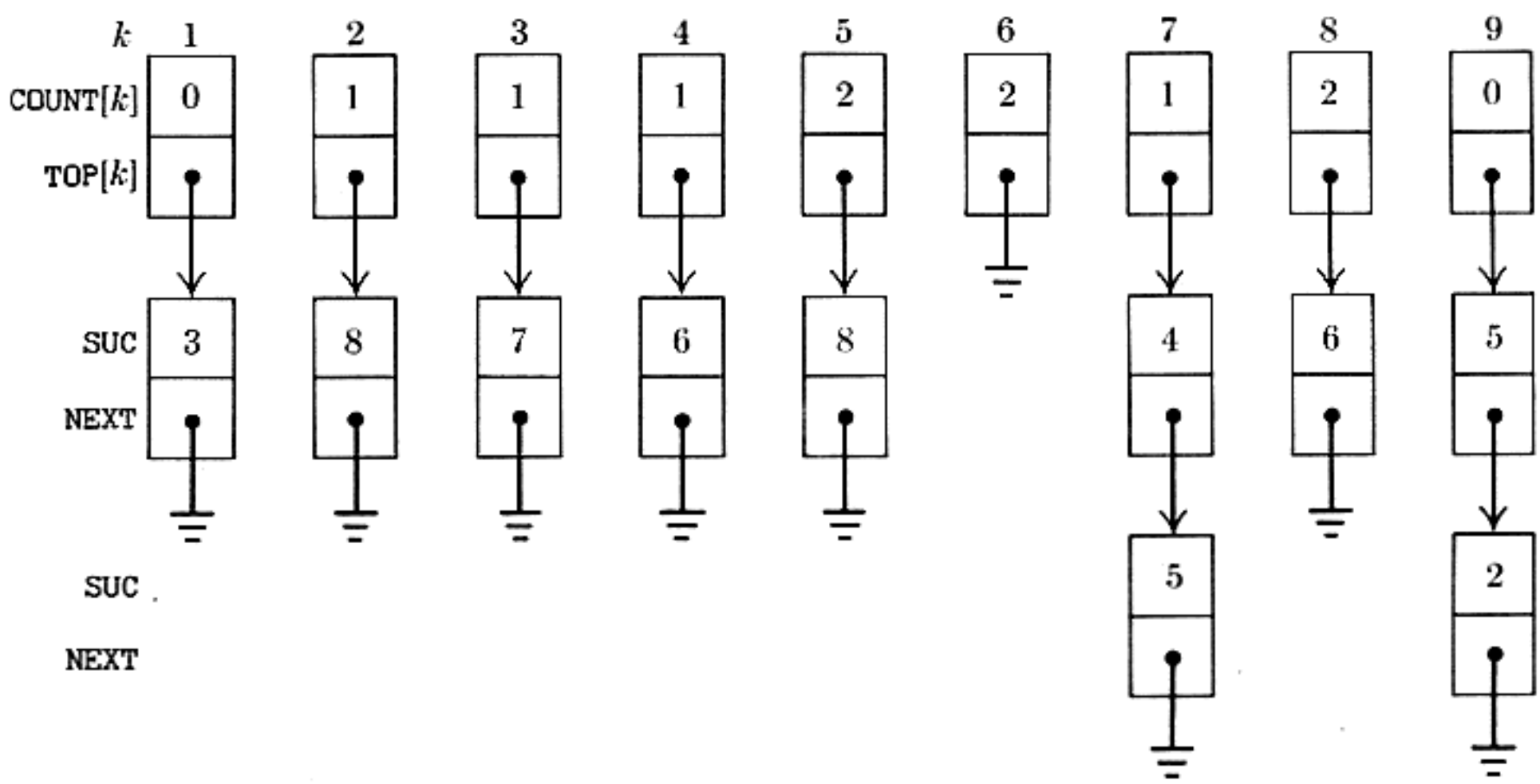


Fig. 8. Computer representation of Fig. 6 corresponding to the relations (18).

Using this memory layout, it is not difficult to work out the algorithm. It is a matter of outputting the nodes whose COUNT field is zero, then decreasing the COUNT fields of all successors of those nodes by one. The trick is to avoid doing any "searching" for nodes whose COUNT field is zero, and this can be done by maintaining a queue containing those nodes whose COUNT field has been reduced to zero but which have not yet been output. The links for this queue

are kept in the COUNT field, which by now has served its previous purpose; for clarity in the algorithm below, we use the notation QLINK[k] to stand for COUNT[k] when that field is no longer being used to keep a count.

Algorithm T (*Topological sort*). This algorithm inputs pairs of relations "$j \prec k$", $1 \le j, k \le n$, indicating that object j precedes object k in a certain partial ordering. The output is the set of objects embedded in linear order. The internal tables used are QLINK[0], COUNT[1] = QLINK[1], COUNT[2] = QLINK[2], ..., COUNT[n] = QLINK[n]; TOP[1], TOP[2], ..., TOP[n]; a storage pool with one node for each input relation and with SUC and NEXT fields as shown above; P, a link variable used to refer to the nodes in the storage pool; F and R, integer-valued variables used to refer to the front and rear of a queue whose links are in the QLINK table; and N, a variable which counts how many objects have yet to be output.

T1. [Initialize.] Input the value of n. Set COUNT[k] $\leftarrow 0$ and TOP[k] $\leftarrow \Lambda$ for $1 \le k \le n$. Set N $\leftarrow n$.

T2. [Next relation.] Get the next relation "$j \prec k$" from the input; if the input has been exhausted, however, go to T4.

T3. [Record the relation.] Increase COUNT[k] by one. Set

$$\texttt{P} \Leftarrow \texttt{AVAIL},\ \texttt{SUC(P)} \leftarrow k,\ \texttt{NEXT(P)} \leftarrow \texttt{TOP}[j],\ \texttt{TOP}[j] \leftarrow \texttt{P}.$$

[This is operation (8).] Go to T2.

T4. [Scan for zeros.] (At this point we have completed the input phase; the input (18) would now have been transformed into the computer representation shown in Fig. 8. Now we initialize the queue of output, which is linked together in the QLINK field.) Set R $\leftarrow 0$ and QLINK[0] $\leftarrow 0$. For $1 \le k \le n$ examine COUNT[k], and if it is zero, set QLINK[R] $\leftarrow k$ and R $\leftarrow k$. After this has been done for all k, set F $\leftarrow$ QLINK[0] (which will contain the first value k encountered for which COUNT[k] was zero).

T5. [Output front of queue.] Output the value of F. If F $= 0$, go to T8; otherwise, set N $\leftarrow$ N $- 1$, and set P $\leftarrow$ TOP[F].

T6. [Erase relations.] If P $= \Lambda$, go to T7. Otherwise decrease COUNT[SUC(P)] by one, and if it has thereby gone down to zero, set QLINK[R] $\leftarrow$ SUC(P) and R $\leftarrow$ SUC(P). Set P $\leftarrow$ NEXT(P) and repeat this step. (We are removing all relations of the form "F $\prec k$" for some k from the system, and putting new nodes into the queue when all their predecessors have been output.)

T7. [Remove from queue.] Set F $\leftarrow$ QLINK[F] and go back to T5.

T8. [End of process.] The algorithm terminates. If N $= 0$, we have output all of the object numbers in the desired "topological order," followed by a zero. Otherwise the N object numbers not yet output contain a loop, in violation of the hypothesis of partial order. (See exercise 23 for an algorithm which prints out the contents of one such loop.) ∎

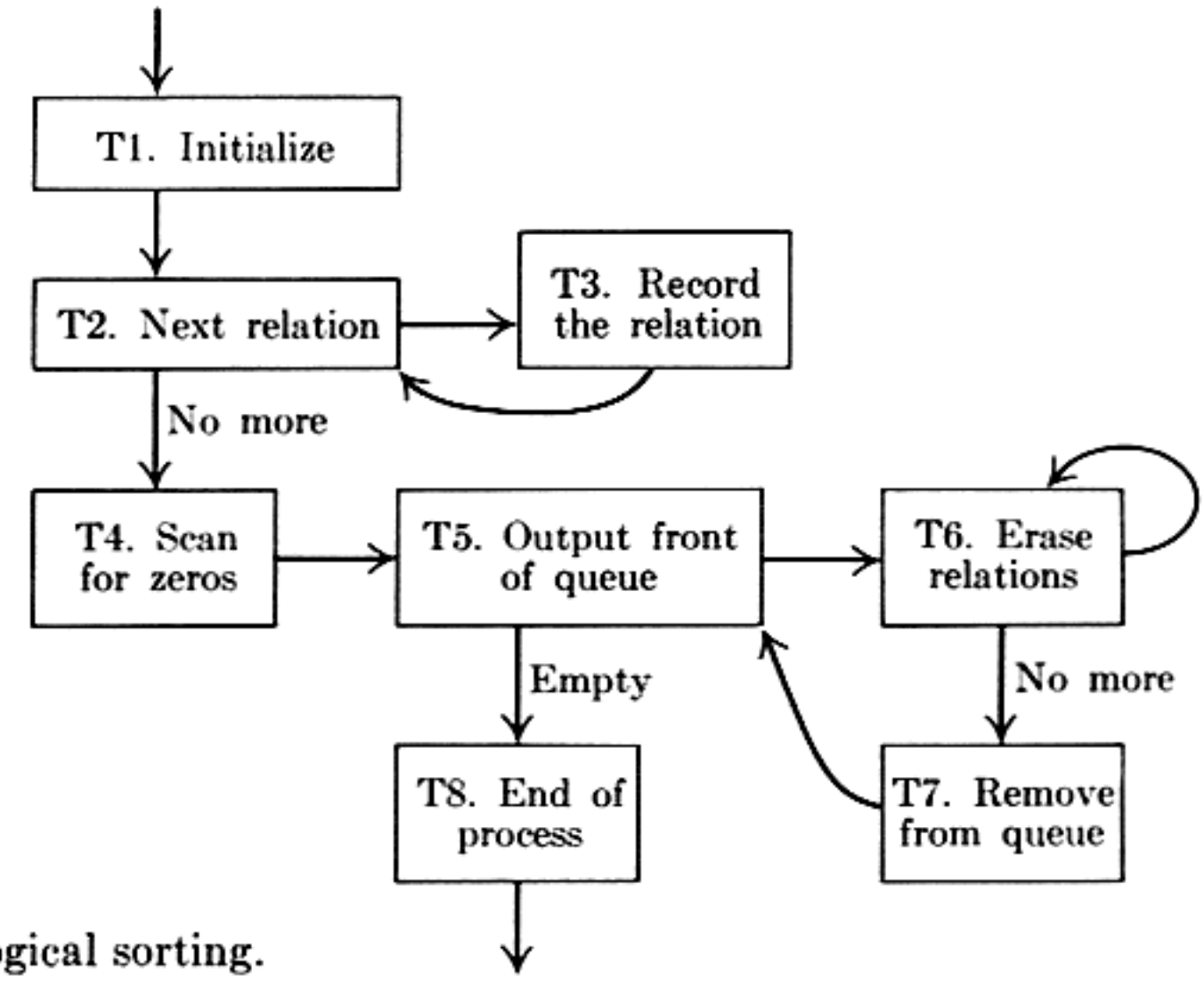


Fig. 9. Topological sorting.

The reader will find it helpful to try this algorithm by hand on the input (18). Algorithm T shows a nice interplay between sequential memory and linked memory techniques. Sequential memory is used for the main table X[1], . . . , X[n], which contains the COUNT[k] and TOP[k] entries, because we want to make references to "random" parts of this table in step T3. (If the input were alphabetic, however, another type of table would be used for speedier search, as in Chapter 6.) Linked memory is used for the tables of "immediate successors," since these table entries come in random order in the input. The queue of nodes waiting to be output is kept in the midst of the sequential table by linking the nodes together in output order. This linking is done by table index instead of by address; i.e., when the front of the queue is X[k], we have F = k instead of F = LOC(X[k]). The queue operations used in steps T4, T6, and T7 are not identical to those in (14) and (17), since we are taking advantage of special properties of the queue in this system; no nodes need to be created or returned to available space during this part of the algorithm.

The coding of Algorithm T in MIX assembly language has a few more points of interest. Since no deletion from tables is made in the algorithm (because no storage must be freed for later use), the P ⇐ AVAIL operation can be done in an extremely simple way, as shown in lines 19 and 32 below; we need not keep any linked pool of memory, we can choose new nodes consecutively. The program includes complete input and output with magnetic tape, according to the conventions mentioned above, except that for simplicity no buffering is shown. The reader should not find it very difficult to follow the details of the coding in this program, since it corresponds directly with Algorithm T. The efficient use of index registers, which is an important aspect of linked memory processing, is illustrated here.

Program T (*Topological sort*). In this program, the following equivalences should be noted: rI6 ≡ N, rI5 ≡ buffer pointer, rI4 ≡ k, rI3 ≡ j and R, rI2 ≡ AVAIL and P, rI1 ≡ F, TOP[j] ≡ X + j(4:5), COUNT[k] ≡ QLINK[k] ≡ X + k(2:3).

```
*  BUFFER AREA AND FIELD DEFINITIONS
COUNT    EQU    2:3                          Definition of symbolic
QLINK    EQU    2:3                               names of fields
TOP      EQU    4:5
SUC      EQU    2:3
NEXT     EQU    4:5
TAPEIN   EQU    1                            Input is on tape unit 1
TAPEOUT  EQU    2                            Output is on tape unit 2
BUFFER   ORIG   *+100                        Tape buffer area
         CON    -1                           Sentinel at end of buffer
*  INPUT PHASE
TOPSORT  IN     BUFFER(TAPEIN)        1      T1. Initialize. Read in first
         JBUS   *(TAPEIN)                          tape block, wait for completion.
1H       LD6    BUFFER+1              1      N ← n.
         ENT4   0,6                   1
         STZ    X,4                   n+1    Set COUNT[k] ← 0, TOP[k] ← Λ,
         DEC4   1                     n+1        for 0 ≤ k ≤ n.
         J4NN   *-2                   n+1    (Anticipate QLINK[0] ← 0 in step T3.)
         ENT2   X,6                   1      Available storage starts after X[n].
         ENT5   BUFFER+2              1      Prepare to read first pair (j, k).
2H       LD3    0,5                   m+b    T2. Next relation.
         J3P    3F                    m+b    Is j > 0?
         J3Z    4F                    b      Is input exhausted?
         IN     BUFFER(TAPEIN)        b-1    Sentinel sensed, read another
         JBUS   *(1)                           tape record, wait for completion.
         ENT5   BUFFER                b-1    Reset buffer pointer.
         JMP    2B                    b-1
3H       LD4    1,5                   m      T3. Record the relation.
         LDA    X,4(COUNT)            m      COUNT[k]
         INCA   1                     m        + 1
         STA    X,4(COUNT)            m          → COUNT[k].
         INC2   1                     m      AVAIL ← AVAIL + 1.
         LDA    X,3(TOP)              m      TOP[j]
         STA    0,2(NEXT)             m        → NEXT(P).
         ST4    0,2(SUC)              m      k → SUC(P).
         ST2    X,3(TOP)              m      P → TOP[j].
         INC5   2                     m      Increase buffer pointer.
         JMP    2B                    m
4H       IOC    0(TAPEIN)             1      Rewind input tape.
         ENT4   0,6                   1      T4. Scan for zeros.
         ENT5   0                     1      Reset buffer pointer for output.
         ENT3   0                     1      R ← 0.
4H       LDA    X,4(COUNT)            n      Examine COUNT[k].
         JAP    *+3                   n      Is it nonzero?
         ST4    X,3(QLINK)            a      QLINK[R] ← k.
         ENT3   0,4                   a      R ← k.
         DEC4   1                     n
         J4P    4B                    n      n ≥ k ≥ 1.
```

```
*  SORTING PHASE
          LD1    X(QLINK)              1        F ← QLINK[0].
5H        JBUS   *(TAPEOUT)                     T5. Output front of queue.
          ST1    BUFFER,5              n + 1    Store F in buffer area.
          J1Z    8F                    n + 1    Is F zero?
          INC5   1                     n        Advance buffer pointer.
          ENTA   -100,5                n
          JAN    *+3                   n        Test if buffer is full.
          OUT    BUFFER(TAPEOUT)       c - 1    If so, output a tape record.
          ENT5   0                     c - 1    Reset buffer pointer.
          DEC6   1                     n        N ← N - 1.
          LD2    X,1(TOP)              n        P ← TOP[F].
          J2Z    7F                    n        T6. Erase relations.
6H        LD4    0,2(SUC)              m        rI4 ← SUC(P).
          LDA    X,4(COUNT)            m        COUNT[rI4]
          DECA   1                     m          - 1
          STA    X,4(COUNT)            m            → COUNT[rI4].
          JAP    *+3                   m        Has zero been reached?
          ST4    X,3(QLINK)            n - a    If so, set QLINK[R] ← rI4.
          ENT3   0,4                   n - a    R ← rI4.
          LD2    0,2(NEXT)             m        P ← NEXT(P).
          J2P    6B                    m        If P ≠ Λ, repeat.
7H        LD1    X,1(QLINK)            n        T7. Remove from queue.
          JMP    5B                    n        F set to QLINK[F]
8H        OUT    BUFFER(TAPEOUT)       1        T8. End of process.
          IOC    0(TAPEOUT)            1        Output last block and rewind.
          HLT    0,6                   1        Stop, displaying N on console.
X         EQU    *                              Beginning of table area
          END    TOPSORT                                  ▮
```

The analysis of Algorithm T is quite simple with the aid of Kirchhoff's law; the execution time has the approximate form $c_1m + c_2n$, where m is the number of input relations, n is the number of objects, and c_1 and c_2 are constants. It is hard to imagine a faster algorithm for this problem! The exact quantities in the analysis are given with Program T above, where a = number of objects with no predecessor, b = number of tape records in input $= \lceil (m + 2)/50 \rceil$, and c = number of tape records in output $= \lceil (n + 1)/100 \rceil$. Exclusive of input/output operations, the total running time in this case is only $(32m + 25n + 7b + 2c + 16)u$.

A topological sorting technique similar to Algorithm T (but without the important feature of the queue links) was first published by A. B. Kahn, *CACM* **5** (1962), 558–562.

EXERCISES

▶ **1.** [*10*] Operation (9) for popping up a stack mentions the possibility of `UNDERFLOW`; why doesn't operation (8) for pushing down a stack mention the possibility of `OVERFLOW`?

2. [*22*] Write a "general purpose" MIX subroutine to do the insertion operation, (10). This subroutine should have the following specifications:

Calling sequence: JMP INSERT Jump to subroutine.
NOP T Location of pointer variable
Entry conditions: rA = information to be put into the INFO field of a new node.
Exit conditions: The stack whose pointer is the link variable T has the new node on top; rI1 = T; rI2, rI3 are altered.

3. [*22*] Write a "general purpose" MIX subroutine to do the deletion operation, (11). This subroutine should have the following specifications:

Calling sequence: JMP DELETE Jump to subroutine.
NOP T Location of pointer variable
JMP UNDERFLOW First exit, if UNDERFLOW sensed
Entry conditions: None
Exit conditions: If the stack whose pointer is the link variable T is empty, the first exit is taken; otherwise the top node of that stack is deleted, and exit is made to three locations following "JMP DELETE". In the latter case, rI1 = T and rA is the contents of the INFO field of the deleted node. In either case, rI2 and rI3 are used by this subroutine.

4. [*22*] The program in (10) is based on the operation P ⇐ AVAIL, as given in (6). Show how to write an OVERFLOW subroutine so that, without *any* change in the coding (10), the operation P ⇐ AVAIL makes use of SEQMIN, as given by (7). For general-purpose use, your subroutine should not change the contents of any registers and it should exit to location (rJ − 2), instead of the usual (rJ).

▸ **5.** [*24*] Operations (14) and (17) give the effect of a queue; show how to define the further operation "insert at front" so as to obtain all the actions of an output-restricted deque. How could the operation "delete from rear" be defined (so that we would have a general deque)?

6. [*21*] In operation (14) we set LINK(P) ← Λ, while the very next insertion at the rear of the queue will change the value of this same link field. Show how the setting of LINK(P) in (14) could be eliminated if we make a change to the testing of "F = Λ" in (17).

▸ **7.** [*23*] Design an algorithm to "invert" a linked linear list such as (1), i.e., to change its links so that the items appear in the opposite order. [Thus, if the list (1) were inverted, we would have FIRST linking to the node containing item 5; that node would link to the one containing item 4; etc.] Assume that the nodes have the form (3).

8. [*24*] Write a MIX program for the problem of exercise 7, attempting to design your program to operate as fast as possible.

9. [*20*] Which of the following relations is a partial ordering on the specified set S? [*Note:* If the relation "$x \prec y$" is defined below, the intent is to define the relation "$x \preceq y \equiv (x \prec y \text{ or } x = y)$," and then to determine whether $\preceq$ is a partial ordering.] (a) S = all rational numbers, $x \prec y$ means $x > y$. (b) S = all people, $x \prec y$ means x is an ancestor of y. (c) S = all integers, $x \preceq y$ means x is a multiple of y (that is, $x \bmod y = 0$). (d) S = all the mathematical results proved in this book, $x \prec y$ means the proof of y depends upon the truth of x. (e) S = all positive integers, $x \preceq y$ means

$x + y$ is even. (f) S = a set of subroutines, $x \prec y$ means "x calls y," that is, y may be in operation while x is in operation.

10. [*M21*] Given that "$\subset$" is a relation which satisfies properties (i) and (ii) of a partial ordering, prove that the relation "$\preceq$", defined by the rule "$x \preceq y$ if and only if $x = y$ or $x \subset y$," satisfies all three properties of a partial ordering.

▶ **11.** [*24*] The result of topological sorting is not always completely determined, since there may be several ways to arrange the nodes and to satisfy the conditions of topological order. Find all possible ways to arrange the nodes of Fig. 6 into topological order.

12. [*M20*] There are 2^n subsets of a set of n elements, and these subsets are partially ordered by the set-inclusion relation. Give two interesting ways to arrange these subsets in topological order.

13. [*M48*] How many ways are there to arrange the 2^n subsets described in exercise 12 into topological order? (Give the answer as a function of n.)

14. [*M24*] A *linear ordering* of a set S is a partial ordering which satisfies the additional condition

(iv) For any two objects x, y in S, either $x \preceq y$ or $y \preceq x$.

Prove directly from the definitions given that a topological sort can result in only one possible output if and only if the relation $\preceq$ is a linear ordering. (You may assume that the set S is finite.)

15. [*M25*] Show that for any partial ordering on a finite set S there is a *unique* set of irredundant pairs of relations [such as (18) corresponding to Fig. 6] which characterizes this ordering. Is the same fact true also when S is an infinite set?

16. [*M22*] Given any partial ordering on a set $S = \{x_1, \ldots, x_n\}$, we can construct its "incidence matrix" (a_{ij}), where $a_{ij} = 1$ if $x_i \preceq x_j$, and $a_{ij} = 0$ otherwise. Show that there is a way to permute the rows and columns of this matrix so that all entries below the diagonal are zero.

▶ **17.** [*21*] What output does Algorithm T produce if it is presented with the input (18)?

18. [*20*] What, if anything, is the significance of the values of `QLINK[0]`, `QLINK[1]`, ..., `QLINK[n]` when Algorithm T terminates?

19. [*18*] In Algorithm T we examine the front position of the queue in step T5, but do not remove that element from the queue until step T7. What would happen if we set `F` ← `QLINK[F]` at the conclusion of step T5, instead of in T7?

▶ **20.** [*24*] Algorithm T uses `F`, `R`, and the `QLINK` table to obtain the effect of a queue which contains those nodes whose `COUNT` field has become zero but whose successor relations have not yet been removed. Could a stack be used for this purpose instead of a queue? If so, compare the resulting algorithm with Algorithm T.

21. [*21*] Would Algorithm T still perform a valid topological sort if one of the relations "$j \prec k$" were repeated several times in the input? What if the input contained a relation of the form "$j \prec j$"?

22. [*23*] Program T assumes that its input tape contains valid information, but a program that is intended for general use should always make careful tests on its input so that clerical errors can be detected, and the program cannot "destroy itself." For

example, if one of the input relations for k were negative, Program T may erroneously change one of its own instructions when storing into X[k]. Suggest ways to modify Program T so that it is suitable for general use.

▶ **23.** [*27*] When the topological sort algorithm cannot proceed because it has detected a loop in the input (see step T8), it is usually of no use just to stop and say, "There was a loop." It is helpful to print out one of the loops, thereby showing part of the input which was in error. Extend Algorithm T so that it will do this additional printing of a loop when necessary. [*Hint:* The text gives a proof for the existence of a loop when N > 0 in step T8; that proof suggests an algorithm.]

24. [*24*] Incorporate the extensions of Algorithm T made in exercise 23 into Program T.

25. [*44*] Design as efficient an algorithm as possible for doing a topological sort of very large sets S, which have considerably more nodes than the computer memory can contain. Assume that the input, output, and temporary working space are done with magnetic tape. [*Possible hint:* A conventional sort of the input allows us to assume that all relations for a given node appear together. But then what can be done? (In particular, we must consider the worst case in which the given ordering is already a linear ordering that has been wildly permuted; if possible we want to avoid doing $O(n)$ iterations through the entire data tape.)]

26. [*29*] (Subroutine allocation.) Suppose that we have a tape containing the main "subroutine library" for a computer installation in relocatable form. The loading routine wants to determine the amount of relocation for each subroutine used so it can make one pass through the tape to load the necessary routines. The problem is that some subroutines require others to be present in memory. Infrequently used subroutines (which appear toward the end of the tape) may call on frequently used subroutines (which appear toward the beginning of the tape), and we want to know all of the subroutines which are required, before passing through the tape.

One way to tackle this problem is to have a "tape directory" which fits in memory. The loading routine has access to two tables:

a) The tape directory. This table is composed of variable-length nodes having the form

B	SPACE	LINK
B	SUB1	SUB2
	⋮	
B	SUBn	0

or

B	SPACE	LINK
B	SUB1	SUB2
	⋮	
B	SUB(n−1)	SUBn

where SPACE is the number of words of memory required by the subroutine; LINK is a link to the directory entry for the subroutine which appears on the tape following this subroutine; SUB1, SUB2, . . . , SUBn ($n \geq 0$) are links to the directory entries for any other subroutines required by this one; B = 0 on all words except the last, B = −1 on the last word of a node. The address of the directory entry for the first subroutine on the library tape is specified by the link variable FIRST.

b) The list of subroutines directly referred to by the program to be loaded. This is stored in consecutive locations X[1], X[2], . . . , X[N], where N $\geq$ 0 is a variable known

to the loading routine. Each entry in this list is a link to the directory entry for the subroutine desired.

The loading routine also knows MLOC, the amount of relocation to be used for the first subroutine loaded.

As a small example, consider the following configuration:

Tape directory

	B	SPACE	LINK
1000:	0	20	1005
1001:	−1	1002	0
1002:	−1	30	1010
1003:	0	200	1007
1004:	−1	1000	1006
1005:	−1	100	1003
1006:	−1	60	1000
1007:	0	200	0
1008:	0	1005	1001
1009:	−1	1006	0
1010:	−1	20	1006

List of subroutines needed

X[1] = 1003
X[2] = 1010

N = 2
FIRST = 1002
MLOC = 2400

The tape directory in this case shows that the subroutines on tape are 1002, 1010, 1006, 1000, 1005, 1003, and 1007 in that order. Subroutine 1007 takes 200 locations and implies the use of subroutines 1005, 1001, and 1006; etc. The program to be loaded requires subroutines 1003 and 1010, which are to be placed into locations $\geq$ 2400. These subroutines in turn imply that 1000, 1006, and 1002 must also be loaded.

The subroutine allocator is to change the X-table so that each entry X[1], X[2], . . . has the form

+	0	BASE	SUB

(except the last entry which is explained below), where SUB is a subroutine to be loaded, and BASE is the amount of relocation. These entries are to be in the order in which the subroutines appear on tape. In the above example one possible answer would be

	BASE	SUB
X[1]:	2400	1002
X[2]:	2430	1010
X[3]:	2450	1006
X[4]:	2510	1000
X[5]:	2530	1003
X[6]:	−2730	0

Note that the last entry contains the negative of the first unused memory address.

(Clearly, this is not the only way to treat a library of subroutines. The proper way to design a library is heavily dependent upon the computer used and the applications to be handled. Several modern computers require an entirely different approach to subroutine libraries.)

The problem in this exercise is to design an algorithm for this subroutine allocation task. The subroutine allocator may transform the tape directory in any way as it prepares its answer, since the tape directory can be read in anew by the subroutine allocator on its next assignment, and the tape directory is not needed by other parts of the loading routine.

27. [*25*] Write a MIX program for the subroutine allocation algorithm of exercise 26.

28. [*40*] The following construction shows how to "solve" a fairly general type of two-person game, including chess, nim, and many simpler games: Consider a finite set of nodes, each of which represents a possible "position" in the game. For each position there are zero or more "moves" which transform that position into some other position. We say that position x is a predecessor of position y (and y is a successor of x) if there is a move from x to y. Certain positions which have no successors are classified as "won" or "lost" positions. The player to move in position x is the opponent of the player to move in the successors of position x.

Given such a configuration of positions, we can compute the complete set of "won" positions (those in which it is possible for the player to force a victory) and the complete set of "lost" positions (those in which the player must lose against an expert opponent) by repeatedly doing the following operation until it yields no change: mark a position "lost" if all its successors are marked "won"; mark a position "won" if at least one of its successors is marked "lost."

After this operation has been repeated as many times as possible, there may be some positions that have not been marked at all; a player in such a position cannot force a victory, nor can he be compelled to lose.

This procedure for obtaining the complete set of "won" and "lost" positions can be adapted to an efficient algorithm for computers that closely resembles Algorithm T. We may keep with each position a count of the number of its successors that have not been marked "won," and a list of all its predecessors.

The problem in this exercise is to work out the details of the algorithm that has just been so vaguely described, and to apply it to some interesting games that do not involve too many possible positions [like the "military game": *Sci. Am.* (October, 1963), 124, or E. Lucas, *Récréations Mathématiques*, 3 (Paris, 1893) 105–116].

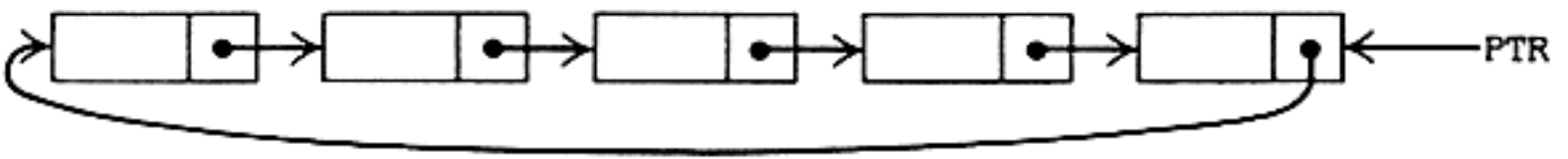
▶ **29.** [*21*] (a) Give an algorithm to "erase" an entire list like (1), i.e., to put all of its nodes on the AVAIL stack, given only the value of FIRST. The algorithm should operate as fast as possible. (b) Repeat part (a) for a list like (12), given the values of F and R.

2.2.4. Circular Lists

A slight change in the manner of linking furnishes us with an important alternative to the methods of the preceding section.

A *circularly-linked list* (briefly: a circular list) has the property that its last node links back to the first instead of to Λ. It is then possible to access all of the list starting at any given point; we also achieve an extra degree of symmetry, and if we choose we need not think of the list as having a "last" or "first" node.

The following situation is typical:



(1)

Assume that the nodes have two fields, INFO and LINK, as in the preceding section. There is a link variable PTR which points to the rightmost node of the list, and LINK(PTR) is the address of the leftmost node. The following primitive operations are most important:

a) Insert Y at left: P ⇐ AVAIL, INFO(P) ← Y, LINK(P) ← LINK(PTR), LINK(PTR) ← P.
b) Insert Y at right: Insert Y at left, then PTR ← P.
c) Set Y to left node and delete: P ← LINK(PTR), Y ← INFO(P), LINK(PTR) ← LINK(P), AVAIL ⇐ P.

Operation (b) is a little surprising at first glance; the operation PTR ← LINK(PTR) effectively moves the leftmost node to the right in the diagram (1), and this is quite easy to understand if the list is regarded as a circle instead of a straight line with connected ends.

The alert reader will observe that we have made a serious mistake in the above operations (a), (b), (c). What is it? *Answer.* We have forgotten to consider the possibility of an *empty* list. If for example operation (c) is applied five times to the list (1), we will have PTR pointing to a node in the AVAIL list, and this can lead to serious difficulties; for example, imagine applying operation (c) *six* times to (1)! If we take the position that PTR will equal Λ in the case of an empty list, we could remedy the above operations by inserting the additional instructions "if PTR = Λ, then PTR ← LINK(P) ← P; otherwise . . ." after "INFO(P) ← Y" in (a) and (b); preceding (c) by the test "if PTR = Λ, then UNDERFLOW"; and following (c) by "if PTR = P, then PTR ← Λ."

Note that the operations (a), (b), and (c) give us the actions of an output-restricted deque, in the sense of Section 2.2.1. Therefore we find in particular that a circular list can be used as either a stack or a queue. Operations (a) and (c) combined give us a stack; operations (b) and (c) give us a queue. These operations are only slightly less direct than their counterparts in the previous section, where we saw that operations (a), (b), and (c) can be performed on linear lists using two pointers F and R.

Other important operations become efficient with circular lists. For example, it is very convenient to "erase" a list, i.e., to put an entire circular list onto the AVAIL stack at once:

$$\text{If} \quad \texttt{PTR} \neq \Lambda, \quad \text{then} \quad \texttt{AVAIL} \leftrightarrow \texttt{LINK(PTR)}. \tag{2}$$

[Recall that the "↔" operation denotes interchange, i.e., P ← AVAIL, AVAIL ← LINK(PTR), LINK(PTR) ← P.] Operation (2) is clearly valid if PTR points *anywhere* in the circular list.

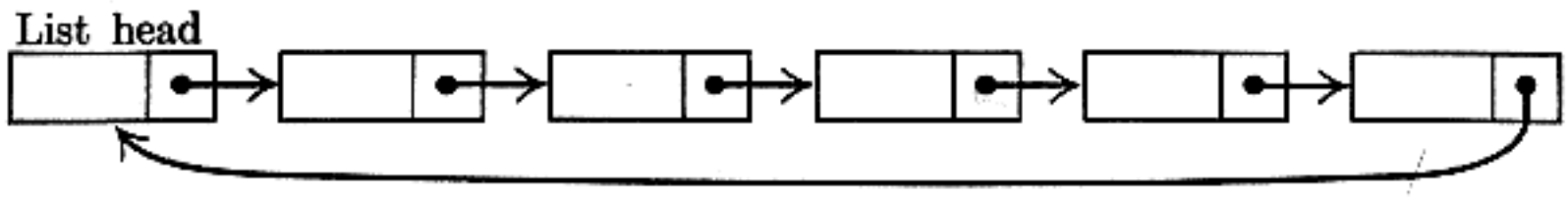
Using a similar technique, if $\texttt{PTR}_1$ and $\texttt{PTR}_2$ point to disjoint circular lists L_1 and L_2, respectively, we can insert the entire list L_2 at the right of L_1:

$$\begin{aligned} &\text{If} \quad \texttt{PTR}_2 \neq \Lambda, \quad \text{then} \\ &(\text{if} \quad \texttt{PTR}_1 \neq \Lambda, \quad \text{then} \quad \texttt{LINK}(\texttt{PTR}_1) \leftrightarrow \texttt{LINK}(\texttt{PTR}_2); \\ &\quad \text{set} \quad \texttt{PTR}_1 \leftarrow \texttt{PTR}_2,\ \texttt{PTR}_2 \leftarrow \Lambda). \end{aligned} \tag{3}$$

Splitting one circular list into two, in various ways, is another simple operation that can be done. These operations correspond to the concatenation and deconcatenation of strings.

Thus we see that a circular list can be used not only to represent inherently circular structures, but also to represent linear structures; a circular list with one pointer to the rear node is essentially equivalent to a straight linear list with two pointers to the front and rear. The natural question to ask, in connection with this observation, is, "How do we find the end of the list, in view of the circular symmetry?" There is no Λ link to signal the end. The answer is that if we are performing some operations while moving through the list from one node to the next, we should stop when we get back to our starting place (assuming, of course, that our starting place is still present in the list).

An alternative solution to the problem just posed is to put a special, recognizable node into each circular list, as a convenient stopping place. This special node is called the *list head*, and in applications we often find it is quite convenient to insist that every circular list have exactly one node which is its list head. One advantage is that the circular list will then never be empty. The diagram (1) now becomes



(4)

Instead of a pointer to the right end of the list, references to lists like (4) are usually made via the list head, which is often in a fixed memory location. In this case, we sacrifice operation (b) stated above.

Diagram (4) may be compared with 2.2.3–(1) at the beginning of the previous section, in which the link associated with "item 5" now points to `LOC(FIRST)` instead of to Λ, and `FIRST` is now thought of as a link within a node, `NODE(LOC(FIRST))`. The principal difference between (4) and 2.2.3–(1) is that with (4) it is possible (though not necessarily efficient) to get to any point of the list from any other point.

As an example of the use of circular lists, we will discuss *arithmetic on polynomials* in the variables x, y, and z, with integer coefficients. There are many problems in which a scientist wants to manipulate polynomials instead of just numbers; we are thinking of operations like the multiplication of

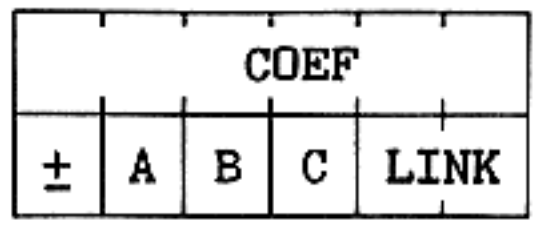
$$(x^4 + 2x^3y + 3x^2y^2 + 4xy^3 + 5y^4) \qquad \text{by} \qquad (x^2 - 2xy + y^2)$$

to get

$$(x^6 - 6xy^5 + 5y^6).$$

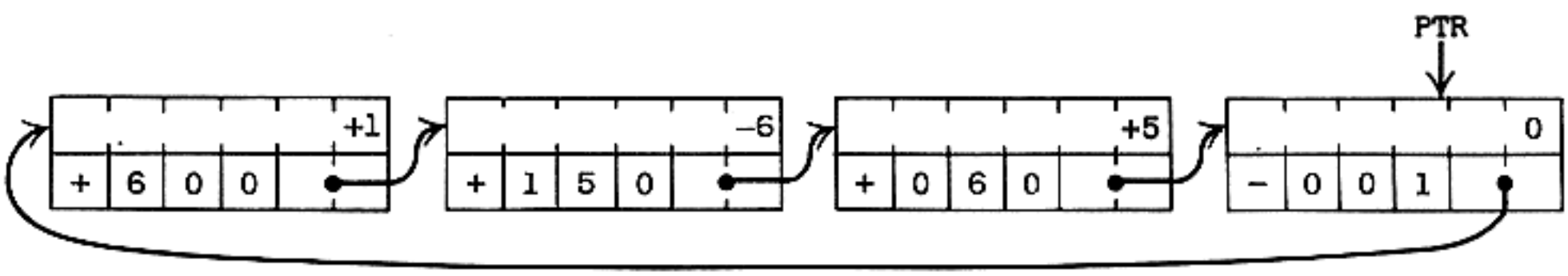
Linked allocation is a natural tool for this purpose, since polynomials can grow to unpredictable sizes and we may want to represent many polynomials in memory at the same time.

We will consider here the two operations of addition and multiplication. Let us suppose that a polynomial is represented as a list in which each node stands for one nonzero term, and has the two-word form



(5)

Here COEF is the coefficient of the term in $x^A y^B z^C$. We will assume that the coefficients and exponents will always lie in the range allowed by this format, and that it is not necessary to check this condition during our calculations. The notation ABC will be used to stand for the ± A B C fields of the node (5), treated as a single unit. The sign of ABC, i.e., the sign of the second word in (5), will always be plus, except that there is a *special node* at the end of every polynomial which has ABC $= -1$ and COEF $= 0$. This special node is a great convenience, analogous to our discussion of a list head above, because it provides a convenient "sentinel" and it avoids the problem of an empty list (corresponding to the polynomial "0"). The nodes of the list always appear in *decreasing order* of the ABC field, if we follow the direction of the links, except that the special node (which has ABC $= -1$) links to the largest value of ABC. For example, the polynomial $x^6 - 6xy^5 + 5y^6$ would be represented thus:



Algorithm A (*Addition of polynomials*). This algorithm adds polynomial(P) to polynomial(Q), assuming that P and Q are pointer variables pointing to polynomials having the form above. The list P will be unchanged, the list Q will retain the sum. Pointer variables P and Q return to their starting points at the conclusion of this algorithm; auxiliary pointer variables Q1 and Q2 are also used.

A1. [Initialize.] Set P ← LINK(P), Q1 ← Q, Q ← LINK(Q). (Now both P and Q point to the leading term of the polynomial. Throughout most of this algorithm the variable Q1 will be "one step behind" Q, in the sense that Q = LINK(Q1).)

A2. [ABC(P):ABC(Q).] If ABC(P) < ABC(Q), set Q1 ← Q and Q ← LINK(Q) and repeat this step. If ABC(P) = ABC(Q), go to step A3. If ABC(P) > ABC(Q), go to step A5.

A3. [Add coefficients.] (We have found terms with equal exponents.) If ABC(P) < 0, the algorithm terminates. Otherwise set COEF(Q) ← COEF(Q) + COEF(P). Now if COEF(Q) = 0, go to A4; otherwise, set Q1 ← Q, P ← LINK(P), Q ← LINK(Q), and go to A2.

A4. [Delete zero term.] Set Q2 ← Q, LINK(Q1) ← Q ← LINK(Q), and AVAIL ⇐ Q2. (A zero term created in step A3 has been removed from polynomial (Q).) Set P ← LINK(P) and go to A2.

A5. [Insert new term.] (Polynomial(P) contains a term that is not present in polynomial(Q), so we insert it in polynomial(Q).) Set Q2 ⇐ AVAIL, COEF(Q2) ← COEF(P), ABC(Q2) ← ABC(P), LINK(Q2) ← Q, LINK(Q1) ← Q2, Q1 ← Q2, P ← LINK(P), and return to step A2. ▮

One of the most noteworthy features of Algorithm A is the manner in which the pointer variable Q1 follows the pointer Q around the list. This is very typical of list processing algorithms, and we will see a dozen more algorithms with the same characteristic. Can the reader see why this idea was used in Algorithm A?

A reader who has little prior experience with linked lists will find it very instructive to study Algorithm A carefully; as a test case, try adding $x + y + z$ to $x^2 - 2y - z$.

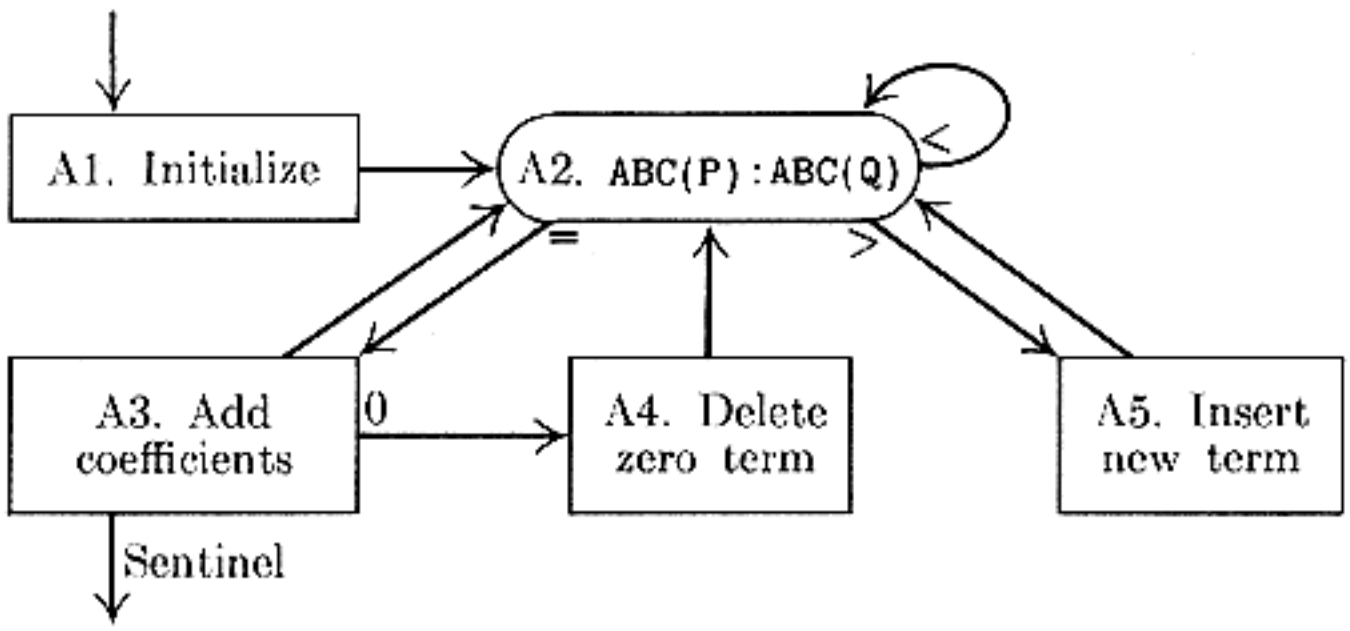


Fig. 10. Addition of polynomials.

Given Algorithm A, the multiplication operation is surprisingly easy:

Algorithm M (*Multiplication of polynomials*). This algorithm, analogous to Algorithm A, replaces polynomial(Q) by polynomial(Q) + polynomial(M) × polynomial(P).

M1. [Next multiplier.] Set M ← LINK(M). If ABC(M) < 0, the algorithm terminates.

M2. [Multiply cycle.] Perform Algorithm A, except wherever the notation "ABC(P)" appears in that algorithm, replace it by "if ABC(P) < 0 then −1, otherwise, ABC(P) + ABC(M)"; wherever "COEF(P)" appears in that algorithm, replace it by "COEF(P) × COEF(M)". Then go back to step M1. ▮

The programming of Algorithm A in MIX language shows again the ease with which linked lists are manipulated in a computer. In the following code we assume that OVERFLOW is a subroutine which either terminates the program (due to lack of memory space) or finds further available space and exits to (rJ) − 2.

Program A (*Addition of polynomials*). This is a subroutine written so that it can be used in conjunction with a multiplication subroutine (see exercise 15).

Calling sequence:	JMP ADD
Entry conditions:	rI1 = P, rI2 = Q.
Exit conditions:	polynomial(Q) has been replaced by polynomial(Q) + polynomial (P); rI1 and rI2 are unchanged; all other registers have undefined contents.

In the coding below, P ≡ rI1, Q ≡ rI2, Q1 ≡ rI3, and Q2 ≡ rI6, in the notation of Algorithm A.

```
01   LINK  EQU   4:5                 Definition of LINK field
02   ABC   EQU   0:3                 Definition of ABC field
03   ADD   STJ   3F          1       Entrance to subroutine
04         ENT3  0,2         1       A1. Initialize. Set Q1 ← Q.
05   OH    LD1   1,1(LINK)   1+p     P ← LINK(P).
06   SW1   LDA   1,1         1+p     rA(0:3) ← ABC(P).
07   1H    LD2   1,3(LINK)   x       Q ← LINK(Q1).
08   2H    CMPA  1,2(ABC)    x       A2. ABC(P):ABC(Q).
09         JE    3F          x       If equal, go to A3.
10         JG    5F          p'+q'   If greater, go to A5.
11         ENT3  0,2         q'      If less, set Q1 ← Q.
12         JMP   1B          q'      Repeat.
13   3H    JAN   *           m+1     A3. Add coefficients.
14   SW2   LDA   0,1         m       COEF(P)
15         ADD   0,2         m         + COEF(Q)
16         STA   0,2         m           → COEF(Q).
17         JANZ  6F          m       Is result zero?
18         ENT6  0,2         m'      A4. Delete zero term. Q2 ← Q.
19         LD2   1,2(LINK)   m'      Q ← LINK(Q).
20         LDX   AVAIL       m'      ⎫
21         STX   1,6(LINK)   m'      ⎬AVAIL ⇐ Q2.
22         ST6   AVAIL       m'      ⎭
23         ST2   1,3(LINK)   m'      LINK(Q1) ← Q.
24         JMP   OB          m'      Go to advance P.
25   6H    ENT3  0,2         m''     Set Q1 ← Q.
26         JMP   OB          m''     Go to advance P, Q.
27   5H    LD6   AVAIL       p'      ⎫ A5. Insert new term.
28         J6Z   OVERFLOW    p'      ⎬Q2 ⇐ AVAIL.
29         LDX   1,6(LINK)   p'      ⎪
30         STX   AVAIL       p'      ⎭
31         STA   1,6         p'      ABC(Q2) ← ABC(P).
32   SW3   LDA   0,1         p'      rA ← COEF(P).
33         STA   0,6         p'      COEF(Q2) ← rA.
34         ST2   1,6(LINK)   p'      LINK(Q2) ← Q.
35         ST6   1,3(LINK)   p'      LINK(Q1) ← Q2.
36         ENT3  0,6         p'      Q1 ← Q2.
37         JMP   OB          p'      Go to advance P.  ▮
```

Note that Algorithm A traverses each of the two lists just once; it is not necessary to loop around several times. Using Kirchhoff's law, we find that an analysis of the execution presents no difficulties; the execution time depends on the quantities

m' = number of matching terms which cancel with each other;
m'' = number of matching terms which do not cancel;
p' = number of unmatched terms in polynomial(P);
q' = number of unmatched terms in polynomial(Q).

The analysis given with Program A uses the abbreviations

$$m = m' + m'', \qquad p = m + p',$$
$$q = m + q', \qquad x = 1 + m + p' + q';$$

the running time for MIX is $(29m' + 19m'' + 29p' + 8q' + 13)u$. The total number of nodes in the storage pool needed during the execution of the algorithm is at least $2 + p + q$, and at most $2 + p + q + p'$.

EXERCISES

1. [*21*] The text suggests at the beginning of this section that an empty circular list could be represented by PTR = Λ. It might be more consistent with the philosophy of circular lists to have PTR = LOC(PTR) indicate an empty list. Does this convention facilitate operations (a), (b), or (c) described at the beginning of this section?

2. [*20*] Draw "before and after" diagrams illustrating the effect of the concatenation operation (3), assuming that PTR_1 and PTR_2 are $\neq \Lambda$.

▸ 3. [*20*] What does operation (3) do if PTR_1 and PTR_2 are both pointing to nodes in the *same* circular list?

4. [*21*] Give insertion and deletion operations corresponding to the representation (4), which give the effect of a *stack*.

▸ 5. [*21*] Design an algorithm which takes a circular list such as (1) and reverses the direction of all the arrows.

6. [*18*] Give diagrams of the list representation for the polynomials (a) "$xz - 3$"; (b) "0".

7. [*10*] Why is it useful to assume that the ABC fields of a polynomial list appear in decreasing order?

8. [*10*] Why is it useful to have Q1 trailing one step behind Q in Algorithm A?

▸ 9. [*23*] Would Algorithm A work properly if P = Q (i.e., both pointer variables point at the same polynomial)? Would Algorithm M work properly if P = M, if P = Q, or if M = Q?

▸ 10. [*20*] The algorithms in this section assume that we are using three variables x, y, and z in the polynomials, and their exponents individually never exceed b (where b is the byte size in MIX's case). Suppose that we want instead to do addition and multiplication of polynomials in only one variable, x, and to let its exponent take on values up to b^3. What changes should be made to Algorithms A and M?

11. [*24*] (The purpose of this exercise and many of those following is to create a "package" of subroutines useful for polynomial arithmetic, in conjunction with Program A.) Since Algorithms A and M change the value of polynomial (Q), it is sometimes desirable to have a subroutine that makes a copy of a given polynomial. Write a MIX subroutine with the following specifications:

Calling sequence:	JMP COPY
Entry conditions:	rI1 = P
Exit conditions:	rI2 points to a newly created polynomial equal to polynomial(P); rI1 is unchanged; other registers are undefined.

12. [*21*] Compare the running time of the program in exercise 11 with that of Algorithm A when polynomial(Q) = "0".

13. [*20*] Write a MIX subroutine with the following specifications:

Calling sequence:	JMP ERASE
Entry conditions:	rI1 = P
Exit conditions:	polynomial (P) has been added to the AVAIL list; all register contents are undefined.

[*Note:* This subroutine can be used in conjunction with the subroutine of exercise 11 in the sequence "LD1 Q; JMP ERASE; LD1 P; JMP COPY; ST2 Q" to achieve the effect "polynomial (Q) ← polynomial (P)".]

14. [*22*] Write a MIX subroutine with the following specifications:

Calling sequence:	JMP ZERO
Entry conditions:	None
Exit conditions:	rI2 points to a newly created polynomial equal to "0"; other register contents are undefined.

15. [*24*] Write a MIX subroutine to perform Algorithm M, having the following specifications:

Calling sequence:	JMP MULT
Entry conditions:	rI1 = P, rI2 = Q, rI4 = M.
Exit conditions:	polynomial (Q) ← polynomial (Q) + polynomial (M) × polynomial (P); rI1, rI2, rI4 are unchanged; other registers undefined.

(*Note:* Use Program A as a subroutine, changing the settings of SW1, SW2, and SW3.)

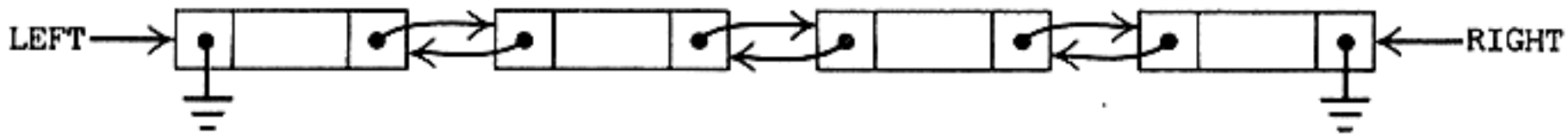
16. [*M22*] Estimate the running time of the subroutine in exercise 15 in terms of some relevant parameters.

▶ **17.** [*22*] What advantage is there in representing polynomials with a circular list as in this section, instead of with a straight linear linked list terminated by Λ as in the previous section?

18. [*25*] Devise a way to represent circular lists inside a computer in such a way that the list can be traversed efficiently in both directions, yet only one link field is used per node. [*Hint:* If we are given two pointers, to two successive nodes x_{i-1} and x_i, it should be possible to locate both x_{i+1} and x_{i-2}.]

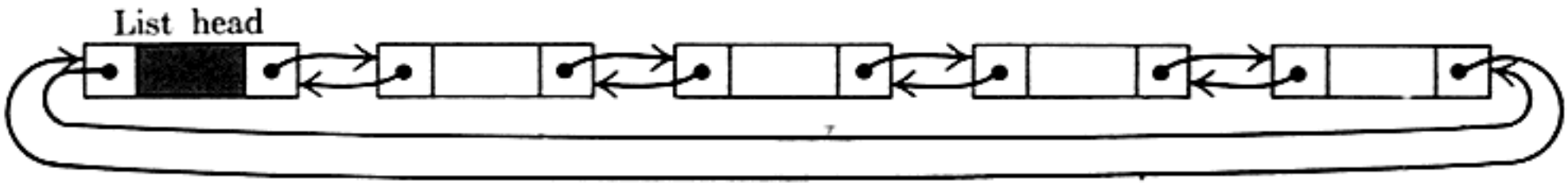
2.2.5. Doubly Linked Lists

For even greater flexibility in the manipulation of linear lists, we can include two links in each node, pointing to the items on either side of that node:



(1)

Here LEFT and RIGHT are pointer variables to the left and right of the list. Each node of the list includes two links, called, for example, LLINK and RLINK. The operations of a general deque are readily performed with the above representation; see exercise 1. However, manipulations of doubly linked lists almost always become much easier if a *list head* node is part of each list, as described in the preceding section. When a list head is present, we have the following typical diagram of a doubly linked list:



(2)

The RLINK and LLINK fields of the list head take the place of LEFT and RIGHT in (1). There is complete symmetry between left and right; the list head could equally well have been shown at the right of (2). If the list is empty, both link fields of the list head point to the head itself.

The list representation (2) clearly satisfies the condition

$$\texttt{RLINK(LLINK(X))} = \texttt{LLINK(RLINK(X))} = \texttt{X} \tag{3}$$

if X is the location of any node in the list (including the head). This fact is the principal reason representation (2) is preferable to (1).

A doubly linked list usually takes more memory space than a singly linked one does (although sometimes there is already room for another link in a node that doesn't fill a complete computer word). The additional operations that can now be performed efficiently are often more than ample compensation for this extra space requirement. Besides the obvious advantage of being able to go back and forth at will when examining a doubly linked list, one of the principal new abilities is the fact that *we can delete* NODE(X) *from the list it is in, given only the value of* X. This deletion operation is easy to derive from a "before and after" diagram (Fig. 11) and it is very simple:

$$\texttt{RLINK(LLINK(X))} \leftarrow \texttt{RLINK(X)}, \qquad \texttt{LLINK(RLINK(X))} \leftarrow \texttt{LLINK(X)},$$
$$\texttt{AVAIL} \Leftarrow \texttt{X}. \tag{4}$$

In a list which has only one-way links, we cannot delete NODE(X) without knowing which node precedes it in the chain, since the preceding node needs to

have its link altered when NODE(X) is deleted. In all the algorithms considered in Sections 2.2.3 and 2.2.4 this additional knowledge was present whenever a node was to be deleted; see, in particular, Algorithm 2.2.4A, where we had pointer Q1 following pointer Q for just this purpose. But we will meet several algorithms which require removing random nodes from the middle of a list, and doubly linked lists are frequently used just for this reason. (We should point out that in a circular list it is possible to delete NODE(X), given X, if we go around the entire circle to find the predecessor of X. But this operation is clearly inefficient when the list is long, so it is rarely an acceptable substitute for doubly linking the list.)

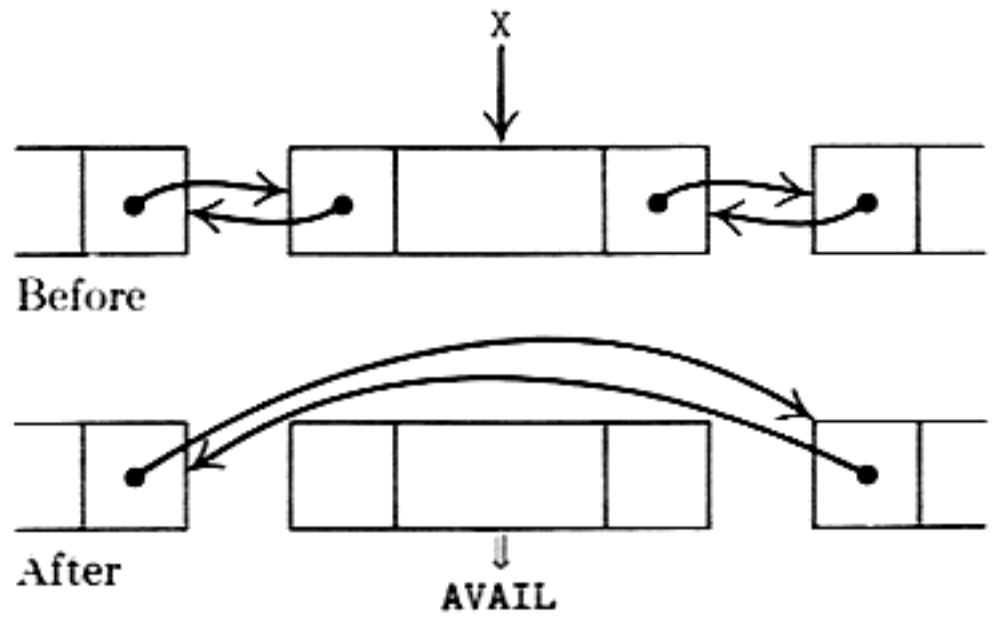


Fig. 11. Deletion from a doubly linked list.

Similarly, a doubly linked list permits the easy insertion of a node adjacent to NODE(X) at either the left or the right. The steps

$$\begin{aligned} &\texttt{P} \Leftarrow \texttt{AVAIL}, \quad \texttt{LLINK(P)} \leftarrow \texttt{X}, \quad \texttt{RLINK(P)} \leftarrow \texttt{RLINK(X)}, \\ &\qquad \texttt{LLINK(RLINK(X))} \leftarrow \texttt{P}, \quad \texttt{RLINK(X)} \leftarrow \texttt{P} \end{aligned} \tag{5}$$

do such an insertion to the right of NODE(X); and by interchanging left and right we get the corresponding algorithm for insertion to the left. Operation (5) changes the settings of five links, so it is a little slower than an insertion operation in a one-way list where only three links need to be changed.

As an example of the use of doubly linked lists, we will now consider the writing of a *discrete simulation* program. "Discrete simulation" means the simulation of a system in which all changes in the state of the system may be assumed to happen at certain discrete instants of time. The "system" being simulated usually is a set of individual activities which are largely independent although they interact with each other; examples are customers at a store, ships in a harbor, people in a corporation. In a discrete simulation, we proceed by doing whatever is to be done at a certain instant of simulated time, then advance the simulated clock to the next time when some action is to occur.

By contrast, a "continuous simulation" would be simulation of activities which are under continuous changes, such as traffic moving on a highway, spaceships traveling to other planets, etc. Continuous simulation can often be satisfactorily approximated by discrete simulation with very small time intervals

between steps; however, in such a case we usually have "synchronous" discrete simulation, in which many parts of the system are slightly altered at each discrete time interval, and that generally calls for a somewhat different type of program organization than the kind considered here.

The program developed below simulates the elevator system in the Mathematics building of the California Institute of Technology. The results of such a simulation will perhaps be of use only to people who make reasonably frequent visits to Caltech; and even for those who do, it may be simpler just to try using the elevator several times instead of writing a computer program. But, as is usual with simulation studies, the methods we will use to achieve the simulation are of much more interest than the answers given by the program. The methods to be discussed below illustrate typical implementation techniques used with discrete simulation programs.

The Mathematics building has five floors: sub-basement, basement, first, second, and third. There is a single elevator, which has automatic controls and can stop at each floor. For convenience we will renumber the floors 0, 1, 2, 3, and 4.

On each floor there are two call buttons, one for `UP` and one for `DOWN`. (Actually floor 0 has only `UP` and floor 4 has only `DOWN`, but we may ignore that anomaly since the excess buttons will never be used.) Corresponding to these buttons, there are ten variables `CALLUP[`j`]` and `CALLDOWN[`j`]`, $0 \le j \le 4$. There are also variables `CALLCAR[`j`]`, $0 \le j \le 4$, representing buttons within the elevator car which direct it to a destination floor. When a man presses a button he sets the appropriate variable to 1; the elevator clears the variable to 0 after the request has been fulfilled.

The above describes the elevator from a man's point of view; the situation is more interesting as viewed by the elevator. The elevator is in one of three states: `GOINGUP`, `GOINGDOWN`, or `NEUTRAL`. (The current state is indicated to passengers by lighted arrows inside the elevator.) If it is in `NEUTRAL` state and not on floor 2, the machine will close its doors and (if no command is given by the time its doors are shut) it will change to `GOINGUP` or `GOINGDOWN`, heading for floor 2. (This is the "home floor," since most passengers get in there.) On floor 2 in `NEUTRAL` state, the doors will eventually close and the machine will wait silently for another command. The first command received for another floor sets the machine `GOINGUP` or `GOINGDOWN` as appropriate; it stays in this state until there are no commands waiting in the same direction, and then it switches direction or switches to `NEUTRAL` just before opening the doors, depending on what other commands are in the `CALL` variables. The elevator takes a certain amount of time to open and close its doors, to accelerate and decelerate, and to get from one floor to another. All these quantities are indicated in the algorithm below, which is much more precise than this informal description can be. The algorithm we will now study may not reflect the elevator's true principles of operation, but it is believed to be the simplest set of rules which explain all the phenomena observed during several hours of experimentation by the author during the writing of this section.

The elevator system is simulated by using two coroutines, one for the passengers and one for the elevator; these routines specify all the actions to be performed, as well as various time delays which are to be used in the simulation. In the following description, the variable TIME represents the current value of the simulated time clock. All units of time are given in *tenths of seconds*. There are also several other variables:

FLOOR, the current position of the elevator;
D1, a variable which is zero except during the time people are getting in or out of the elevator;
D2, a variable which becomes zero if the elevator has sat on one floor without moving for 30 sec or more;
D3, a variable which is zero except during the time the doors are open but nobody is getting in or out of the elevator;
STATE, the current state of the elevator (GOINGUP, GOINGDOWN, or NEUTRAL).

Initially, FLOOR $= 2$, D1 $=$ D2 $=$ D3 $= 0$, and STATE $=$ NEUTRAL.

Coroutine M (*Men*). When each man enters the system, he begins to perform the actions specified below, starting at step M1.

M1. [Enter, prepare for successor.] The following quantities are determined in some manner that will not be specified here:

IN, the floor on which the new man has entered the system;
OUT, the floor to which he wants to go (OUT $\neq$ IN);
INTERTIME, the amount of time before the next man will enter the system;
GIVEUPTIME, the amount of time this man will wait for the elevator before he gives up and decides to walk.

After these quantities have been computed, the simulation program sets things up so that another man enters the system at TIME + INTERTIME.

M2. [Signal and wait.] (The purpose of this step is to call for the elevator; some special cases arise if the elevator is already on the right floor.) If FLOOR $=$ IN and if the elevator's next action is step E6 below (i.e., if the elevator doors are now closing), send the elevator immediately to its step E3 and cancel its activity E6. (This means the doors will open again before the elevator moves.) If FLOOR $=$ IN and if D3 $\neq 0$, set D3 $\leftarrow 0$, set D1 $\neq 0$, and start up the elevator's activity E4 again. (This means the elevator doors are open on this floor, but everyone else has already gotten on or off; elevator step E4 is a sequencing step that grants people permission to enter the elevator according to normal laws of courtesy, and so restarting E4 gives this man a chance to get in before the doors close.) In all other cases, the man sets CALLUP[IN] $\leftarrow 1$ or CALLDOWN[IN] $\leftarrow 1$, according as OUT $>$ IN or OUT $<$ IN; and if D2 $= 0$ or the elevator is in its "dormant" position E1, the DECISION subroutine specified below is performed. (The DECISION subroutine is used to take the elevator out of NEUTRAL state at certain critical times.)

M3. [Enter queue.] Insert this man at the rear of QUEUE[IN], which is a linear list representing the people waiting on this floor. Now this man ceases activity; he will perform action M4 after GIVEUPTIME units of time, unless step E4 of the elevator routine below sends him to M5 earlier.

M4. [Give up.] If FLOOR $\neq$ IN or D1 $= 0$, delete this man from QUEUE[IN] and from the simulated system. (He has decided the elevator is too slow.) If FLOOR $=$ IN and D1 $\neq 0$, he stays and waits (since he knows he will soon be able to get in).

M5. [Get in.] Delete this man from QUEUE[IN], and insert him in ELEVATOR, which is a stack-like list representing the people now in the elevator. Set CALLCAR[OUT] $\leftarrow 1$.

Now if STATE $=$ NEUTRAL, set STATE $\leftarrow$ GOINGUP or GOINGDOWN as appropriate, and set the elevator's activity E5 to be executed after 25 units of time. (This is a special feature of the elevator, that the doors close faster when a man gets in the car and the elevator is in NEUTRAL state. The 25 units of time gives step E4 the opportunity to make sure that D1 is properly set up by the time step E5, the door-closing action, occurs.)

Now the man waits until he is sent to step M6, by step E4 below, when the elevator has reached his floor.

M6. [Get out.] Delete this man from ELEVATOR and from the simulated system. ∎

Coroutine E (*Elevator*). This coroutine represents the actions of the elevator, and also in step E4 the control of when people get in and out.

E1. [Wait for call.] (At this point the elevator is sitting at floor 2 with the doors closed waiting for something to happen.) If someone presses a button, the DECISION subroutine will take us to step E3 or E6. Meanwhile, wait.

E2. [Change of state?] If STATE $=$ GOINGUP and CALLUP[j] $=$ CALLDOWN[j] $=$ CALLCAR[j] $= 0$ for all $j >$ FLOOR, then set STATE $\leftarrow$ NEUTRAL or STATE $\leftarrow$ GOINGDOWN, according as CALLCAR[j] $= 0$ for all $j <$ FLOOR or not, and set all CALL variables for the current floor to zero. If STATE $=$ GOINGDOWN, do similar actions with directions reversed.

E3. [Open door.] Set D1 and D2 to any nonzero values. Set elevator activity E9 to start up independently after 300 units of time. (This activity may be canceled in step E6 below before it occurs.) Also set elevator activity E5 to start up independently after 76 units of time. Then wait 20 units of time (to simulate opening of the doors) and go to E4.

E4. [Let people out, in.] If anyone in the ELEVATOR list has OUT $=$ FLOOR, send the man of this type who has most recently entered immediately to step M6 of his program, wait 25 units, and repeat step E4. If no such men exist, but QUEUE[FLOOR] is not empty, send the front man of that queue immediately to step M5 instead of M4 in his program, wait 25 units, and repeat step E4. But if QUEUE[FLOOR] is empty, set D1 $\leftarrow 0$, D3 $\neq 0$, and wait for

some other activity to initiate further action. (Step E5 will send us to E6, or step M2 will restart E4.)

E5. [Close door.] If D1 $\neq 0$, wait 40 units and repeat this step (the doors flutter a little but spring open again since someone is still getting out or in). Otherwise set D3 $\leftarrow 0$ and set the elevator to start at step E6 after 20 units of time. (This simulates closing the doors after people have finished getting in or out; but if a new man enters on this floor while the doors are closing, they will open again as stated in step M2.)

E6. [Prepare to move.] Set CALLCAR[FLOOR] to zero; also set CALLUP[FLOOR] to zero if STATE $\neq$ GOINGDOWN, and also set CALLDOWN[FLOOR] to zero if STATE $\neq$ GOINGUP. (*Note:* If STATE = GOINGUP, the elevator does not clear out CALLDOWN, since it assumes people who are going down will not have entered; but see exercise 6.) Now perform the DECISION subroutine.

If STATE = NEUTRAL even after the DECISION subroutine has acted, go to E1. Otherwise, if D2 $\neq 0$, cancel the elevator activity E9. Finally, if STATE = GOINGUP, wait 15 units of time (for the elevator to build up speed) and go to E7; if STATE = GOINGDOWN, wait 15 units and go to E8.

E7. [Go up a floor.] Set FLOOR $\leftarrow$ FLOOR + 1 and wait 51 units of time. If now CALLCAR[FLOOR] = 1 or CALLUP[FLOOR] = 1, or if ((FLOOR = 2 or CALLDOWN[FLOOR] = 1) and CALLUP[j] = CALLDOWN[j] = CALLCAR[j] = 0 for all $j >$ FLOOR), wait 14 units (for deceleration) and go to E2. Otherwise, repeat this step.

E8. [Go down a floor.] This step is like E7 with directions reversed, and also the times 51 and 14 are changed to 61 and 23, respectively. (It takes the elevator longer to go down than up.)

E9. [Set inaction indicator.] Set D2 $\leftarrow 0$ and perform the DECISION subroutine. (This independent action is initiated in step E3 but it is almost always canceled in step E6. See exercise 4.) ▮

Subroutine D (DECISION *subroutine*). This subroutine is performed at certain critical times, as specified in the coroutines above, when a decision about the elevator's next direction is to be made.

D1. [Decision necessary?] If STATE $\neq$ NEUTRAL, exit from this subroutine.

D2. [Should door open?] If the elevator is positioned at E1 and if CALLUP[2], CALLCAR[2], or CALLDOWN[2] is not zero, cause the elevator to start its activity E3 after 20 units of time.

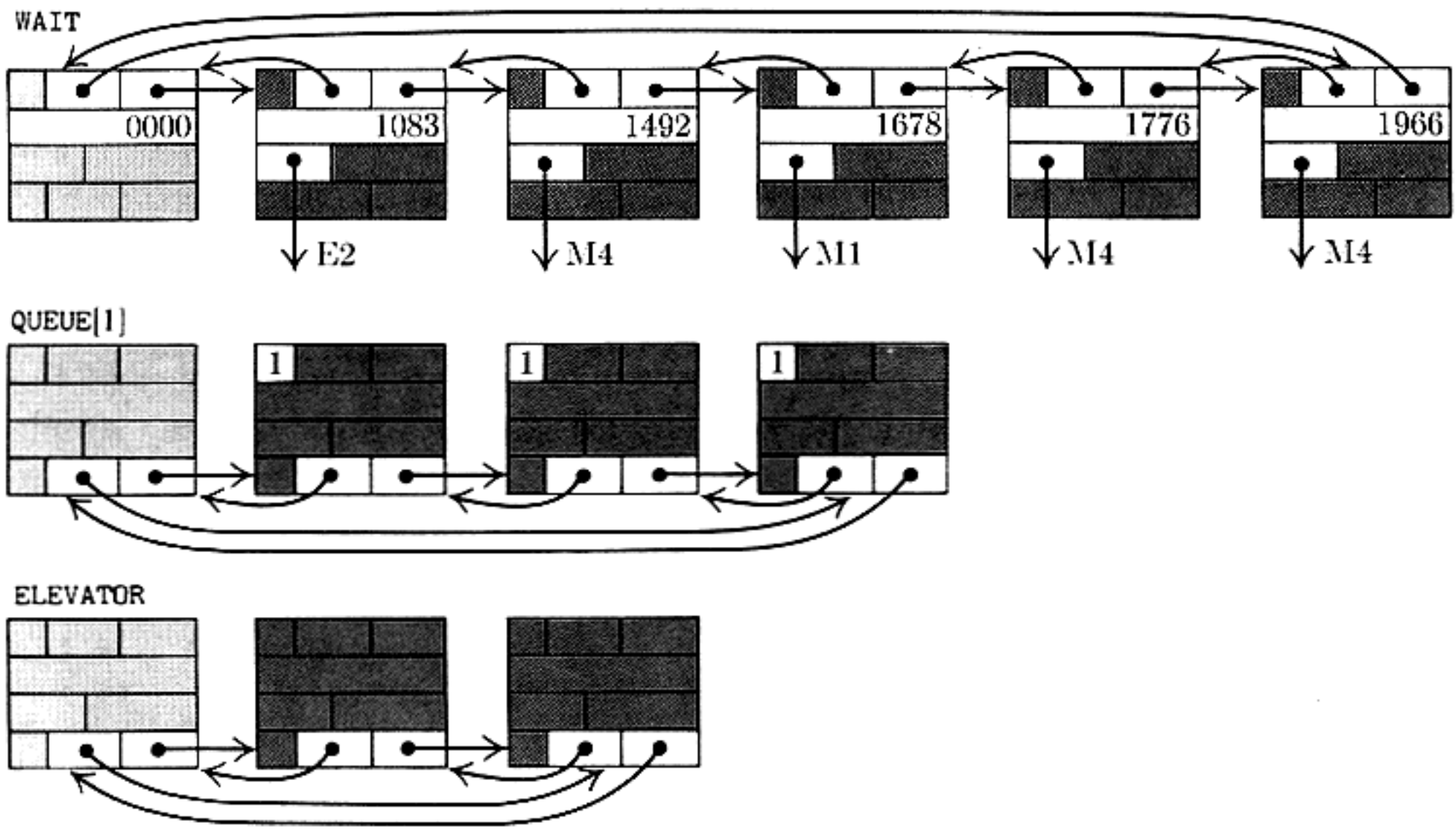


Fig. 12. Some lists used in the elevator simulation program. (List heads appear at the left.)

IN and OUT field are relevant when node (6) represents a man, but they are not relevant for nodes that represent elevator actions. The third word of the node is actually a MIX "JMP" instruction.

Figure 12 shows typical contents of the WAIT list, ELEVATOR list, and one of the QUEUE lists; each node in the QUEUE list is simultaneously in the WAIT list with NEXTINST = M4, but this has not been indicated in the figure, since the complexity of the linking would obscure the basic idea.

Now let us consider the program itself. The program is quite long, although (as with all long programs) it divides into small parts each of which is quite simple in itself. First comes a number of lines of code that just serve to define the initial contents of the tables. There are several points of interest here: We have list heads for the WAIT list (lines 10–11), the QUEUE lists (lines 26–31), and the ELEVATOR list (lines 32–33). Each of these is a node of the form (6), but with unimportant words deleted; the WAIT list head contains only the first two words of a node, and the QUEUE and ELEVATOR list heads require only the last word of a node. We also have four nodes which are always present in the system (lines 12–23): MAN1, a node which is always positioned at step M1 ready to enter a new man into the system; ELEV1, a node which governs the main actions of the elevator at steps E1, E2, E3, E4, E6, E7, and E8; and ELEV2 and ELEV3, nodes which are used for the elevator actions E5 and E9, which take place independently of other elevator actions with respect to simulated time. Each of these four nodes contains only three words, since they never appear in the QUEUE or

ELEVATOR lists. The nodes representing each actual man in the system will appear in a storage pool following the main program.

```
01    * THE ELEVATOR SIMULATION
02    IN        EQU   1:1                          Definition of fields
03    LLINK1    EQU   2:3                              within nodes
04    RLINK1    EQU   4:5
05    NEXTINST  EQU   0:2
06    OUT       EQU   1:1
07    LLINK2    EQU   2:3
08    RLINK2    EQU   4:5
09    * FIXED-SIZE TABLES AND LIST HEADS
10    WAIT      CON   *+2(LLINK1),*+2(RLINK1)      List head for WAIT list
11              CON   0                                NEXTTIME = 0 always
12    MAN1      CON   *-2(LLINK1),*-2(RLINK1)      This node represents action
13              CON   0                                M1 and it is initially the
14              JMP   M1                               sole entry in the WAIT list.
15    ELEV1     CON   0                            This node represents the
16              CON   0                                elevator actions, except
17              JMP   E1                               for E5 and E9.
18    ELEV2     CON   0                            This node represents the
19              CON   0                                independent elevator
20              JMP   E5                               action at E5.
21    ELEV3     CON   0                            This node represents the
22              CON   0                                independent elevator
23              JMP   E9                               action at E9.
24    AVAIL     CON   0                            Link to available nodes
25    TIME      CON   0                            Current simulated time
26    QUEUE     EQU   *-3
27              CON   *-3(LLINK2),*-3(RLINK2)      List head for QUEUE[0]
28              CON   *-3(LLINK2),*-3(RLINK2)      List head for QUEUE[1]
29              CON   *-3(LLINK2),*-3(RLINK2)      All queues initially
30              CON   *-3(LLINK2),*-3(RLINK2)          are empty
31              CON   *-3(LLINK2),*-3(RLINK2)      List head for QUEUE[4]
32    ELEVATOR  EQU   *-3
33              CON   *-3(LLINK2),*-3(RLINK2)      List head for ELEVATOR
34              CON   0                            }
35              CON   0                            } "Padding" for CALL table
36              CON   0                            }     (see lines 183–186)
37              CON   0                            }
38    CALL      CON   0                            CALLUP[0], CALLCAR[0], CALLDOWN[0]
39              CON   0                            CALLUP[1], CALLCAR[1], CALLDOWN[1]
40              CON   0                            CALLUP[2], CALLCAR[2], CALLDOWN[2]
41              CON   0                            CALLUP[3], CALLCAR[3], CALLDOWN[3]
42              CON   0                            CALLUP[4], CALLCAR[4], CALLDOWN[4]
43              CON   0                            }
44              CON   0                            } "Padding" for CALL table
45              CON   0                            }     (see lines 178–181)
46              CON   0                            }
47    D1        CON   0                            Indicates door open, activity
48    D2        CON   0                            Indicates prolonged standstill
49    D3        CON   0                            Indicates door open, inactivity ▮
```

The next part of the program coding contains basic subroutines and the main control routines for the simulation process. Subroutines INSERT and DELETE perform typical manipulations on doubly linked lists; they put the current node into or take it out of a QUEUE or ELEVATOR list. (In the program, the "current node" C is always represented by index register 6.) There are also subroutines for the WAIT list: Subroutine SORTIN adds the current node to the WAIT list, sorting it into the right place based on its NEXTTIME field. Subroutine IMMED inserts the current node at the front of the WAIT list. Subroutine HOLD puts the current node into the WAIT list, with NEXTTIME equal to the current time plus the amount in register A. Subroutine DELETEW deletes the current node from the WAIT list.

The routine CYCLE is the heart of the simulation control: it decides which activity is to act next (namely, the first element of the WAIT list, which we know is nonempty), and jumps to it. There are two special entrances to CYCLE: CYCLE1 first sets NEXTINST in the current node, and HOLDC is the same with an additional call on the HOLD subroutine. Thus, the effect of the instruction "JMP HOLDC" with amount t in register A is to suspend activity for t units of simulated time and then to return to the following location.

```
50    * SUBROUTINES AND CONTROL ROUTINE
51    INSERT    STJ   9F                 Insert NODE(C) to left of NODE(rI1):
52              LD2   3,1(LLINK2)        rI2 ← LLINK2(rI1).
53              ST2   3,6(LLINK2)        LLINK2(C) ← rI2.
54              ST6   3,1(LLINK2)        LLINK2(rI1) ← C.
55              ST6   3,2(RLINK2)        RLINK2(rI2) ← C.
56              ST1   3,6(RLINK2)        RLINK2(C) ← rI1.
57    9H        JMP   *                  Exit from subroutine.
58    DELETE    STJ   9F                 Delete NODE(C) from its list:
59              LD1   3,6(LLINK2)        P ← LLINK2(C).
60              LD2   3,6(RLINK2)        Q ← RLINK2(C).
61              ST1   3,2(LLINK2)        LLINK2(Q) ← P.
62              ST2   3,1(RLINK2)        RLINK2(P) ← Q.
63    9H        JMP   *                  Exit from subroutine.
64    IMMED     STJ   9F                 Insert NODE(C) first in WAIT list:
65              LDA   TIME
66              STA   1,6                Set NEXTTIME(C) ← TIME.
67              ENT1  WAIT               P ← LOC(WAIT).
68              JMP   2F                 Insert NODE(C) to right of NODE(P).
69    HOLD      ADD   TIME               rA ← TIME + rA.
70    SORTIN    STJ   9F                 Sort NODE(C) into WAIT list:
71              STA   1,6                Set NEXTTIME(C) ← rA.
72              ENT1  WAIT               P ← LOC(WAIT).
73              LD1   0,1(LLINK1)        P ← LLINK1(P).
74              CMPA  1,1                Compare NEXTTIME fields, right to left.
75              JL    *-2                Repeat until NEXTTIME(C) ≥ NEXTTIME(P).
76    2H        LD2   0,1(RLINK1)        Q ← RLINK1(P).
77              ST2   0,6(RLINK1)        RLINK1(C) ← Q.
78              ST1   0,6(LLINK1)        LLINK1(C) ← P.
79              ST6   0,1(RLINK1)        RLINK1(P) ← C.
80              ST6   0,2(LLINK1)        LLINK1(Q) ← C.
81    9H        JMP   *                  Exit from subroutine.
```

```
82    DELETEW   STJ   9F                 Delete NODE(C) from WAIT list:
83              LD1   0,6(LLINK1)        (This is same as lines 58–63
84              LD2   0,6(RLINK1)            except LLINK1, RLINK1 are used
85              ST1   0,2(LLINK1)            instead of LLINK2, RLINK2)
86              ST2   0,1(RLINK1)
87    9H        JMP   *
88    CYCLE1    STJ   2,6(NEXTINST)      Set NEXTINST(C) ← rJ.
89              JMP   CYCLE
90    HOLDC     STJ   2,6(NEXTINST)      Set NEXTINST(C) ← rJ.
91              JMP   HOLD               Insert NODE(C) in WAIT, delay (rA).
92    CYCLE     LD6   WAIT(RLINK1)       Set current node C ← RLINK1(LOC(WAIT)).
93              LDA   1,6                NEXTTIME(C)
94              STA   TIME                   becomes new value of simulated TIME.
95              JMP   DELETEW            Remove NODE(C) from WAIT list.
96              JMP   2,6                Jump to NEXTINST(C). ▮
```

Now comes the program for coroutine M. At the beginning of step M1, the current node C is MAN1 (see lines 12–14 above), and lines 099–100 of the program cause MAN1 to be reinserted into the WAIT list so that the next man will be generated after INTERTIME units of simulated time. The following lines 101–114 take care of setting up a node for the newly generated man; his IN and OUT floors are recorded in this node position. The AVAIL stack is singly linked in the RLINK1 field of each node. Note that lines 101–108 perform the action "C ⇐ AVAIL" using the POOLMAX technique, 2.2.3–(7); no test for OVERFLOW is necessary here, since the total size of the storage pool (the number of men in the system at any one time) rarely exceeds 10 nodes (40 words). The return of a node to the AVAIL stack appears in lines 156–158.

Throughout the program, index register 4 equals the variable FLOOR, and index register 5 is positive, negative, or zero, depending on whether STATE = GOINGUP, GOINGDOWN, or NEUTRAL, respectively. The variables CALLUP[j], CALLCAR[j], and CALLDOWN[j] occupy the respective fields (1:1), (3:3), and (5:5) of location CALL + j.

```
097   * COROUTINE M.                     M1. Enter, prepare for successor.
098   M1        JMP   VALUES             Compute IN, OUT, INTERTIME, GIVEUPTIME.
099             LDA   INTERTIME          INTERTIME is computed by VALUES subroutine.
100             JMP   HOLD               Put NODE(C) in WAIT, delay INTERTIME.
101             LD6   AVAIL              C ← AVAIL.
102             J6P   1F                 If AVAIL ≠ Λ, jump.
103             LD6   POOLMAX
104             INC6  4                  C ← POOLMAX + 4.
105             ST6   POOLMAX            POOLMAX ← C.
106             JMP   *+3
107   1H        LDA   0,6(RLINK1)
108             STA   AVAIL              AVAIL ← RLINK1(AVAIL).
109             LD1   INFLOOR            rI1 ← INFLOOR (computed by VALUES above).
110             ST1   0,6(IN)            IN(C) ← rI1.
111             LD2   OUTFLOOR           rI2 ← OUTFLOOR (computed by VALUES).
112             ST2   3,6(OUT)           OUT(C) ← rI2.
113             ENTA  39                 Put constant 39 (JMP operation code)
114             STA   2,6                    into third word of node format (6).
```

```
M2      ENTA  0,4                 M2. Signal and wait. Set rA ← FLOOR.
        DECA  0,1                 FLOOR — IN
        ST6   TEMP                Save value of C.
        JANZ  2F                  Jump if FLOOR ≠ IN.
        ENT6  ELEV1               Set C ← LOC(ELEV1).
        LDA   2,6(NEXTINST)       Is elevator positioned at E6?
        DECA  E6
        JANZ  3F
        ENTA  E3                  If so, reposition it at E3.
        STA   2,6(NEXTINST)
        JMP   DELETEW             Remove it from WAIT list
        JMP   4F                      and reinsert it at front of WAIT.
3H      LDA   D3
        JAZ   2F                  Jump if D3 = 0.
        ST6   D1                  Otherwise set D1 ≠ 0.
        STZ   D3                  Set D3 ← 0.
4H      JMP   IMMED               Insert ELEV1 at front of WAIT list.
        JMP   M3                  (rI1, rI2 have changed.)
2H      DEC2  0,1                 rI2 ← OUT−IN.
        ENTA  1
        J2P   *+3                 Jump if going up.
        STA   CALL,1(5:5)         Set CALLDOWN[IN] ← 1.
        JMP   *+2
        STA   CALL,1(1:1)         Set CALLUP[IN] ← 1.
        LDA   D2
        JAZ   DECISION            If D2 = 0, call the DECISION subroutine.
        LDA   ELEV1+2(NEXTINST)
        DECA  E1                  If the elevator is at E1, call
        JAZ   DECISION                the DECISION subroutine.
M3      LD6   TEMP                M3. Enter queue.
        LD1   0,6(IN)
        ENT1  QUEUE,1             rI1 ← LOC(QUEUE[IN]).
        JMP   INSERT              Insert NODE(C) at right end of QUEUE[IN].
M4A     LDA   GIVEUPTIME
        JMP   HOLDC               Wait GIVEUPTIME units.
M4      LDA   0,6(IN)             M4. Give up.
        DECA  0,4                 IN(C) — FLOOR
        JANZ  *+3
        LDA   D1                  FLOOR = IN(C).
        JANZ  M4A                 See exercise 7.
M6      JMP   DELETE              M6. Get out. NODE(C) is deleted
        LDA   AVAIL                   from QUEUE or ELEVATOR.
        STA   0,6(RLINK1)         AVAIL ⇐ C.
        ST6   AVAIL
        JMP   CYCLE               Continue simulation.
M5      JMP   DELETE              M5. Get in. NODE(C) is deleted
        ENT1  ELEVATOR                from QUEUE.
        JMP   INSERT              Insert it at right of ELEVATOR.
        ENTA  1
        LD2   3,6(OUT)
        STA   CALL,2(3:3)         Set CALLCAR[OUT(C)] ← 1.
        J5NZ  CYCLE               Jump if STATE ≠ NEUTRAL.
        DEC2  0,4
        ENT5  0,2                 Set STATE to proper direction.
        ENT6  ELEV2               Set C ← LOC(ELEV2).
        JMP   DELETEW             Remove E5 action from WAIT list.
        ENTA  25
        JMP   E5A                 Restart E5 action 25 units from now. ▮
```

The program for coroutine E is a rather straightforward rendition of the semiformal description given earlier. Perhaps the most interesting portion is the preparation for the elevator's independent actions in step E3, and the searching of the ELEVATOR and QUEUE lists in step E4.

```
173     * COROUTINE E.
174     E1A     JMP     CYCLE1          Set NEXTINST ← E1, go to CYCLE.
175     E1      EQU     *               E1. Wait for call. (no action)
176     E2A     JMP     HOLDC
177     E2      J5N     1F              E2. Change of state?
178             LDA     CALL+1,4        State is GOINGUP.
179             ADD     CALL+2,4
180             ADD     CALL+3,4
181             ADD     CALL+4,4
182             JAP     E3              Are there calls for higher floors?
183             LDA     CALL-1,4(3:3)   If not, have passengers in the
184             ADD     CALL-2,4(3:3)       elevator called for lower floors?
185             ADD     CALL-3,4(3:3)
186             ADD     CALL-4,4(3:3)
187             JMP     2F
188     1H      LDA     CALL-1,4        State is GOINGDOWN.
190             ADD     CALL-2,4        Actions are like lines 178–186.
. . .
196             ADD     CALL+4,4(3:3)
197     2H      ENN5    0,5             Reverse direction of STATE.
198             STZ     CALL,4          Set CALL variables to zero.
199             JANZ    E3              Jump if calls for opposite direction,
200             ENT5    0                   otherwise, set STATE ← NEUTRAL.
201     E3      ENT6    ELEV3           E3. Open door.
202             LDA     0,6             If activity E9 is already scheduled,
203             JANZ    DELETEW             remove it from the WAIT list.
204             ENTA    300
205             JMP     HOLD            Schedule activity E9 after 300 units.
206             ENT6    ELEV2
207             ENTA    76
208             JMP     HOLD            Schedule activity E5 after 76 units.
209             ST6     D2              Set D2 ≠ 0.
210             ST6     D1              Set D1 ≠ 0.
211             ENTA    20
212     E4A     ENT6    ELEV1
213             JMP     HOLDC
214     E4      ENTA    0,4             E4. Let people out, in.
215             SLA     4               Set OUT field of rA to FLOOR.
216             ENT6    ELEVATOR        C ← LOC(ELEVATOR).
217     1H      LD6     3,6(LLINK2)     C ← LLINK2(C).
218             CMP6    =ELEVATOR=      Search ELEVATOR list, right to left.
219             JE      1F              If C = LOC(ELEVATOR), search is complete
220             CMPA    3,6(OUT)        Compare OUT(C) with FLOOR.
221             JNE     1B              If not equal, continue search,
222             ENTA    M6                  otherwise, prepare to send man to M6
223             JMP     2F
224     1H      LD6     QUEUE+3,4(RLINK2)  Set C ← RLINK2(LOC(QUEUE[FLOOR])).
225             CMP6    3,6(RLINK2)     Is C = RLINK2(C)?
226             JE      1F              If so, the queue is empty.
227             JMP     DELETEW         If not, cancel action M4 for this man.
228             ENTA    M5              Prepare to send man to M5.
```

```
229   2H     STA   2,6(NEXTINST)     Set NEXTINST(C).
230          JMP   IMMED             Put him at front of WAIT list.
231          ENTA  25
232          JMP   E4A               Wait 25 units and repeat E4.
233   1H     STZ   D1                Set D1 ← 0.
234          ST6   D3                Set D3 ≠ 0.
235          JMP   CYCLE             Return to simulate other events.
236   E5A    JMP   HOLDC
237   E5     LDA   D1                E5. Close door.
238          JAZ   *+3               Is D1 = 0?
239          ENTA  40                If not, people are still getting in or out.
240          JMP   E5A               Wait 40 units, repeat E5.
241          STZ   D3                If D1 = 0, set D3 ← 0.
242          ENTA  20
243          ENT6  ELEV1
244   E6A    JMP   HOLDC             Wait 20 units, then go to E6.
245   E6     J5N   *+2               E6. Prepare to move.
246          STZ   CALL,4(1:3)       If STATE ≠ GOINGDOWN, CALLUP and CALLCAR
247          J5P   *+2                    on this floor are reset.
248          STZ   CALL,4(3:5)       If ≠ GOINGUP, reset CALLCAR and CALLDOWN.
249          J5Z   DECISION          Perform DECISION subroutine.
250   E6B    J5Z   E1A               If STATE = NEUTRAL, go to E1 and wait.
251          LDA   D2
252          JAZ   *+4
253          ENT6  ELEV3             Otherwise, if D2 ≠ 0,
254          JMP   DELETEW                cancel activity E9
255          STZ   ELEV3                  (see line 202).
256          ENTA  15
257          ENT6  ELEV1             Wait 15 units of time.
258          J5N   E8A               If STATE = GOINGDOWN, go to E8.
259   E7A    JMP   HOLDC
260   E7     INC4  1                 E7. Go up a floor.
261          ENTA  51
262          JMP   HOLDC             Wait 51 units.
263          LDA   CALL,4(1:3)       Is CALLCAR[FLOOR] or CALLUP[FLOOR] ≠ 0?
264          JAP   1F
265          ENT1  -2,4              If not,
266          J1Z   2F                     is FLOOR = 2?
267          LDA   CALL,4(5:5)       If not, is CALLDOWN[FLOOR] ≠ 0?
268          JAZ   E7                If not, repeat step E7.
269   2H     LDA   CALL+1,4
270          ADD   CALL+2,4
271          ADD   CALL+3,4
272          ADD   CALL+4,4
273          JANZ  E7                Are there calls for higher floors?
274   1H     ENTA  14                It is time to stop the elevator.
275          JMP   E2A               Wait 14 units and go to E2.
276   E8A    JMP   HOLDC
. . .                                (See exercise 8.)
293   E9     STZ   0,6               E9. Set inaction indicator.
294          STZ   D2                D2 ← 0.
295          JMP   DECISION          Perform DECISION subroutine.
296          JMP   CYCLE             Return to simulation of other events. ▮
```

We will not consider here the DECISION subroutine (see exercise 9), nor the VALUES subroutine which is used to specify the demands on the elevator. At the very end of the program comes the code

```
BEGIN     ENT4  2        Start with FLOOR = 2
          ENT5  0            and STATE = NEUTRAL.
          JMP   CYCLE    Begin simulation.
POOLMAX   CON   *+10     Storage pool follows literals, temp storage
          END   BEGIN          ▮
```

The above program does a fine job of simulating the elevator system, as it goes through its paces. But it would be useless to run this program, since there is no output! Actually, the author added a PRINT subroutine which was called at most of the critical steps in the program above, and this was used to prepare Table 1; the details have been omitted, since they are very straightforward but only clutter up the code.

Several programming languages have been devised which make it quite easy to specify the actions in a discrete simulation, and to use a compiler to translate these specifications into machine language. Assembly language was used in this section, of course, since we are concerned here with the techniques of manipulating linked lists, and the details of how discrete simulations are actually performed by a computer (although it has a one-track mind). We will consider the question of higher-level notations for describing these systems in Chapter 8. The technique of using a WAIT list or "agenda" to control the sequencing of coroutines, as we have done in this section, is called *quasi-parallel processing*.

It is quite difficult to give a precise analysis of the running time of such a long program, because of the complex interactions involved; it is easy to time various smaller parts of the program (like the INSERT subroutine) and this gives an indication of its efficiency. It is often useful to employ a special trace routine which executes the program, and which records how often each instruction was performed; this shows the "bottlenecks" in the program, places which should be given special attention. The author made such an experiment with the above program; the program ran for 10000 units of simulated time, and 26 men entered the simulated system. The instructions in the SORTIN loop, lines 73–75, were executed by far the most often, 1432 times, while the SORTIN subroutine itself was called 437 times. The CYCLE routine was performed 407 times, and this suggests that the DELETEW subroutine should not have been called at line 95; the four lines of that subroutine should have been written out in full (to save $4u$ each time CYCLE is used). The special trace routine also showed that the DECISION subroutine was called only 32 times and the loop in E4 (lines 216–218) was executed only 142 times.

It is hoped that some reader will learn as much about simulation from the above example as the author learned about elevators while the example was being prepared.

EXERCISES

1. [*21*] Give specifications for the insertion and deletion of information at the left end of a doubly linked list represented as in (1). (With the dual operations at the right end, which are obtained by symmetry, we therefore have all the actions of a general deque.)

▶ 2. [*22*] Explain why a list that is singly linked cannot allow efficient operation as a general deque; the deletion of items can be done efficiently at only one end of a singly linked list.

▶ 3. [*22*] The elevator system described in the text uses three call variables, `CALLUP`, `CALLCAR`, and `CALLDOWN`, for each floor, representing what buttons have been pushed by the men in the system. It is conceivable that internally the elevator needs only one or two relay circuits (i.e., binary variables) for the call buttons on each floor, instead of three. Show how a man could push buttons in a certain sequence with this elevator system to *prove* that there are three separate relays for each floor (except the top and bottom floors).

4. [*24*] Activity E9 in the elevator coroutine is usually canceled by step E6, and even when it hasn't been canceled, it doesn't do very much. Explain under what circumstances the elevator would behave differently (i.e., it would operate at a different speed, or visit floors in a different order) if activity E9 were deleted from the system.

5. [*20*] In Table 1, man no. 10 arrived on floor 0 at time 1048. Suppose he had arrived on floor 2 instead of floor 0; show that under these conditions the elevator would have gone *up* after receiving its passengers on floor 1, instead of down, in spite of the fact that man no. 8 wants to go down to floor 0.

6. [*23*] Note that in Table 1, time 1183–1233, men nos. 7, 8, and 9 all get in the elevator on floor 1; then the elevator goes down to floor 0 and only man no. 8 gets out. Now the elevator stops on floor 1, presumably to pick up men nos. 7 and 9 who are already aboard, and nobody is actually on floor 1 waiting to get in. (This situation occurs not infrequently at Caltech; if you get on the elevator going the wrong way, you must wait for an extra stop as you go by your original floor again.) In many elevator systems, men nos. 7 and 9 would not have gotten in the elevator at time 1183, since lights outside the elevator would show it was going down, not up; these men would have waited until the elevator came back up and stopped for them. On the system described, there are no such lights and it is impossible to tell which way the elevator is going to go until you are in it; hence Table 1 reflects the actual situation.

What changes should be made to coroutines M and E if we were to simulate the same elevator system, but with indicator lights, so that people do not get on the elevator when its state is contrary to their desired direction?

7. [*25*] Although "bugs" in programs are often embarrassing to a programmer, if we are to learn from our mistakes we should record them and tell other people about them instead of forgetting them! The following error (among others) was made by the author when he first wrote the program in this section: line 154 said "`JANZ CYCLE`" instead of "`JANZ M4A`". The reasoning was that if indeed the elevator had arrived at this man's floor and he would soon be able to get on, there was no need for him to perform his "give up" activity M4 any more, so we could simply go to `CYCLE` and continue simulating other activities. What is the error?

8. [*22*] Write the code for step E8, lines 277–292, which have been omitted from the program in the text.

9. [*23*] Write the code for the DECISION subroutine which has been omitted from the program in the text.

10. [*40*] It is perhaps significant to note that although the author had used the elevator system for years and thought he knew it well, it wasn't until he attempted to write this section that he realized there were quite a few facts about the elevator's system of choosing directions that he did not know. He went back to experiment with the elevator six separate times, each time believing he had finally achieved a complete understanding of its *modus operandi*. (Now he is reluctant to ride it for fear some new facet of its operation will appear, contradicting the algorithms given.) We often fail to realize how little we know about a thing until we attempt to simulate it on a computer.

Try to specify the actions of some elevator you are familiar with. Check the algorithm by experiments with the elevator itself (looking at its circuitry is not fair!); then design a discrete simulator for the system and run it on a computer.

▶ **11.** [*25*] (An "update-memory.") The following problem often arises in *synchronous* simulations: The system has n variables V[1], . . . , V[n], and at every simulated step new values for some of these are calculated from the old values. These calculations are assumed done "simultaneously" in the sense that the variables do not change to their new values until after all assignments have been made. Thus, the two statements

$$\mathtt{V[1]} \leftarrow \mathtt{V[2]} \qquad \text{and} \qquad \mathtt{V[2]} \leftarrow \mathtt{V[1]}$$

appearing at the same simulated time would interchange the values of V[1] and V[2], and this is quite different from what would happen in a sequential calculation.

The desired action can of course be simulated by keeping an additional table NEWV[1], . . . , NEWV[n]. Before each simulated step, we could set NEWV[k] ← V[k] for $1 \le k \le n$, then record all changes of V[k] in NEWV[k], and finally, after the step we could set V[k] ← NEWV[k], $1 \le k \le n$. But this "brute force" approach is often not completely satisfactory, for the following reasons: (1) Often n is very large, but the number of variables changed per step is rather small. (2) The variables are often not arranged in a nice table V[1], . . . , V[n], but are scattered throughout memory in a rather chaotic fashion. (3) This method does not detect the situation (usually an error in the model) when one variable is given two values in the same simulated step.

Assuming that the number of variables changed per step is rather small, design an efficient algorithm that simulates the desired actions, using two auxiliary tables NEWV[k] and LINK[k], $1 \le k \le n$. If possible, your algorithm should give an error stop if the same variable is being given two different values on the same step.

▶ **12.** [*22*] Why is it a good idea to use doubly linked lists instead of singly linked or sequential lists in the simulation program of this section?

2.2.6. Arrays and Orthogonal Lists

One of the simplest generalizations of a linear list is a two-dimensional or higher-dimensional array of information. For example, consider the case of an $m \times n$ matrix

$$\begin{pmatrix} \mathtt{A}[1, 1] & \mathtt{A}[1, 2] & \ldots & \mathtt{A}[1, n] \\ \mathtt{A}[2, 1] & \mathtt{A}[2, 2] & \ldots & \mathtt{A}[2, n] \\ \vdots & & & \vdots \\ \mathtt{A}[m, 1] & \mathtt{A}[m, 2] & \ldots & \mathtt{A}[m, n] \end{pmatrix}. \tag{1}$$

In this two-dimensional array, each node A[j, k] belongs to two linear lists: the "row j" list A[j, 1], A[j, 2], . . . , A[j, n], and the "column k" list A[1, k], A[2, k], . . . , A[m, k]. These orthogonal row and column lists essentially account for the two-dimensional structure of a matrix. Similar remarks apply to higher-dimensional arrays of information.

Sequential Allocation. When an array is stored in *sequential* memory locations, storage is usually allocated so that

$$\texttt{LOC(A[J,K])} = a_0 + a_1\texttt{J} + a_2\texttt{K}, \tag{2}$$

where a_0, a_1, and a_2 are constants. Let us consider a more general case: Suppose we have a four-dimensional array with one-word elements Q[I,J,K,L] for $0 \le \texttt{I} \le 2$, $0 \le \texttt{J} \le 4$, $0 \le \texttt{K} \le 10$, $0 \le \texttt{L} \le 2$. We would like to allocate storage so that

$$\texttt{LOC(Q[I,J,K,L])} = a_0 + a_1\texttt{I} + a_2\texttt{J} + a_3\texttt{K} + a_4\texttt{L}. \tag{3}$$

This means that a change in I, J, K, or L leads to a readily calculated change in the location of Q[I,J,K,L]. The most natural (and most commonly used) way to allocate storage is to let the array appear in memory in the "lexicographic order" of its indices, sometimes called "row major order":

Q[0, 0, 0, 0], Q[0, 0, 0, 1], Q[0, 0, 0, 2], Q[0, 0, 1, 0], Q[0, 0, 1, 1], . . . ,
Q[0, 0, 10, 2], Q[0, 1, 0, 0], . . . , Q[0, 4, 10, 2], Q[1, 0, 0, 0], . . . ,
Q[2, 4, 10, 2].

It is easy to see that this order satisfies the requirements of (3), and we have

$$\texttt{LOC(Q[I, J, K, L])} = \texttt{LOC(Q[0, 0, 0, 0])} + 165\texttt{I} + 33\texttt{J} + 3\texttt{K} + \texttt{L}. \tag{4}$$

In general, given a k-dimensional array with c-word elements A[$\texttt{I}_1$, $\texttt{I}_2$, . . . , $\texttt{I}_k$] for $0 \le \texttt{I}_1 \le d_1$, $0 \le \texttt{I}_2 \le d_2$, . . . , $0 \le \texttt{I}_k \le d_k$, we can store it in memory as

$$\begin{aligned}\texttt{LOC(A[I}_1, \texttt{I}_2, \ldots, \texttt{I}_k\texttt{])} &= \texttt{LOC(A[0, 0, . . . , 0])} \\ &\quad + c(d_2+1)\cdots(d_k+1)\texttt{I}_1 + \cdots \\ &\quad + c(d_k+1)\texttt{I}_{k-1} + c\texttt{I}_k \\ &= \texttt{LOC(A[0, 0, . . . , 0])} + \sum_{1\le r\le k} a_r\texttt{I}_r,\end{aligned}$$

where

$$a_r = c \prod_{r<s\le k} (d_s + 1). \tag{5}$$

To see why this formula works, observe that a_r is the amount of memory needed to store the subarray A[$\texttt{I}_1$, . . . , $\texttt{I}_r$, $\texttt{J}_{r+1}$, . . . , $\texttt{J}_k$] if $\texttt{I}_1$, . . . , $\texttt{I}_r$ are constant and $\texttt{J}_{r+1}$, . . . , $\texttt{J}_k$ vary through all values $0 \le \texttt{J}_{r+1} \le d_{r+1}$, . . . , $0 \le \texttt{J}_k \le d_k$; hence by the nature of lexicographic order the address of A[$\texttt{I}_1$, . . . , $\texttt{I}_k$] should change by precisely this amount when $\texttt{I}_r$ changes by 1.

Note the similarity between formula (5) and the value of the number $I_1 I_2 \ldots I_k$ in a mixed-radix number system. For example, if we had the array TIME[W, D, H, M, S] with $0 \le W < 4$, $0 \le D < 7$, $0 \le H < 24$, $0 \le M < 60$, and $0 < S < 60$, the location of TIME[W, D, H, M, S] would be the location of TIME[0, 0, 0, 0, 0] plus the quantity "W weeks + D days + H hours + M minutes + S seconds" converted to seconds. Of course, it takes a pretty large computer and a pretty fancy application to make use of an array which has 2419200 elements.

The above method for storing arrays is generally suitable when the array has a complete rectangular structure, i.e., when all elements $A[I_1, I_2, \ldots, I_k]$ are present for indices in the independent ranges $l_1 \le I_1 \le u_1$, $l_2 \le I_2 \le u_2$, $\ldots$, $l_k \le I_k \le u_k$. There are many situations in which this is not the case; most common among these is the *triangular matrix*, where we want to store only the entries $A[j, k]$ for, say, $0 \le k \le j \le n$:

$$\begin{pmatrix} A[0,0] & & & \\ A[1,0] & A[1,1] & & \\ \vdots & & & \\ A[n,0] & A[n,1] & \ldots & A[n,n] \end{pmatrix} \tag{6}$$

We may know that all other entries are zero, or that $A[j, k] = A[k, j]$, so only half of the values need to be stored. If we want to store the lower triangular matrix (6) in $\frac{1}{2}(n+1)(n+2)$ consecutive memory positions, we are forced to give up the possibility of linear allocation as in Eq. (2), but we can now ask instead for an allocation arrangement of the form

$$\mathrm{LOC}(A[J,K]) = a_0 + f_1(J) + f_2(K) \tag{7}$$

where f_1 and f_2 are functions of one variable. (The constant a_0 may be absorbed into either f_1 or f_2 if desired.) When the addressing has the form (7), a random element $A[j, k]$ can be quickly accessed if we keep two (rather short) auxiliary tables of the values of f_1 and f_2, so that these functions need to be calculated only once.

It turns out that lexicographic order of indices for the array (6) satisfies condition (7), and, assuming one-word entries, we have in fact the rather simple formula

$$\mathrm{LOC}(A[J,K]) = \mathrm{LOC}(A[0,0]) + \frac{J(J+1)}{2} + K. \tag{8}$$

There is a far better way to store triangular matrices if we are fortunate enough to have two of them with the same size. If $A[j, k]$ and $B[j, k]$ are both to be stored for $0 \le k \le j \le n$, we can interlace them within a single matrix $C[j, k]$ for $0 \le j \le n$, $0 \le k \le n+1$, using the convention

$$A[j,k] = C[j,k], \qquad B[j,k] = C[k, j+1]. \tag{9}$$

Thus

$$\begin{pmatrix} \texttt{C[0, 0]} & \texttt{C[0, 1]} & \texttt{C[0, 2]} & \ldots & \texttt{C[0, }n+1\texttt{]} \\ \texttt{C[1, 0]} & \texttt{C[1, 1]} & \texttt{C[1, 2]} & \ldots & \texttt{C[1, }n+1\texttt{]} \\ \vdots & & & & \vdots \\ \texttt{C[}n\texttt{, 0]} & \texttt{C[}n\texttt{, 1]} & \texttt{C[}n\texttt{, 2]} & \ldots & \texttt{C[}n\texttt{, }n+1\texttt{]} \end{pmatrix}$$

$$\equiv \begin{pmatrix} \texttt{A[0, 0]} & \texttt{B[0, 0]} & \texttt{B[1, 0]} & \ldots & \texttt{B[}n\texttt{, 0]} \\ \texttt{A[1, 0]} & \texttt{A[1, 1]} & \texttt{B[1, 1]} & \ldots & \texttt{B[}n\texttt{, 1]} \\ \vdots & & & & \vdots \\ \texttt{A[}n\texttt{, 0]} & \texttt{A[}n\texttt{, 1]} & \texttt{A[}n\texttt{, 2]} & \ldots & \texttt{B[}n\texttt{, }n\texttt{]} \end{pmatrix}$$

The two triangular matrices are packed together tightly within the space of $(n+1)(n+2)$ locations, and we have linear addressing as in (2).

The generalization of triangular matrices to higher dimensions is called a *tetrahedral array*. This interesting topic is the subject of exercises 6 through 8.

As an example of typical programming techniques for use with sequentially stored arrays, see exercise 1.3.2–10 and the two answers given for that exercise. The fundamental techniques for efficient traversal of rows and columns, as well as the uses of sequential stacks, are of particular interest within those programs.

Linked Allocation. Linked memory allocation also applies to higher-dimensional arrays of information in a natural way. In general, our nodes can contain k link fields, one for each list the node is in. The use of linked memory is generally for cases in which the arrays are not strictly rectangular in character.

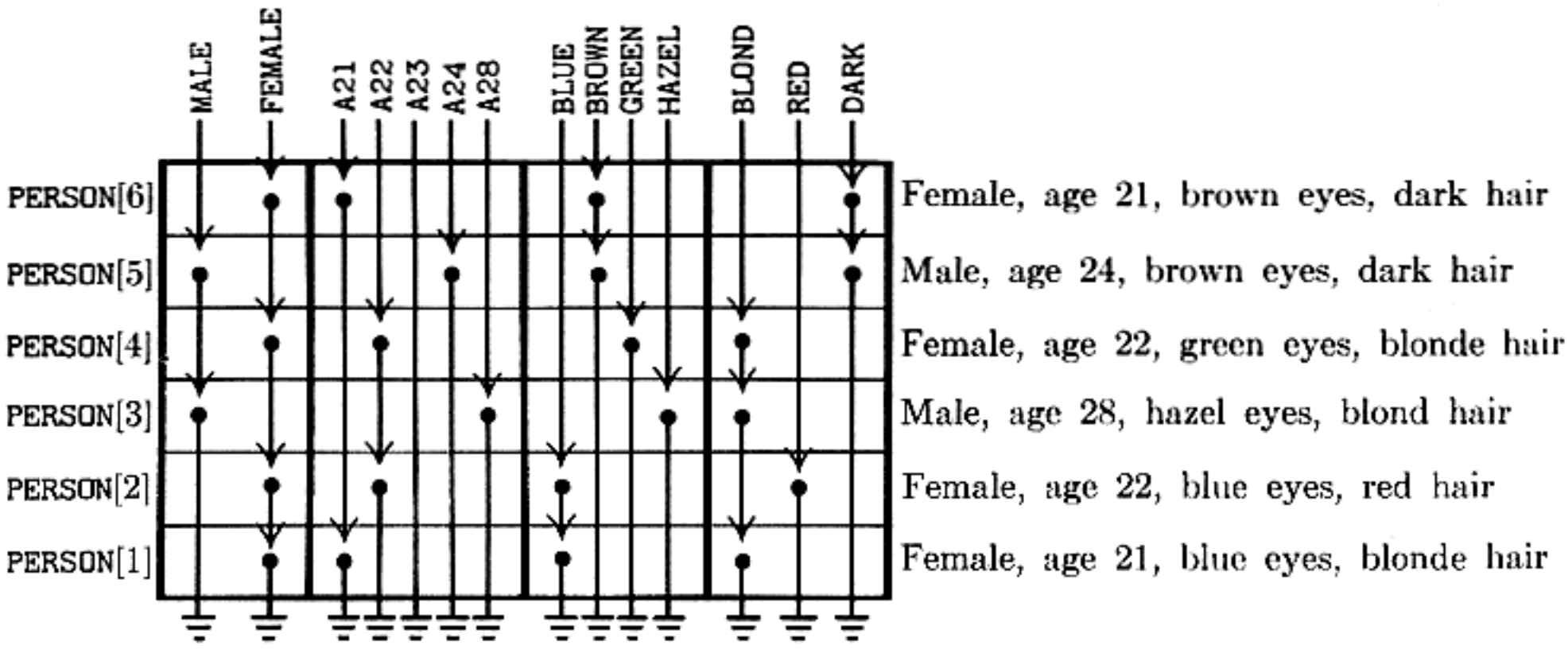


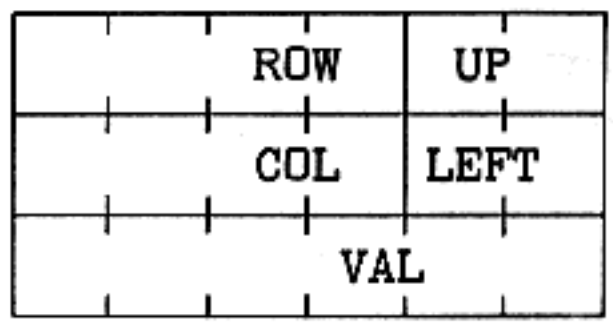
Fig. 13. Each node in four different lists.

As an example, suppose that we have a list in which every node is to represent a person, and that there are four link fields, SEX, AGE, EYES, and HAIR. In the EYES field we link together all nodes with the same eye color, etc. (See Fig. 13.) It is easy to visualize algorithms for inserting new people into this list. (Deletion would be harder, without double linking.) We can also conceive of algorithms

of varying degrees of efficiency for doing things like "Find all blue-eyed blonde women of ages 21 through 23"; see exercises 9 and 10. Problems in which each node of a list is to reside in several kinds of other lists at once arise rather frequently; indeed, the elevator system simulation described in the preceding section has nodes which are in both the QUEUE and WAIT lists at once.

As a detailed example of the use of linked allocation for orthogonal lists, we will consider the case of *sparse matrices* (i.e., matrices of large order in which most of the elements are zero). The goal is to operate on these matrices as though the entire matrix were present, but to save great amounts of memory space because the zero entries need not be represented. One way to do this, intended for random references to elements of the matrix, would be to use the storage-and-retrieval methods of Chapter 6, to find A[j, k] from the key "j, k"; however, there is another way to deal with sparse matrices that is usually preferable because it reflects the matrix structure more appropriately, and this is the method we will discuss here.

The representation we will discuss consists of circularly linked lists for each row and column. Each node of the matrix contains three words and five fields:



(10)

Here ROW and COL are the row and column indices of the node; VAL is the value stored at that part of the matrix; LEFT and UP are links to the next nonzero entry to the left in the row, or upward in the column, respectively. There are special list head nodes, BASEROW[i] and BASECOL[j], for every row and column. These nodes are identified by

$$\texttt{COL(LOC(BASEROW[}i\texttt{]))} < 0 \qquad \text{and} \qquad \texttt{ROW(LOC(BASECOL[}j\texttt{]))} < 0.$$

As usual in a circular list, the LEFT link in BASEROW[i] is the location of the rightmost value in that row, and UP in BASECOL[j] is the lowest value in that column. For example, the matrix

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix} \tag{11}$$

would be represented as shown in Fig. 14.

Using sequential allocation of storage, a 200×200 matrix would take 40000 words, and this is more memory than many computers have; but a suitably sparse 200×200 matrix can be represented as above even in MIX's 4000-word memory. (See exercise 11.) The amount of time taken to access a

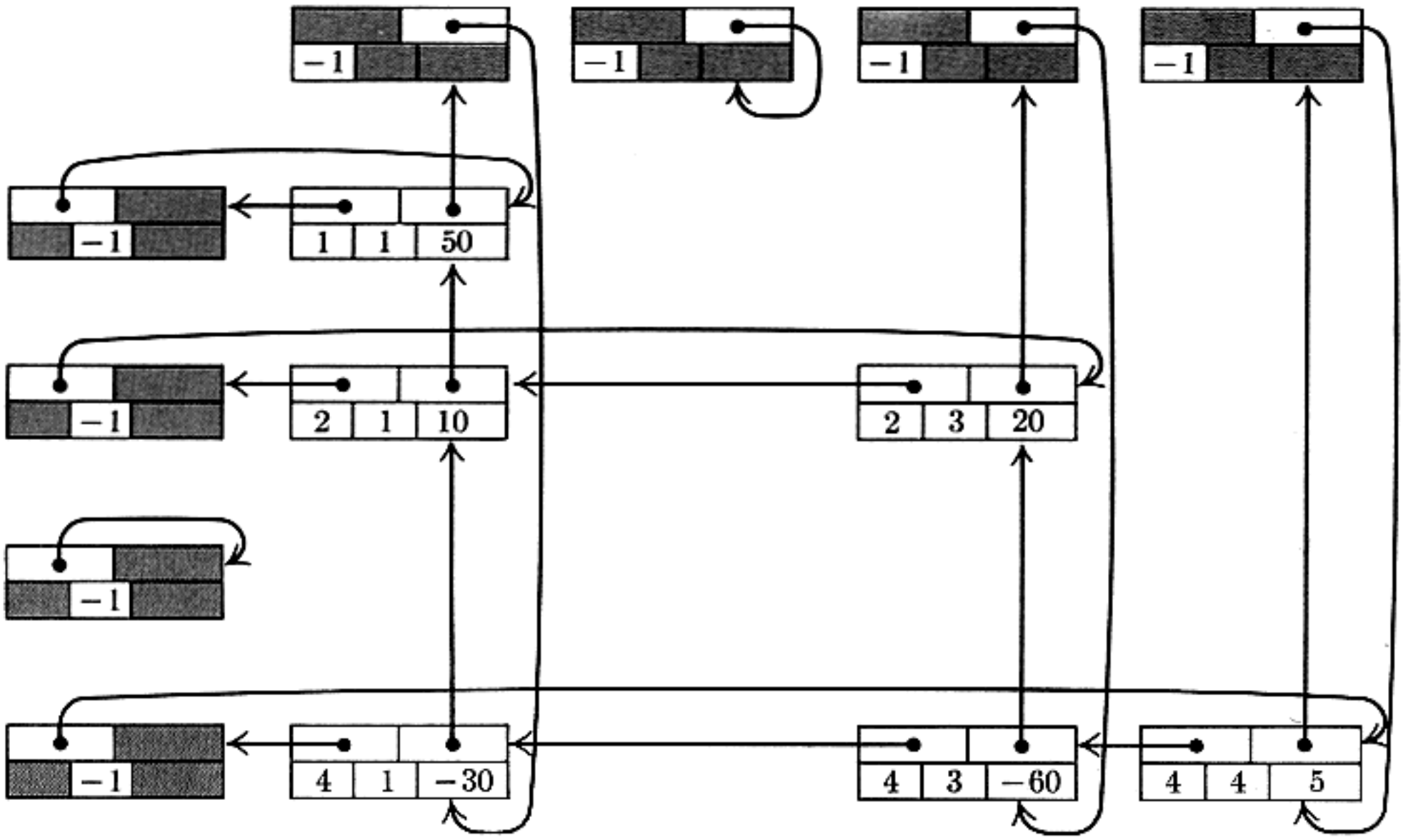


Fig. 14. Representation of the matrix (11); the nodes are illustrated in the format

LEFT		UP
ROW	COL	VAL

.

List heads appear at the left and the top.

random element A[j, k] is also quite reasonable, *if* there are but few elements in each row or column; and since most matrix algorithms proceed by walking sequentially through a matrix, instead of accessing elements at random, this linked representation entails little loss of running speed.

As a typical example of a nontrivial algorithm dealing with sparse matrices in this notation, we will consider the *pivot step* operation, which is an important part of algorithms for solving linear equations, for inverting matrices, and for solving linear programming problems by the simplex method. A pivot step is the following matrix transformation:

$$\begin{array}{r} \\ \\ \text{Pivot row} \\ \\ \text{Any other row} \\ \\ \end{array}
\begin{array}{c} \text{Before pivot step} \\ \begin{array}{ccccc} & \text{Pivot column} & & \text{Any other column} & \\ \end{array} \\ \begin{pmatrix} & \vdots & & \vdots & \\ \cdots & a & \cdots & b & \cdots \\ & \vdots & & \vdots & \\ \cdots & c & \cdots & d & \cdots \\ & \vdots & & \vdots & \end{pmatrix} \end{array},
\begin{array}{c} \text{After pivot step} \\ \begin{array}{ccccc} & \text{Pivot column} & & \text{Any other column} & \\ \end{array} \\ \begin{pmatrix} & \vdots & & \vdots & \\ \cdots & 1/a & \cdots & b/a & \cdots \\ & \vdots & & \vdots & \\ \cdots & -c/a & \cdots & d - \dfrac{bc}{a} & \cdots \\ & \vdots & & \vdots & \end{pmatrix} \end{array}. \tag{12}$$

It is assumed that the "pivot element" a is nonzero. For example, a pivot step applied to matrix (11), with the element 10 in row 2 column 1 as pivot, leads to

$$\begin{pmatrix} -5 & 0 & -100 & 0 \\ 0.1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 5 \end{pmatrix}. \tag{13}$$

Our goal is to design an algorithm which performs this pivot operation on sparse matrices that are represented as in Fig. 14. It is clear that the transformation (12) affects only those rows of a matrix for which there is a nonzero element in the pivot column, and it affects only those columns for which there is a nonzero entry in the pivot row. Hence when a large sparse matrix is being considered, we are not only achieving a reduction in space by the linked representation for nonzero elements, but are also perhaps achieving an increase in the speed of pivoting.

The pivoting algorithm is in many ways a straightforward application of linking techniques we have already discussed; in particular, it bears strong resemblances to Algorithm 2.2.4A for addition of polynomials. There are two things, however, which make the problem a little tricky: if in (12) we have $b \neq 0$ and $c \neq 0$ but $d = 0$, the sparse matrix representation has no entry for d and we must insert a new entry; and if $b \neq 0$, $c \neq 0$, $d \neq 0$, but $d - bc/a = 0$, we must delete the entry that was formerly there. These insertion and deletion operations in a two-dimensional array are more interesting than the one-dimensional case; to do them we must know what links are affected. Our algorithm processes the matrix rows successively from bottom to top. The efficient ability to insert and delete involves the introduction of a set of pointer variables PTR[j], one for each column considered, which traverse the columns upwards and thereby give us the ability to update the proper links in both dimensions.

Algorithm S (*Pivot step in a sparse matrix*). Given a matrix represented as in Fig. 14, we perform the pivot operation (12). Assume that PIVOT is a link variable pointing to the pivot element. The algorithm makes use of an auxiliary table of link variables PTR[j], one for each column of the matrix.

S1. [Initialize.] Set IO ← ROW(PIVOT), JO ← COL(PIVOT), ALPHA ← 1.0/VAL(PIVOT), VAL(PIVOT) ← 1.0, PO ← LOC(BASEROW[IO]), QO ← LOC(BASECOL[JO]). (*Note:* The variable ALPHA and the VAL field of each node are assumed to be floating-point or rational quantities, while everything else in this algorithm has integer values.)

S2. [Process pivot row.] Set PO ← LEFT(PO), J ← COL(PO). If J < 0, go on to step S3 (the pivot row has been traversed). Otherwise set PTR[J] ← LOC(BASECOL[J]) and VAL(PO) ← ALPHA × VAL(PO), and repeat step S2.

S3. [Find new row.] Set QO ← UP(QO). (The remainder of the algorithm deals successively with each row, from bottom to top, for which there is an entry

in the pivot column.) Set I ← ROW(Q0). If I < 0, the algorithm terminates. If I = I0, repeat step S3 (we have already done the pivot row). Otherwise set P ← LOC(BASEROW[I]), P1 ← LEFT(P). (The pointers P and P1 will now proceed across row I from right to left, as P0 goes in synchronization across row I0; Algorithm 2.2.4A is analogous. At this point,

$$\texttt{P0} = \texttt{LOC(BASEROW[I0])}.)$$

S4. [Find new column.] Set P0 ← LEFT(P0), J ← COL(P0). If J < 0, set VAL(Q0) ← −ALPHA × VAL(Q0) and return to S3. If J = J0, repeat step S4. (Thus we process the pivot column entry in row I *after* all other column entries have been processed; the reason is that VAL(Q0) is needed in step S7.)

S5. [Find I, J element.] If COL(P1) > J, set P ← P1, P1 ← LEFT(P), and repeat step S5. If COL(P1) = J, go to step S7. Otherwise go to step S6 (we need to insert a new element in column J of row I).

S6. [Insert I, J element.] If ROW(UP(PTR[J])) > I, then set PTR[J] ← UP(PTR[J]), and repeat step S6. (Otherwise, we will have ROW(UP(PTR[J])) < I; the new element is to be inserted just after NODE(PTR[J]) in the vertical dimension, and after NODE(P) in the horizontal dimension.) Otherwise set X ⇐ AVAIL, VAL(X) ← 0, ROW(X) ← I, COL(X) ← J, LEFT(X) ← P1, UP(X) ← UP(PTR[J]), LEFT(P) ← X, UP(PTR[J]) ← X, P1 ← X, PTR[J] ← X.

S7. [Pivot.] Set VAL(P1) ← VAL(P1) − VAL(Q0) × VAL(P0). If now VAL(P1) = 0, go to S8. (*Note:* When floating-point arithmetic is being used, this test "VAL(P1) = 0" should be replaced by "|VAL(P1)| < EPSILON" or better yet by the condition "most of the significant figures of VAL(P1) were lost in the subtraction.") Otherwise, set PTR[J] ← P1, P ← P1, P1 ← LEFT(P), and go back to S4.

S8. [Delete I, J element.] If UP(PTR[J]) ≠ P1 (or, what is essentially the same thing, if ROW(UP(PTR[J])) > I), set PTR[J] ← UP(PTR[J]) and repeat step S8; otherwise, set UP(PTR[J]) ← UP(P1), LEFT(P) ← LEFT(P1), AVAIL ⇐ P1, P1 ← LEFT(P). (NODE(P1) has been deleted.) Go back to S4. ▮

The programming of this algorithm is left as a very instructive exercise for the reader (see exercise 15). It is worth pointing out here that it is necessary to allocate only one word of memory to each of the nodes BASEROW[i], BASECOL[j], since most of their fields are irrelevant. (See the shaded areas in Fig. 14, and see the program of Section 2.2.5.) Furthermore, the value −PTR[j] can be stored as ROW(LOC(BASECOL[j])) for additional storage space economy. The running time of Algorithm S is very roughly proportional to the number of matrix elements affected by the pivot operation.

EXERCISES

1. [*17*] Give a formula for `LOC(A[J,K])` if `A` is the matrix of (1), and if each node of the array is two words long, assuming that the nodes are stored consecutively in lexicographic order of the indices.

▶ **2.** [*21*] Formula (5) has been derived from the assumption $0 \le \mathtt{I}_r \le d_r$ for $1 \le r \le k$; give a general formula that applies to the case $l_r \le \mathtt{I}_r \le u_r$, where l_r and u_r are any lower and upper bounds on the dimensionality.

3. [*21*] The text considers lower triangular matrices $\mathtt{A}[j, k]$ for $0 \le k \le j \le n$. How can the discussion of such matrices readily be modified for the case that subscripts start at 1 instead of 0, i.e., the case that $1 \le k \le j \le n$?

4. [*22*] Show that if we store the *upper* triangular array $\mathtt{A}[j, k]$ for $0 \le j \le k \le n$ in lexicographic order of the indices, the allocation satisfies the condition of Eq. (7). Find a formula for `LOC(A[J,K])` in this case.

5. [*20*] Show that it is possible to bring the value of `A[J,K]` into register A in one `MIX` instruction, using the "indirect addressing" feature of exercise 2.2.2–3, even when `A` is a *triangular* matrix as in (8). (Assume that the values of `J` and `K` are in index registers.)

▶ **6.** [*M24*] Consider the "tetrahedral arrays" $\mathtt{A}[i, j, k]$, $\mathtt{B}[i, j, k]$, where $0 \le k \le j \le i \le n$ in `A`, and $0 \le i \le j \le k \le n$ in `B`. Suppose that both of these arrays are stored in consecutive memory locations in lexicographic order of the indices; show that `LOC(A[I,J,K])` $= a_0 + f_1(\mathtt{I}) + f_2(\mathtt{J}) + f_3(\mathtt{K})$ for certain functions f_1, f_2, f_3, but that `LOC(B[I,J,K])` *cannot* be expressed in such a simple form.

7. [*M23*] Find a general formula to allocate storage for the k-dimensional tetrahedral array $\mathtt{A}[i_1, i_2, \ldots, i_k]$, where $0 \le i_k \le \cdots \le i_2 \le i_1 \le n$.

8. [*33*] (P. Wegner.) Suppose that `A[I,J,K]`, `B[I,J,K]`, `C[I,J,K]`, `D[I,J,K]`, `E[I,J,K]`, and `F[I,J,K]` are six tetrahedral arrays that are to be stored in memory for $0 \le \mathtt{K} \le \mathtt{J} \le \mathtt{I} \le n$. Is there a neat way to accomplish this, analogous to (9) in the two-dimensional case?

9. [*22*] Suppose a table, like that indicated in Fig. 13, but much larger, has been set up so that all links go in the same direction as shown there (i.e., `LINK(X)` $<$ `X` for all nodes and links). Design an algorithm which finds the addresses of all blue-eyed blonde girls of ages 21 through 23, by going through the various link fields in such a way that upon completion of the algorithm at most one pass has been made through each of the lists `FEMALE`, `A21`, `A22`, `A23`, `BLOND`, and `BLUE`.

10. [*26*] Can you think of a better way to organize a personnel table so that searches as described in the previous exercise would be more efficient? (The answer to this exercise is *not* merely "yes" or "no.")

11. [*11*] Suppose that we have a 200×200 matrix in which there are at most four nonzero entries per row. How much storage is required to represent this matrix as in Fig. 14, if we use three words per node except for list heads, which will use one word?

▶ **12.** [*20*] What are `VAL(Q0)`, `VAL(P0)`, and `VAL(P1)` at the beginning of step S7, in terms of the notation a, b, c, d used in (12)?

▶ **13.** [*22*] Why were circular lists used in Fig. 14 instead of straight linear lists? Could Algorithm S be rewritten so that it does not make use of the circular linkage?

14. [*22*] Algorithm S actually saves pivoting time in a sparse matrix, since it avoids consideration of those columns in which the pivot row has a zero entry. Show that this savings in running time can be achieved in a large sparse matrix that is stored sequentially, with the help of an auxiliary table $\mathtt{LINK}[j]$, $1 \le j \le n$.

▶ **15.** [*29*] Write a MIXAL program for Algorithm S. Assume that the VAL field is a floating-point number, and that MIX's floating-point arithmetic operators FADD, FSUB, FMUL, and FDIV can be used for operations on this field. Assume for simplicity that FADD and FSUB return the answer zero when the operands added or subtracted cancel most of the significance, so that the test "VAL(P1) = 0" may be safely used in step S7.

16. [*25*] Design an algorithm to *copy* a sparse matrix. (In other words, the algorithm is to yield two distinct representations of a matrix in memory, having the form of Fig. 14, given just one such representation initially.)

17. [*26*] Design an algorithm to *multiply* two sparse matrices; given matrices A and B, form a new matrix C, where $C[i, j] = \sum_k A[i, k]B[k, j]$. The two input matrices and the output matrix should be represented as in Fig. 14.

18. [*22*] The following algorithm replaces a matrix by the inverse of that matrix, assuming that the entries are $A[i, j]$, for $1 \leq i, j \leq n$, and using "Gauss-Jordan reduction":

a) For $k = 1, 2, \ldots, n$ do the following: Search row k in all columns not yet used as a pivot column, to find the entry with the greatest absolute value; set $C[k]$ equal to the column in which this entry was found, and do a pivot step with this entry as pivot. (If all such entries are zero, the matrix is singular and has no inverse.)

b) Permute rows and columns so that what was row k becomes row $C[k]$, and what was column $C[k]$ becomes column k.

The problem in this exercise is to use the above algorithm to find the inverse of the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

by hand calculation.

19. [*31*] Modify the "Gauss-Jordan reduction" algorithm described in exercise 18 so that it obtains the inverse of a sparse matrix that is represented in the form of Fig. 13. Pay special attention to making the row- and column-permutation operations of step (b) efficient.

2.3. TREES

We now turn to a study of trees, the most important nonlinear structures arising in computer algorithms. Generally speaking, tree structure means a "branching" relationship between nodes, much like that found in the trees of nature.

Let us define a *tree* formally as a finite set T of one or more nodes such that

a) There is one specially designated node called the *root* of the tree, root (T); and
b) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$, and each of these sets in turn is a tree. The trees $T_1, \ldots, T_m$ are called the *subtrees* of the root.

The definition just given is recursive, i.e., we have defined a tree in terms of trees. Of course, there is no problem of circularity involved here, since trees with one node must consist of only the root, and trees with $n > 1$ nodes are defined in terms of trees with less than n nodes; hence the concept of a tree with two nodes, three nodes, or ultimately any number of nodes, is determined by the definition given. There are nonrecursive ways to define trees (for example, see exercises 10, 12, and 14, and Section 2.3.4), but a recursive definition seems most appropriate since recursion is an innate characteristic of tree structures. The recursive character of trees is present also in nature, since buds on young trees eventually grow into subtrees with buds of their own, etc. It is easy to give rigorous proofs of important facts about trees based on a recursive definition such as the one above, by using induction on the number of nodes in a tree; for example, see exercise 3.

It follows from our definition that every node of a tree is the root of some subtree contained in the whole tree. The number of subtrees of a node is called the *degree* of that node. A node of degree zero is called a *terminal node* or sometimes a "leaf." A nonterminal node is often called a *branch node*. The *level* of a node with respect to T is defined by saying that the root has level 1, and other nodes have a level that is one higher than they have with respect to the subtree of the root, T_j, which contains them.

These concepts are illustrated in Fig. 15, which shows a tree with seven nodes. The root is A, and it has the two subtrees $\{B\}$ and $\{C, D, E, F, G\}$. The tree $\{C, D, E, F, G\}$ has node C as its root. Node C is on level 2 with respect to the whole tree, and it has three subtrees $\{D\}$, $\{E\}$, and $\{F, G\}$;

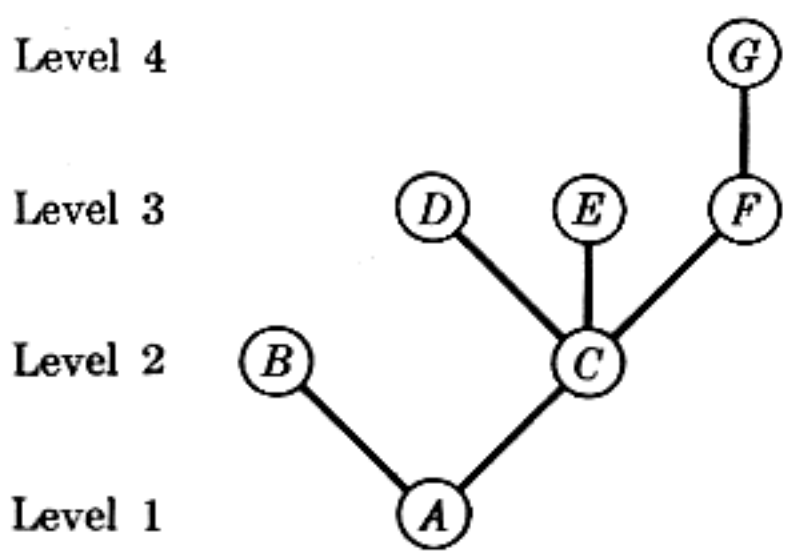


Fig. 15. A tree.

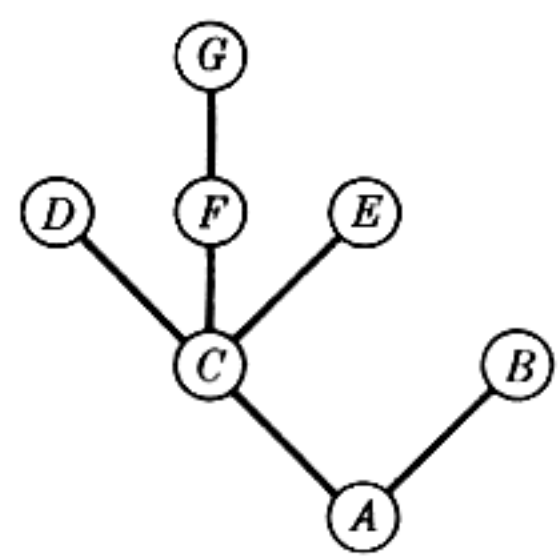


Fig. 16. Another tree.

therefore C has degree 3. The terminal nodes in Fig. 15 are B, D, E, and G; F is the only node with degree 1; G is the only node with level 4.

If the relative order of the subtrees $T_1, \ldots, T_m$ in (b) of the definition is important, we say the tree is an *ordered tree;* when $m \geq 2$ in an ordered tree, it makes sense to call T_2 the "second subtree" of the root, etc. If we do not care to regard two trees as different when they differ only in the respective ordering of subtrees of nodes, the tree is said to be *oriented,* since only the relative orientation of the nodes, not their order, is being considered. The very nature of computer representation defines an implicit ordering for any tree, so in most cases ordered trees are of greatest interest to us. We will therefore tacitly assume that *all trees we discuss are ordered, unless it is explicitly stated otherwise.* Accordingly, the trees of Figs. 15 and 16 will generally be considered to be different, although they would be the same as oriented trees.

A *forest* is a set (usually an ordered set) of zero or more disjoint trees. Another way to phrase part (b) of the definition of tree would be to say that *the nodes of a tree excluding the root form a forest.* (Some authors use the term "n-tuply rooted tree" to denote a forest of n trees.)

There is very little distinction between abstract forests and trees; if we delete the root of a tree, we have a forest, and, conversely, if we add just one node to any forest we get a tree. Therefore the words tree and forest are often used almost interchangeably during informal discussions about tree structures.

There are many ways to draw diagrams of trees. Besides the diagram of Fig. 15, three of the principal alternatives are shown in Fig. 17, depending on where the root is placed. It is not a frivolous joke to worry about how a tree structure is drawn in diagrams, since there are many occasions in which we would like to say one node is "above" or "higher than" another node, or to refer to the "rightmost" element, etc. Certain algorithms for dealing with tree structures have become known as "top down" methods, as opposed to "bottom up." This terminology leads to confusion unless we adhere to a uniform convention for drawing trees.

It may seem that the form of Fig. 15 would be preferable simply because that is how trees grow in nature; in the absence of any compelling reason to

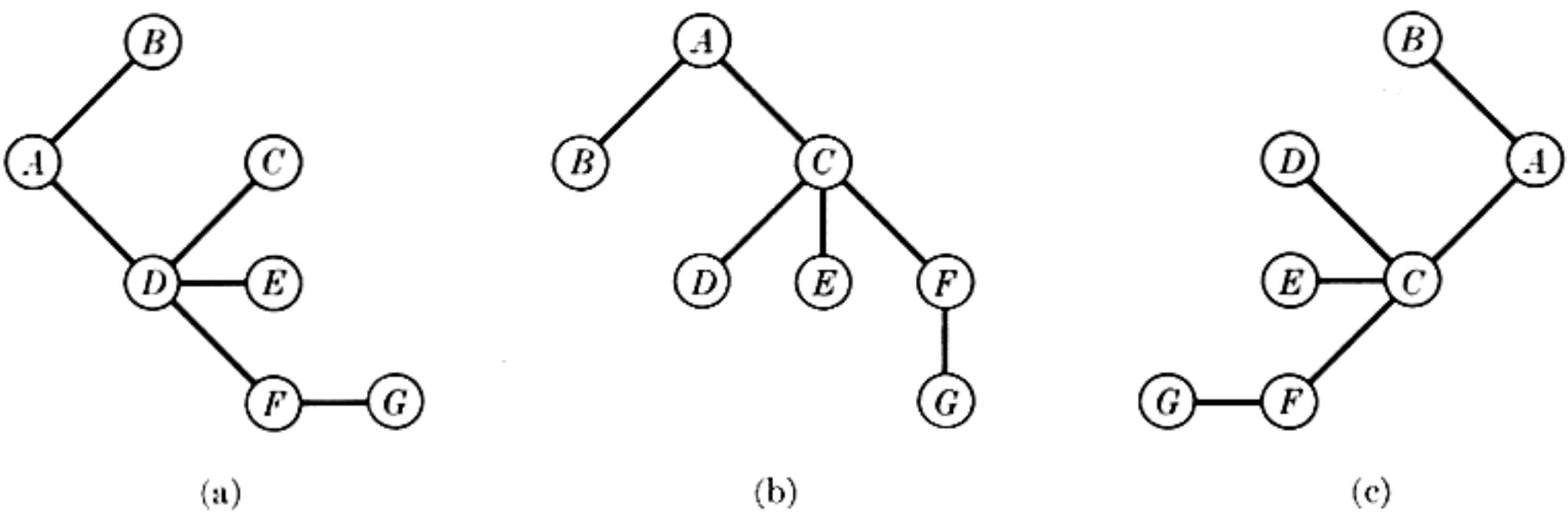


Fig. 17. How shall we draw a tree?

adopt any of the other three forms, we might as well adopt nature's time-honored tradition. With this in mind, the author consistently followed a root-at-the-bottom convention as the present set of books was being prepared, but after two years of trial it was found to be a mistake: observations of the computer literature and numerous informal discussions with computer scientists about a wide variety of algorithms showed that trees were drawn with the *root at the top* in over 80 percent of the cases examined. There is an overwhelming tendency to make hand-drawn charts grow downwards instead of upward (and this is easy to understand in view of the way we write); even the word "subtree" (as opposed to "supertree") tends to connote a downward relationship. From these considerations we conclude that *Fig. 15 is upside down.* Henceforth we will almost always draw trees as in Fig. 17(b), with the root at the top and leaves at the bottom. Until everyone is accustomed to trees with this orientation, however, it may be wise to refrain from using words that imply any vertical relationship, unless accompanying diagrams resolve the ambiguity.

It is necessary to have good descriptive terminology for talking about trees. Instead of the ambiguous references to "above" and "below" in a tree, genealogical words taken from the terminology of *family trees* have been found very appropriate for this purpose. Figure 19 shows two common types of family trees. The two types are obviously quite different: a "pedigree" shows the ancestors of a given individual, while a "lineal chart" shows the descendants.

If "cross-breeding" occurs, a family tree is not really a tree, because different branches of a tree (as we have defined it) can never be joined together. To compensate for this discrepancy, note that Queen Victoria and Prince Albert appear twice in the sixth generation in Fig. 19(a), and that King Christian IX actually appears in both the fifth and sixth generations. The family tree is a true tree if each of its nodes represents, not a person, but "a person in his role as parent of so-and-so."

Standard terminology for tree structures is taken from the *second* form of family tree, the lineal chart: Each root is said to be the *father* of the roots of its subtrees, and the latter are said to be *brothers*, and they are *sons* of their father. The root of the entire tree has no father. For example, in Fig. 18, C has three sons, D, E, and F; E is the father of G; B and C are brothers. Extension of this terminology (e.g., B is an uncle of F; A is the great-grandfather of G; H and F are first cousins) is clearly possible. Some authors use the feminine designations "mother, daughter, sister" instead of "father, son, brother"; but for some reason the masculine words seem more professional. Other authors, wishing to promote equality

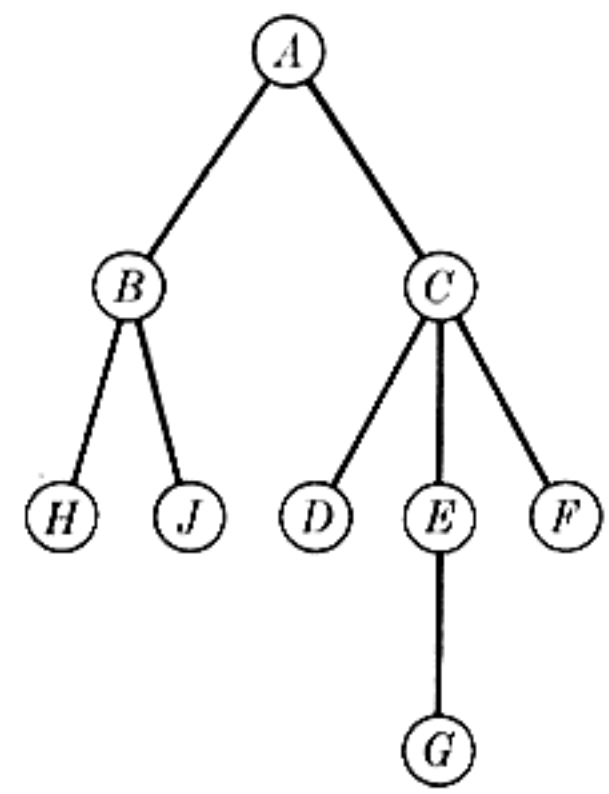


Fig. 18. Conventional tree diagram.

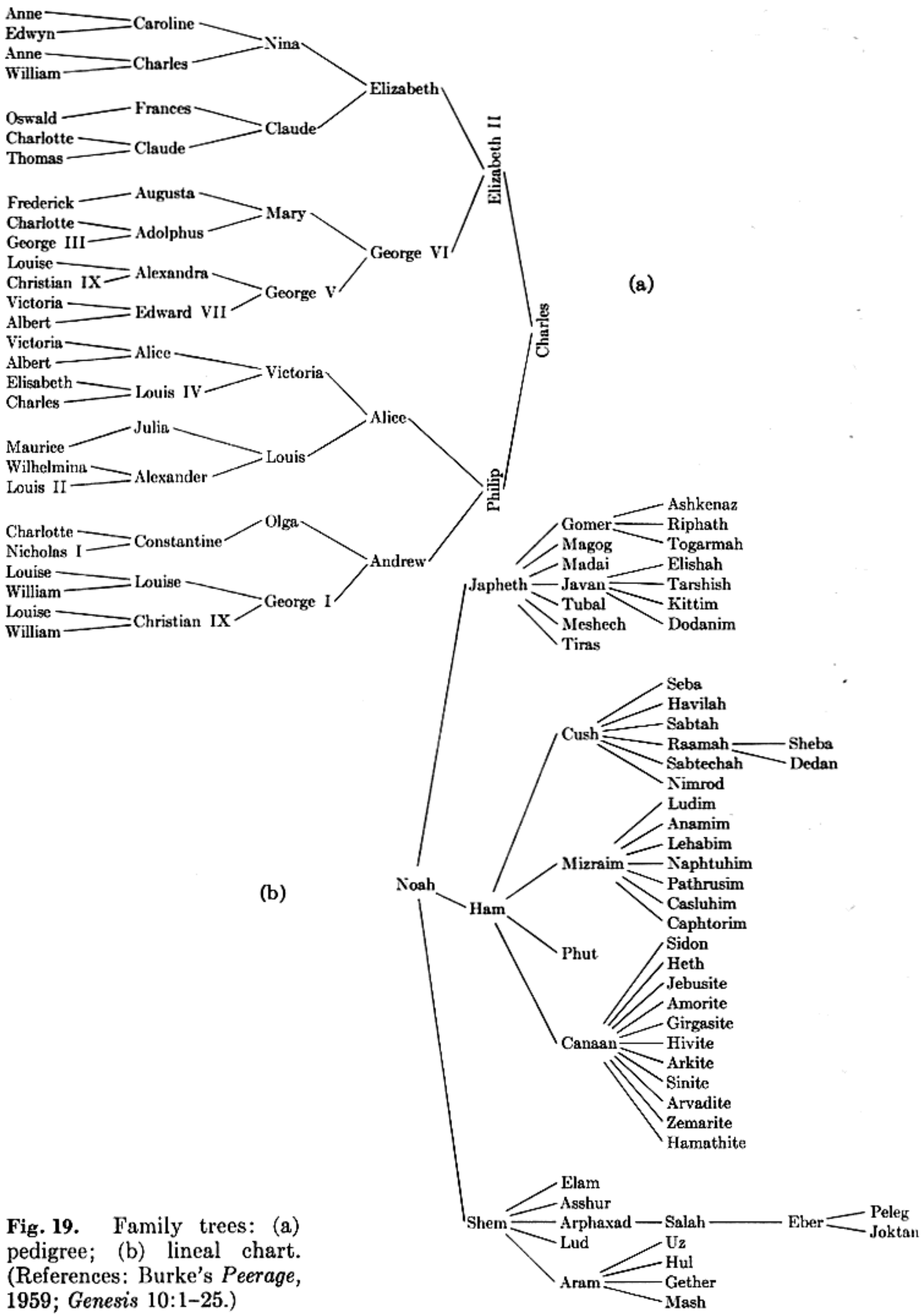
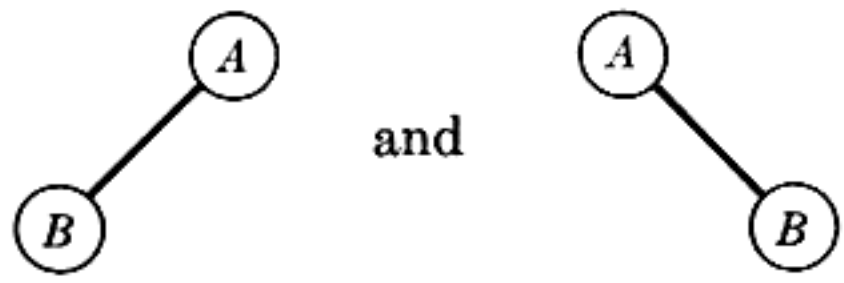


Fig. 19. Family trees: (a) pedigree; (b) lineal chart. (References: Burke's *Peerage*, 1959; *Genesis* 10:1–25.)

of the sexes, use the neutral words "parent, offspring, sibling" instead. In any case we use the words *ancestor* and *descendant* to denote a relationship that may span several levels of the tree: The descendants of C in Fig. 18 are D, E, F, and G; the ancestors of G are A, C, and E.

Figure 19(a) is an example of a *binary tree*, which is an important type of tree structure. The reader has undoubtedly seen other examples of binary trees in connection with tennis tournaments, etc. In a binary tree each node has at most two subtrees, and when there is only one subtree present, we distinguish between the left and right subtree. More formally, let us define a binary tree as *a finite set of nodes which either is empty, or consists of a root and two disjoint binary trees* called the left and right subtrees of the root.

This recursive definition of binary tree should be studied carefully. Note that a binary tree is *not* a special case of a tree; it is another concept entirely (although we will see many relations between the two concepts). For example, the binary trees



(1)

are distinct (the root has an empty left subtree in one case and a nonempty left subtree in the other), although as trees they would be identical. Therefore we will always be careful to use the word "binary" to distinguish between binary trees and ordinary trees. Some authors define binary tree in a slightly different manner (see exercise 20).

Tree structure can be represented graphically in several other ways bearing no resemblance to actual trees. Figure 20 shows three diagrams which reflect the structure of Fig. 18: Figure 20(a) essentially represents Fig. 18 as an *oriented* tree; this diagram is a special case of the general idea of *nested sets*, i.e., a collection of sets in which any pair of sets is either disjoint or one contains the other. (See exercise 10.) Part (b) of the figure shows nested sets in a line,

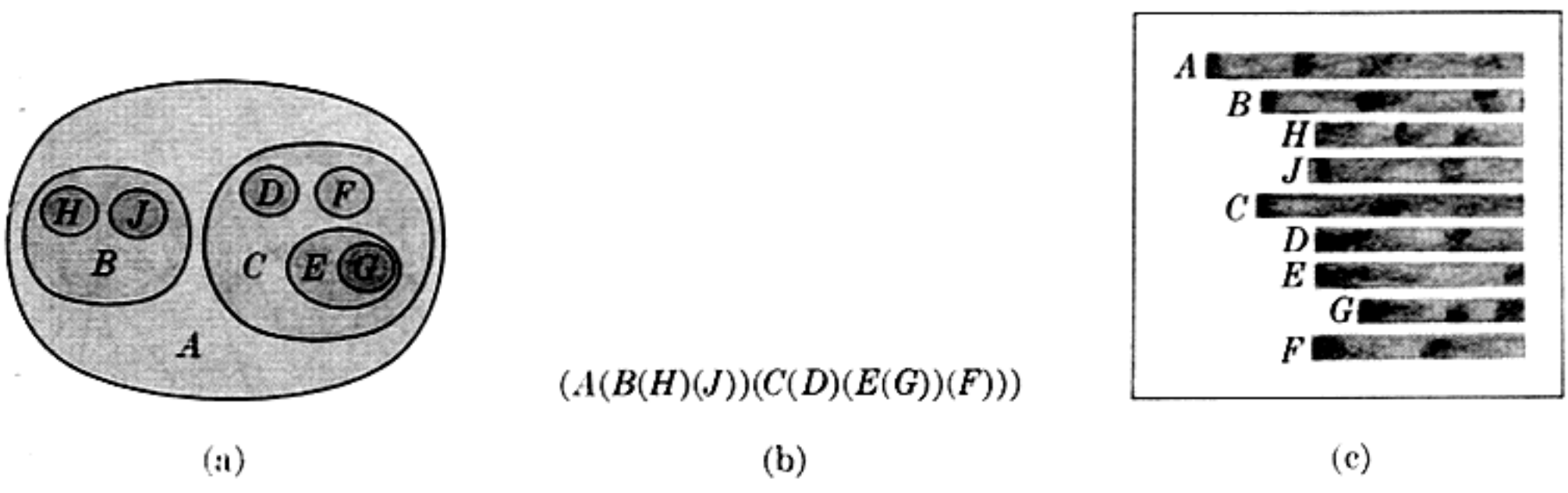


Fig. 20. Further ways to show tree structure: (a) Nested sets; (b) Nested parentheses; (c) Indentation.

much as part (a) shows them in a plane; in part (b) the ordering of the tree is also indicated. Part (b) may also be regarded as an outline of an algebraic formula involving "nested parentheses." Part (c) shows still another common way to represent tree structure, using *indentation*. The number of different representation methods in itself is ample evidence for the importance of tree structures in everyday life as well as in computer programming. Any hierarchical classification scheme leads to a tree structure.

The similarity between an indented list like Fig. 20(c) and outlines or tables of contents in books is noteworthy. This book itself has a tree structure: the tree structure of Chapter 2 is shown in Fig. 21. Here we notice a significant idea: *the method used to number sections in this book is another way to specify tree structure.* Such a method is often called "Dewey decimal notation" for trees, by analogy with the similar classification scheme of this name used in libraries. The Dewey decimal notation for the tree of Fig. 18 is

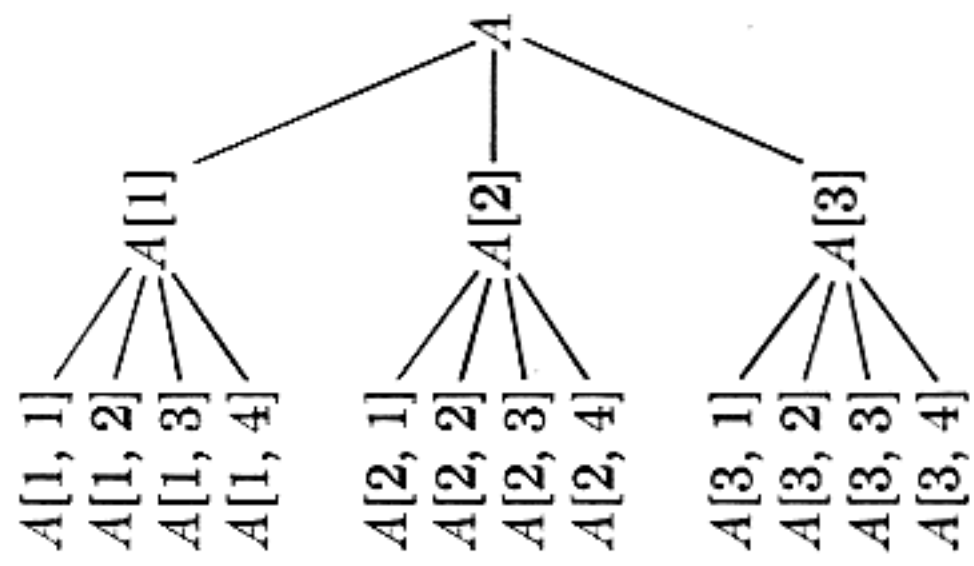
$$1\ A;\quad 1.1\ B;\quad 1.1.1\ H;\quad 1.1.2\ J;\quad 1.2\ C;\quad 1.2.1\ D;\quad 1.2.2\ E;$$
$$1.2.2.1\ G;\quad 1.2.3\ F.$$

Dewey decimal notation applies to any forest: the root of the kth tree in the forest is given number k; and if α is the number of any node of degree m, its sons are numbered $\alpha.1, \alpha.2, \ldots, \alpha.m$. The Dewey decimal notation satisfies many simple mathematical properties, and it is a useful tool in the analysis of trees. One example of this is the natural sequential ordering it gives to the nodes of an arbitrary tree, analogous to the ordering of sections within this book.

There is an intimate relation between Dewey decimal notation and the notation for indexed variables which we have already been using extensively. If F is a forest of trees, we may let $F[1]$ denote the first tree, $F[1][2] \equiv F[1, 2]$ the second subtree of the root of this first tree, $F[1, 2, 1]$ the first subtree of the latter, etc.; node $a.b.c.d$ in Dewey decimal notation is $\text{root}(F[a, b, c, d])$. This notation is an extension of the index notation in that the admissible range of each index depends on the values in the preceding index positions.

We see that, in particular, any rectangular array can be thought of as a special case of a tree structure. For example, here are two representations of a 3×4 matrix:

$$\begin{pmatrix} A[1, 1] & A[1, 2] & A[1, 3] & A[1, 4] \\ A[2, 1] & A[2, 2] & A[2, 3] & A[2, 4] \\ A[3, 1] & A[3, 2] & A[3, 3] & A[3, 4] \end{pmatrix}$$



It is important to observe, however, that this tree structure does not faithfully

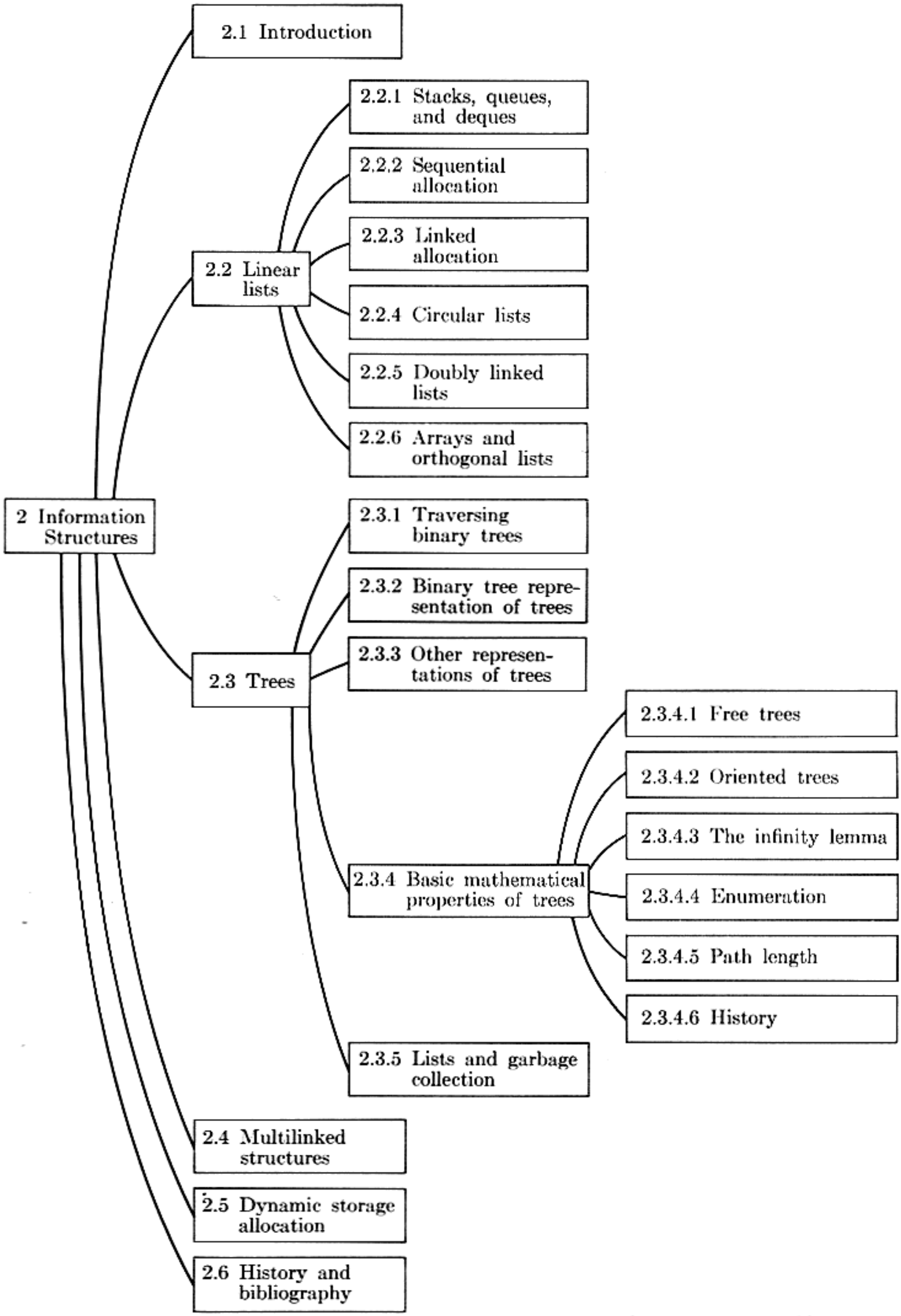


Fig. 21. The structure of Chapter 2.

reflect all of the matrix structure; the row relationships appear explicitly in the tree but the column relationships do not.

Algebraic formulas provide us with another example of tree structure. Figure 22 shows a tree corresponding to the arithmetic expression

$$a - b(c/d + e/f). \tag{2}$$

The connection between formulas and trees is very important in applications, as we shall see later (especially in Chapters 10 and 12).

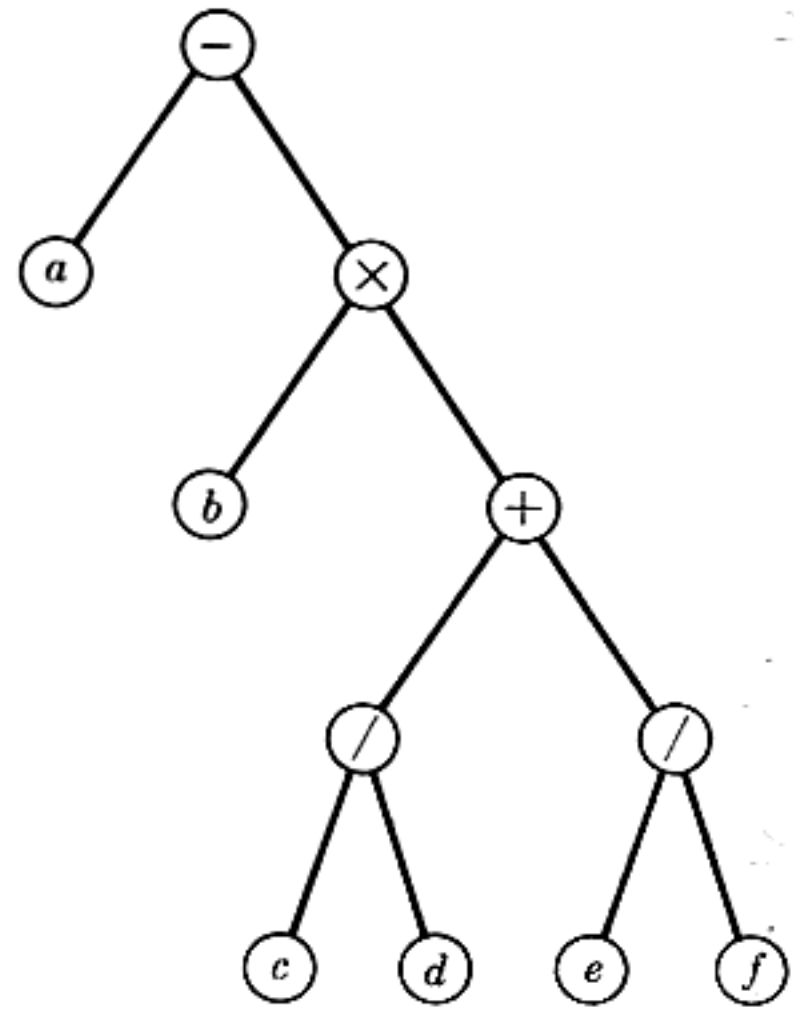


Fig. 22. Tree representing formula (2).

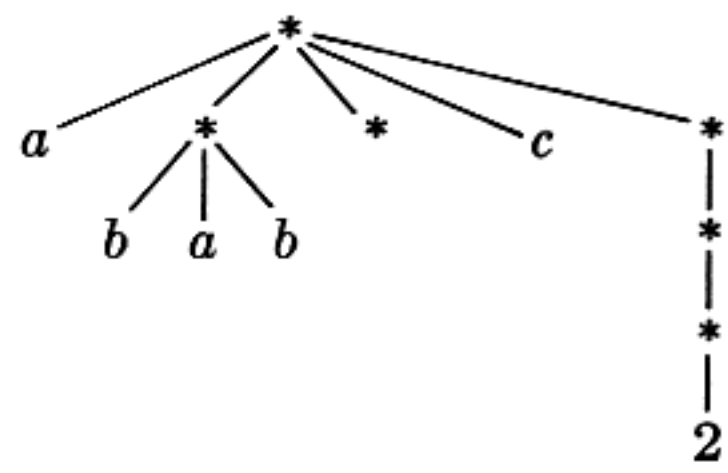
In addition to trees, forests, and binary trees, there is a fourth type of structure, commonly called a "list structure," which is closely related to both of the former two. The word "list" is being used here in a very technical sense, and to distinguish this use of the word we will always capitalize it, "List." A List is defined (recursively) as *a finite sequence of zero or more atoms or Lists.* Here "atom" is an undefined concept referring to elements from any universe of objects that might be desired, so long as it is possible to distinguish an atom from a List. By means of an obvious notational convention involving commas and parentheses, we can distinguish between atoms and Lists and we can conveniently display the ordering within a List. As an example, consider

$$L = (a,\ (b, a, b),\ (\ ),\ c,\ (((2)))) \tag{3}$$

which is a List with five elements: first the atom a, then the List (b, a, b), then the empty List "( )", then the atom c, and finally the List $(((2)))$. The latter List consists of the List $((2))$, which consists of the List (2), which consists of the atom 2.

The following tree structure corresponds to L:

$L =$



(4)

The asterisks in this diagram indicate the definition and appearance of a List,

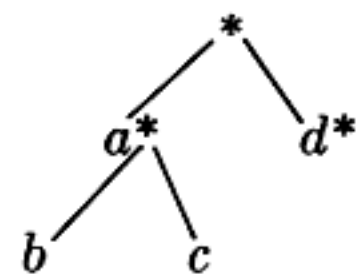
as opposed to the appearance of an atom. Index notation applies to Lists as it does to trees, e.g., $L[2] = (b, a, b)$, $L[2, 2] = a$.

No data is carried in the nodes for the Lists in (4) other than the fact that they are Lists. It is possible to label the elements of Lists with information, as we have done for trees and other structures; e.g.,

$$A = (a{:}(b, c), d{:}(\,))$$

would correspond to a tree we can draw as follows:

$A =$



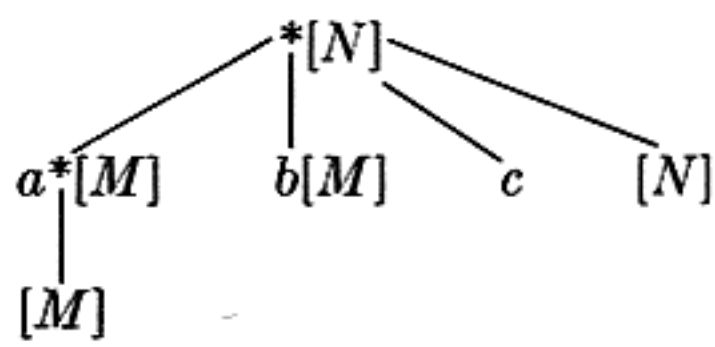
The big difference between Lists and trees is that Lists may overlap (i.e., sub-Lists need not be disjoint) and they may even be recursive (may contain themselves). The List

$$M = (M) \tag{5}$$

corresponds to no tree structure, nor does the List

$$N = (a{:}M, b{:}M, c, N). \tag{6}$$

(In these examples, capital letters refer to Lists, lower case letters to labels and atoms.) We might diagram (5) and (6) as follows:



(7)

Actually, Lists are not so complicated as the above examples might indicate; they are, in essence, a rather simple generalization of linear lists such as we have considered in Section 2.2, with the additional proviso that the elements of the linear lists may be link variables which point to other linear lists (and possibly to themselves).

Summary: Four kinds of closely related information structures (trees, forests, binary trees, Lists) arise from many sources, and they are therefore important in computer algorithms. We have seen various methods of diagraming these structures, and we have considered some notations and terminology which is useful in talking about them. The following sections develop these ideas in greater detail.

EXERCISES

1. [*18*] How many different trees are there with three nodes, A, B, and C?

2. [*20*] How many different *oriented* trees are there with three nodes, A, B, and C?

▶ 3. [*M20*] Prove rigorously from the definitions that for every node X in a tree there is a unique "path up to the root," i.e., a unique sequence of $k \geq 1$ nodes $X_1, X_2, \ldots, X_k$ such that X_1 is the root of the tree, $X_k = X$, and X_j is the father of X_{j+1} for $1 \leq j < k$. (This proof will be typical of the proofs of nearly all the elementary facts about tree structures.) [*Hint:* Use induction on the number of nodes in the tree.]

4. [*01*] True or false: In a conventional tree diagram (root at the top), if node X has a *higher* level number than node Y, then node X appears *lower* in the diagram than node Y.

5. [*02*] If node A has three brothers and B is the father of A, what is the degree of B?

▶ 6. [*21*] Define the statement "X is an mth cousin of Y, n times removed" as a meaningful relation between nodes X and Y of a tree, by analogy with family trees, if $m > 0$ and $n \geq 0$. (See a dictionary for the meaning of these terms in regard to family trees.)

7. [*23*] Extend the definition given in the previous exercise to all $m \geq -1$ and to all integers $n \geq -(m+1)$ in such a way that for any two nodes X and Y of a tree there are unique m and n such that X is an mth cousin of Y, n times removed.

8. [*03*] What binary tree is not a tree?

9. [*00*] In the two binary trees of (1), which node is the root (B or A)?

10. [*M20*] A collection of nonempty sets is said to be *nested* if, given any pair X, Y of the sets, either $X \subseteq Y$ or $X \supseteq Y$ or X and Y are disjoint. (In other words, $X \cap Y$ is either X, Y, or $\emptyset$.) Figure 20(a) indicates that any tree corresponds to a collection of nested sets; conversely, does every such collection correspond to a tree?

▶ 11. [*HM32*] Extend the definition of tree to infinite trees by considering collections of nested sets as in exercise 10. Can the concepts of level, degree, father, and son be defined for each node of an infinite tree? Give examples of nested sets of real numbers which correspond to a tree in which

a) every node has uncountable degree and there are infinitely many levels;
b) there are nodes with uncountable level;
c) every node has degree at least 2 and there are uncountably many levels.

12. [*M23*] Under what conditions does a partially ordered set (cf. Section 2.2.3) correspond to an unordered tree or forest?

13. [*10*] Suppose that node X is numbered $a_1 . a_2 . \cdots . a_k$ in the Dewey decimal system; what are the numbers of the nodes in the path from X to the root (see exercise 3)?

14. [*M22*] Let S be any nonempty set of elements having the form "$1 . a_1 . \cdots . a_k$", where $k \geq 0$ and $a_1, \ldots, a_k$ are positive integers. Show that S specifies a tree when it is finite and satisfies the following condition: "If $\alpha . m$ is in the set, then so is

$\alpha \cdot (m - 1)$ if $m > 1$, or α if $m = 1$." (This condition is clearly satisfied in the Dewey decimal notation for a tree, so as a consequence of this exercise it is another way to characterize tree structure.)

▶ **15.** [*20*] Invent a notation for binary trees corresponding to the Dewey decimal notation for trees.

16. [*20*] Draw trees analogous to Fig. 22 corresponding to the arithmetic expressions

a) "$2(a - b/c)$";
b) "$a + b + 5c$".

17. [*01*] If T is the tree of Fig. 18, what node is root(T[2, 2])?

18. [*08*] In List (3), what is L[5, 1, 1]? What is L[3, 1]?

▶ **19.** [*15*] Draw a List diagram analogous to (7) for the List $L = (a, (L))$. What is L[2] in this list? What is L[2, 1, 1]?

20. [*M21*] Define a "b-tree" as a tree in which each node has exactly zero or two sons. (Formally, a "b-tree" consists of a single node, called its root, plus 0 or 2 disjoint b-trees.) Show that every b-tree has an odd number of nodes; and give a one-to-one correspondence between binary trees with n nodes and (ordered) b-trees with $2n + 1$ nodes.

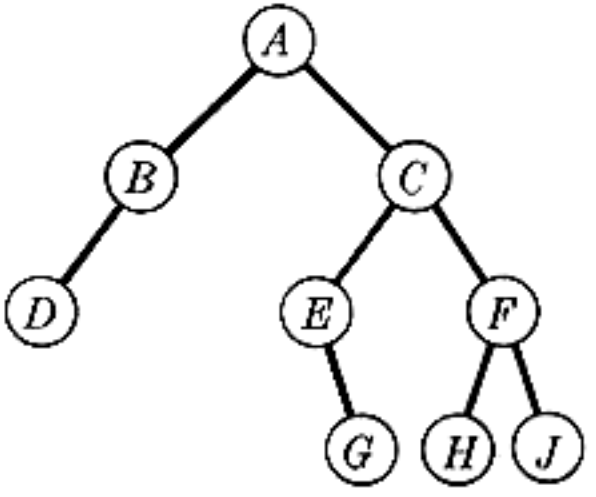
21. [*M22*] If a tree has n_1 nodes of degree 1, n_2 nodes of degree 2, . . . , n_m nodes of degree m, then how many terminal nodes does it have?

22. [*45*] Develop a computer system to display tree structures graphically on a cathode ray tube, with facilities for on-line manipulation of the structures.

2.3.1. Traversing Binary Trees

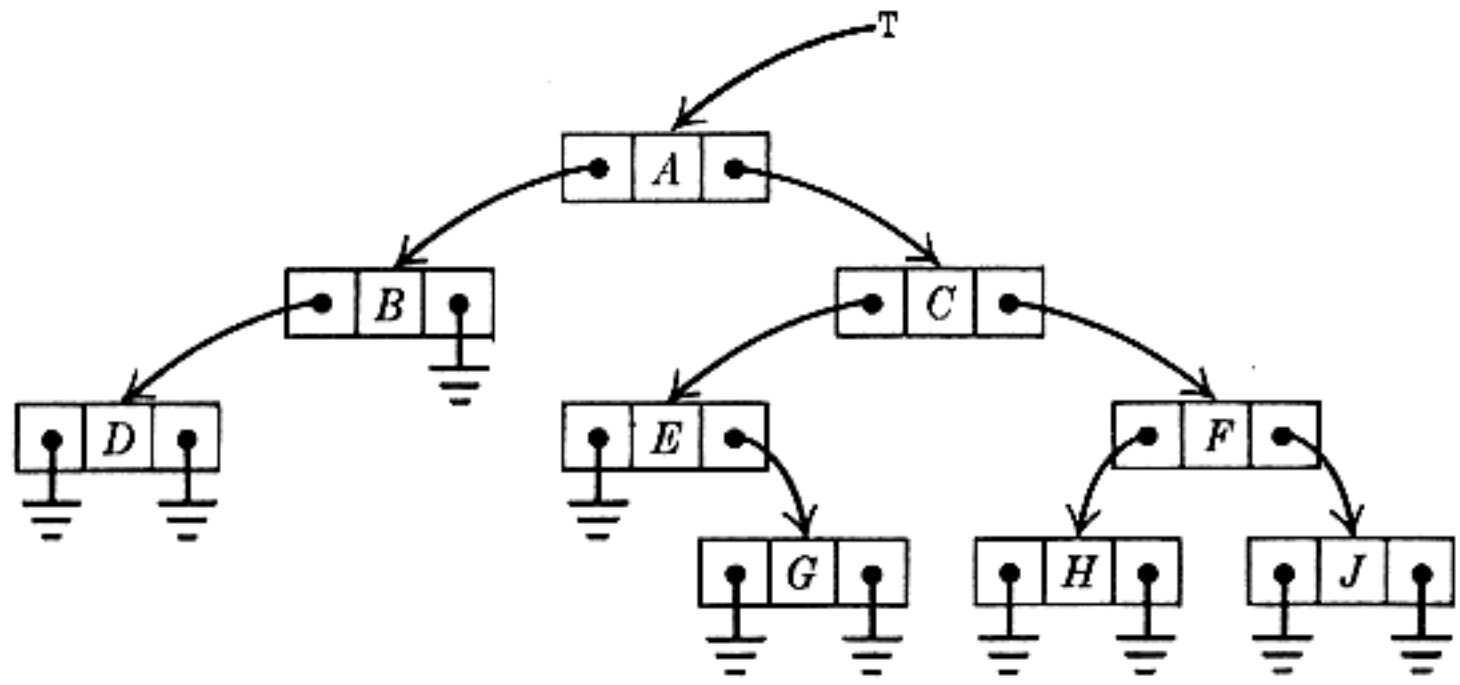
It is important to acquire a good understanding of the properties of binary trees before making further investigations of trees, since general trees are usually represented in terms of some equivalent binary tree inside a computer.

We have defined a binary tree as a finite set of nodes that either is empty, or consists of a root together with two binary trees. This definition suggests a natural way to represent binary trees within a computer: We may have two links, `LLINK` and `RLINK`, within each node, and a link variable `T` which is a "pointer to the tree." If the tree is empty, `T` = Λ; otherwise `T` is the address of the root of the tree, and `LLINK(T)`, `RLINK(T)` are pointers to the left and right, respectively, subtrees of the root. These rules recursively define the memory representation of any binary tree; for example,



(1)

is represented by



(2)

This simple and natural memory representation accounts for the special importance of binary tree structures. Besides the fact that general trees are conveniently representable as binary trees, many trees that arise in applications are themselves inherently binary, so binary trees are of interest in their own right.

There are many algorithms for manipulation of tree structures, and one idea that occurs repeatedly in these algorithms is the notion of *traversing* or "walking through" a tree. This is a method of examining the nodes of the tree systematically so that each node is visited exactly once. A complete traversal of the tree gives us a linear arrangement of the nodes, and many algorithms are facilitated if we can talk about the "next" node following or preceding a given node in such a sequence.

Three principal ways may be used to traverse a binary tree: we can visit the nodes in *preorder*, *postorder*, or *endorder*. These three methods are defined recursively; when the binary tree is empty, it is "traversed" by doing nothing, and otherwise the traversal proceeds in three steps:

Preorder traversal	Postorder traversal
Visit the root	Traverse the left subtree
Traverse the left subtree	Visit the root
Traverse the right subtree	Traverse the right subtree

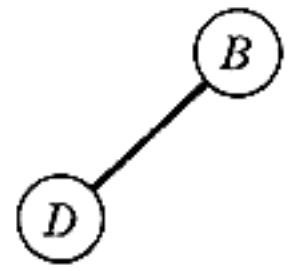
Endorder traversal

Traverse the left subtree
Traverse the right subtree
Visit the root

If we apply these definitions to the binary tree (2), we find that the nodes in preorder are

$$A\ B\ D\ C\ E\ G\ F\ H\ J. \tag{3}$$

(First comes the root A, then comes the left subtree



in preorder, and finally comes the right subtree in preorder.) In postorder we visit the root between visits to the nodes of each subtree, essentially as though the nodes were "projected" down onto a single horizontal line, and this gives the order

$$D\ B\ A\ E\ G\ C\ H\ F\ J. \tag{4}$$

The endorder for the nodes of this binary tree is, similarly,

$$D\ B\ G\ E\ H\ J\ F\ C\ A.$$

We will see that these three ways of arranging the nodes of a binary tree into a sequence are extremely important, as they are intimately connected with most of the computer methods dealing with trees. (*Note:* The names *preorder* and *postorder* come from the fact that each root is visited before or after its left subtree; this terminology is motivated by the special significance given to left subtrees when binary trees are used to represent general trees, as explained in the following section. In many applications of binary trees, there is more symmetry between the meanings of left subtrees and right subtrees, and in such cases the term *symmetric order* is used as a synonym for postorder. Clearly, postorder, which puts the root in the middle, is essentially symmetric between left and right: if the tree is reflected about a vertical axis, the symmetric order is simply reversed.)

A recursively stated definition, such as the one just given for the three basic orders, must be reworked in order to make it directly applicable to computer implementation. General methods for doing this are discussed in Chapter 8; we usually make use of an auxiliary stack, as in the following algorithm:

Algorithm T (*Traverse binary tree in postorder*). Let T be a pointer to a binary tree having a representation as in (2); this algorithm visits all the nodes of the binary tree in postorder, making use of an auxiliary stack A.

T1. [Initialize.] Set stack A empty, and set the link variable P ← T.

T2. [P = Λ?] If P = Λ, go to step T4.

T3. [Stack ⇐ P.] (Now P points to a nonempty binary tree which is to be traversed.) Set A ⇐ P, i.e., push the value of P onto stack A. (See Section 2.2.1.) Then set P ← LLINK(P) and return to step T2.

T4. [P ⇐ Stack.] If stack A is empty, the algorithm terminates; otherwise set P ⇐ A.

T5. [Visit P.] "Visit" NODE(P). (See below.) Then set P ← RLINK(P) and return to step T2. ▮

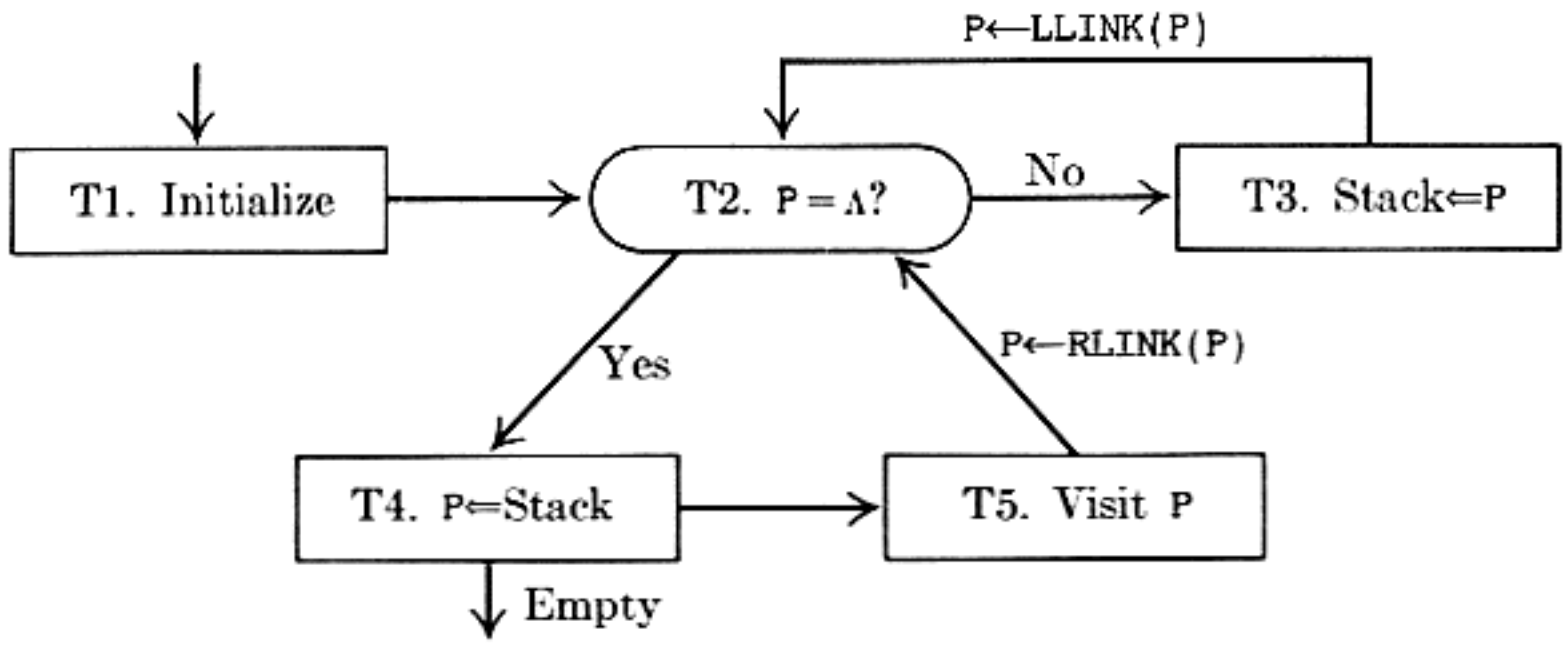


Fig. 23. Flow chart for Algorithm T.

In the final step of this algorithm, the word "visit" means we do whatever activity is intended as the tree is being traversed. Algorithm T runs like a coroutine with respect to this other activity: the main program activates this coroutine whenever it wants P to move from one node to its postorder successor. Of course, since this coroutine calls the main routine in only one place, it is not much different from a subroutine (see Section 1.4.2). Algorithm T assumes that the external activity deletes neither NODE(P) nor any of its ancestors from the tree.

If the reader will now attempt to play through Algorithm T using the tree (2) as a test case, he will easily see the reasons behind the procedure: When we get to step T3, we want to traverse the binary tree whose root is indicated by pointer P. The idea is to save P on a stack and then to traverse the left subtree; when this has been done, we will get to step T4 and will find the old value of P on the stack again. After visiting the root, NODE(P), in step T5, the remaining job is to traverse the right subtree.

Since Algorithm T is typical of so many other algorithms we will see later, it is worth while giving a formal proof of the remarks made in the preceding paragraph. Let us now attempt to *prove* that Algorithm T traverses a binary tree of n nodes in postorder, by using induction on n. Our goal is readily established if we can prove a slightly more general result:

> "Starting at step T2 with P a pointer to a binary tree of n nodes and with the stack A containing A[1] . . . A[m] for some $m \geq 0$, the procedure of steps T2–T5 will traverse the binary tree in question in postorder and will then arrive at step T4 with stack A returned to its original value A[1] . . . A[m]."

This statement is obviously true when $n = 0$, because of step T2. If $n > 0$, let P_0 be the value of P upon entry to step T2. Since $P_0 \neq \Lambda$, we will perform step T3, which means that stack A is changed to A[1] . . . A[m] P_0 and P is set to

LLINK(P_0). Now the left subtree has less than n nodes, so by induction we will traverse the left subtree in postorder and will ultimately arrive at step T4 with A[1] . . . A[m] P_0 on the stack. Step T4 returns the stack to A[1] . . . A[m] and sets P $\leftarrow$ P_0. Step T5 now visits NODE(P_0) and sets P $\leftarrow$ RLINK(P_0). Now the right subtree has less than n nodes, so by induction we will traverse the right subtree in postorder and arrive at step T4 as required. The tree has been traversed in postorder, by the definition of that order. This completes the proof.

An almost identical algorithm may be formulated which traverses binary trees in preorder (see exercise 12). It is slightly more difficult to achieve the traversal in endorder (see exercise 13) and for this reason endorder is not quite as important as the others.

It is convenient to define a new notation for the successors and predecessors of nodes in these various orders. If P points to a node of a binary tree, let

$$\begin{aligned}
\texttt{P*} &= \text{address of successor of NODE(P) in preorder;}\\
\texttt{P\$} &= \text{address of successor of NODE(P) in postorder;}\\
\texttt{P\#} &= \text{address of successor of NODE(P) in endorder;}\\
\texttt{*P} &= \text{address of predecessor of NODE(P) in preorder;}\\
\texttt{\$P} &= \text{address of predecessor of NODE(P) in postorder;}\\
\texttt{\#P} &= \text{address of predecessor of NODE(P) in endorder.}
\end{aligned} \tag{5}$$

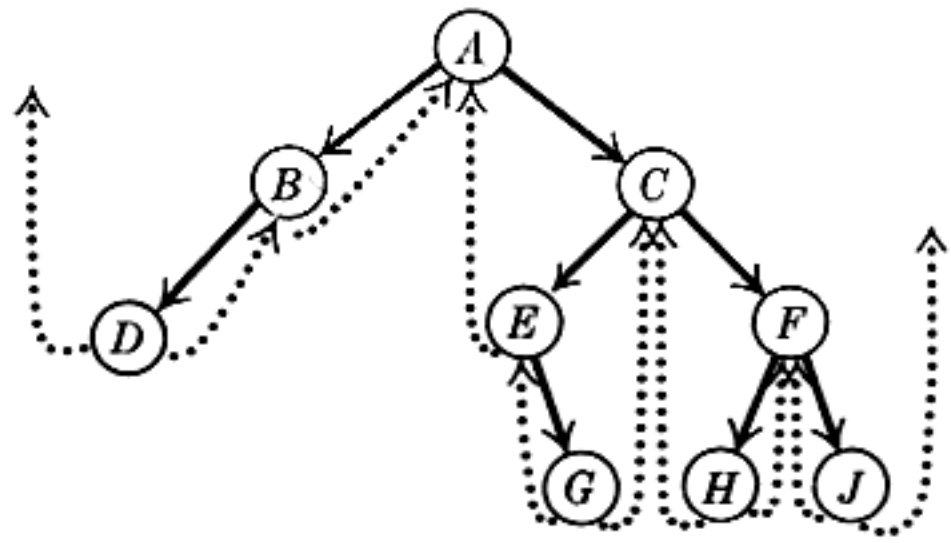
If there is no such successor or predecessor of NODE(P), the value LOC(T) is generally used, where T is a pointer to the tree in question. We have *(P*) = (*P)* = P, $(P$) = ($P)$ = P, etc. As an example of this notation, let INFO(P) be the letter shown in NODE(P) in the tree (2); then if P points to the root, we have INFO(P) = A, INFO(P*) = B, INFO(P$) = E, INFO($P) = B, INFO(#P) = C, and P# = *P = LOC(T).

At this point the reader will perhaps experience a feeling of insecurity about the intuitive meanings of P*, P$, etc. As we proceed further, the ideas will gradually become clearer; exercise 16 at the end of this section may also be of help.

There is an important alternative to the memory representation of binary trees given in (2), which is somewhat analogous to the difference between circular lists and straight one-way lists. Note that there are more null links than other pointers in the tree (2), and indeed this is true of any binary tree represented by that method (see exercise 14). This seems to be wasteful of memory space, and there are various ways to make use of memory more efficiently; for example, we could store two "tag" indicators with each node, which tell in just two bits of memory whether or not the LLINK or RLINK, or both, are null, and the memory space for terminal links can be used for other purposes.

An ingenious use of this extra memory space has been suggested by A. J. Perlis and C. Thornton, who devised the so-called *threaded* tree representation. In this method, terminal links are replaced by "threads" to other parts of the

tree, as an aid to traversing the tree. The threaded tree equivalent to (2) is



(6)

Here dotted lines represent the "threads," which always go to a higher node of the tree. *Every* node now has two links: some nodes, like C, have two ordinary links to left and right subtrees; other nodes, like H, have two thread links, and some nodes have one link of each type. The special threads emanating from D and J will be explained later; they appear in the "leftmost" and "rightmost" nodes.

In the memory representation of a threaded binary tree it is necessary to distinguish between the dotted and solid links; this is done as suggested above by two additional one-bit fields in each node, LTAG and RTAG. The threaded representation may be precisely defined as follows:

Unthreaded representation	Threaded representation
LLINK(P) $= \Lambda$	LTAG(P) = "$-$", LLINK(P) = \$P
LLINK(P) $=$ Q $\neq \Lambda$	LTAG(P) = "$+$", LLINK(P) = Q
RLINK(P) $= \Lambda$	RTAG(P) = "$-$", RLINK(P) = P\$
RLINK(P) $=$ Q	RTAG(P) = "$+$", RLINK(P) = Q

In some algorithms it can be guaranteed that the root of any subtree always will appear in a lower memory location than the other nodes of the subtree. Thus LTAG(P) will be "$-$" if and only if LLINK(P) $<$ P, so that LTAG (and similarly RTAG) is redundant.

According to the definition above, each new thread link points directly to the predecessor or successor of the node in question, in postorder (symmetric order). Figure 24 illustrates the general orientation of thread links in any binary tree.

The great advantage of threaded trees is that the traversal algorithms become simpler. For example, the following algorithm calculates P\$, given P:

Algorithm S (*Symmetric (postorder) successor in a threaded binary tree*). If P points to a node of a threaded binary tree, this algorithm sets Q $\leftarrow$ P\$.

S1. [RLINK(P) a thread?] Set Q $\leftarrow$ RLINK(P). If RTAG(P) = "$-$", terminate the algorithm.

S2. [Search to left.] If LTAG(Q) = "$+$", set Q $\leftarrow$ LLINK(Q) and repeat this step. Otherwise the algorithm terminates. ∎

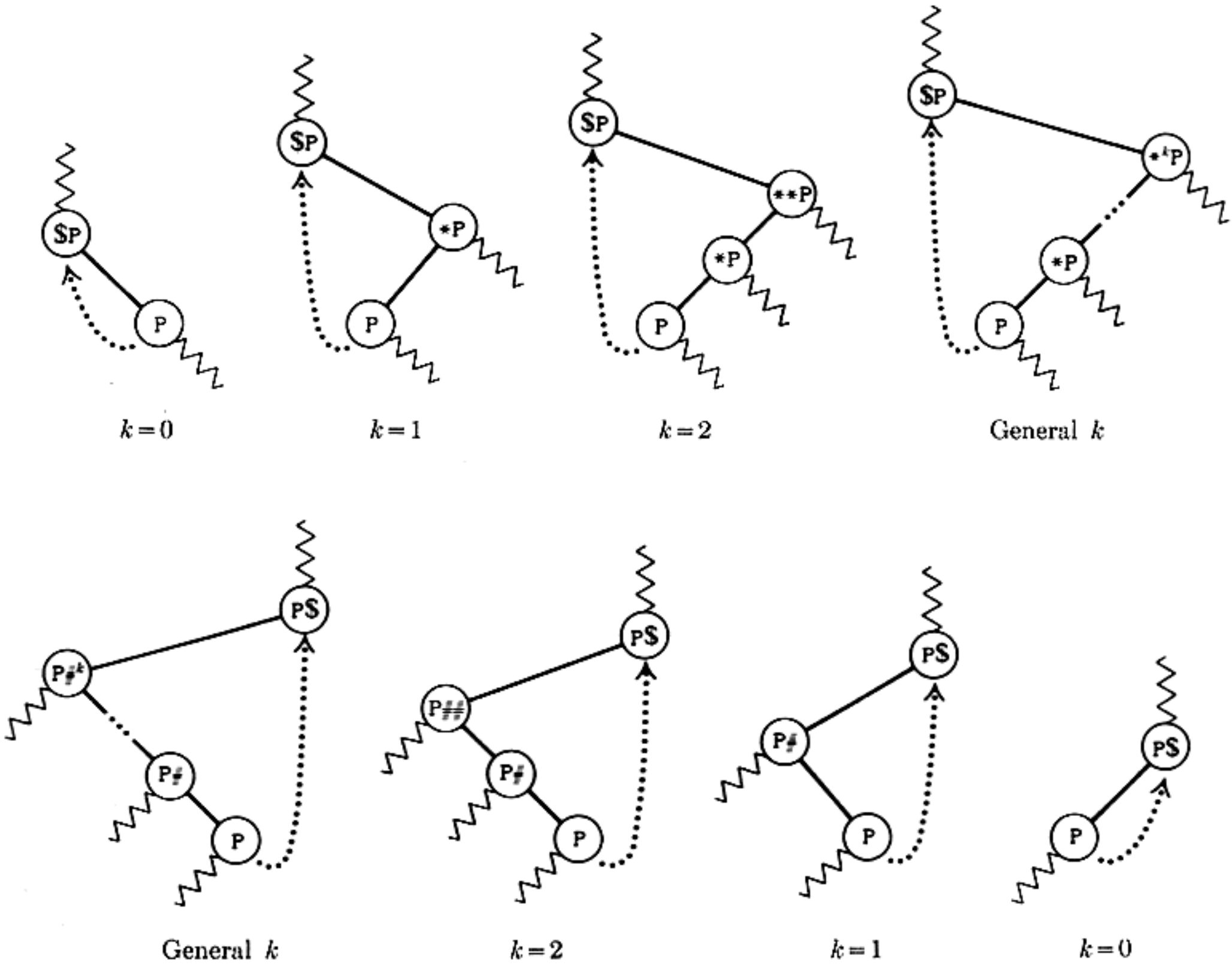


Fig. 24. General orientation of left and right thread links in a binary tree. "∧∧" lines indicate links or threads to other parts of the tree.

Note that no stack is needed here to accomplish what was done using a stack in Algorithm T. In fact, the ordinary representation (2) makes it impossible to find P$ efficiently, given only the address of a random point P in the tree; since no links point upward in the unthreaded representation, there is no clue to what nodes are above a given node, unless we retain a history of how we reached that point (and that is essentially the stack in Algorithm T).

We claim that Algorithm S *is* "efficient," although this property is not immediately obvious, since step S2 can be executed any number of times. In view of the loop in step S2, would it perhaps be faster to use a stack after all, as Algorithm T does? To investigate this question, we will consider the average number of times step S2 must be performed if P is a "random" point in the tree; or what is the same, we will determine the total number of times step S2 is performed if Algorithm S is used repeatedly to traverse an entire tree.

At the same time as this analysis is being carried out, it will be instructive to study MIX programs for both Algorithms S and T. The following programs

assume that the nodes have the two-word form

LTAG	LLINK	INFO1
RTAG	RLINK	INFO2

.

In an unthreaded tree, LTAG and RTAG will always be "+" and terminal links will be represented by zero. The abbreviations LLINKT and RLINKT will be used to stand for the combined LTAG–LLINK and RTAG–RLINK fields, respectively.

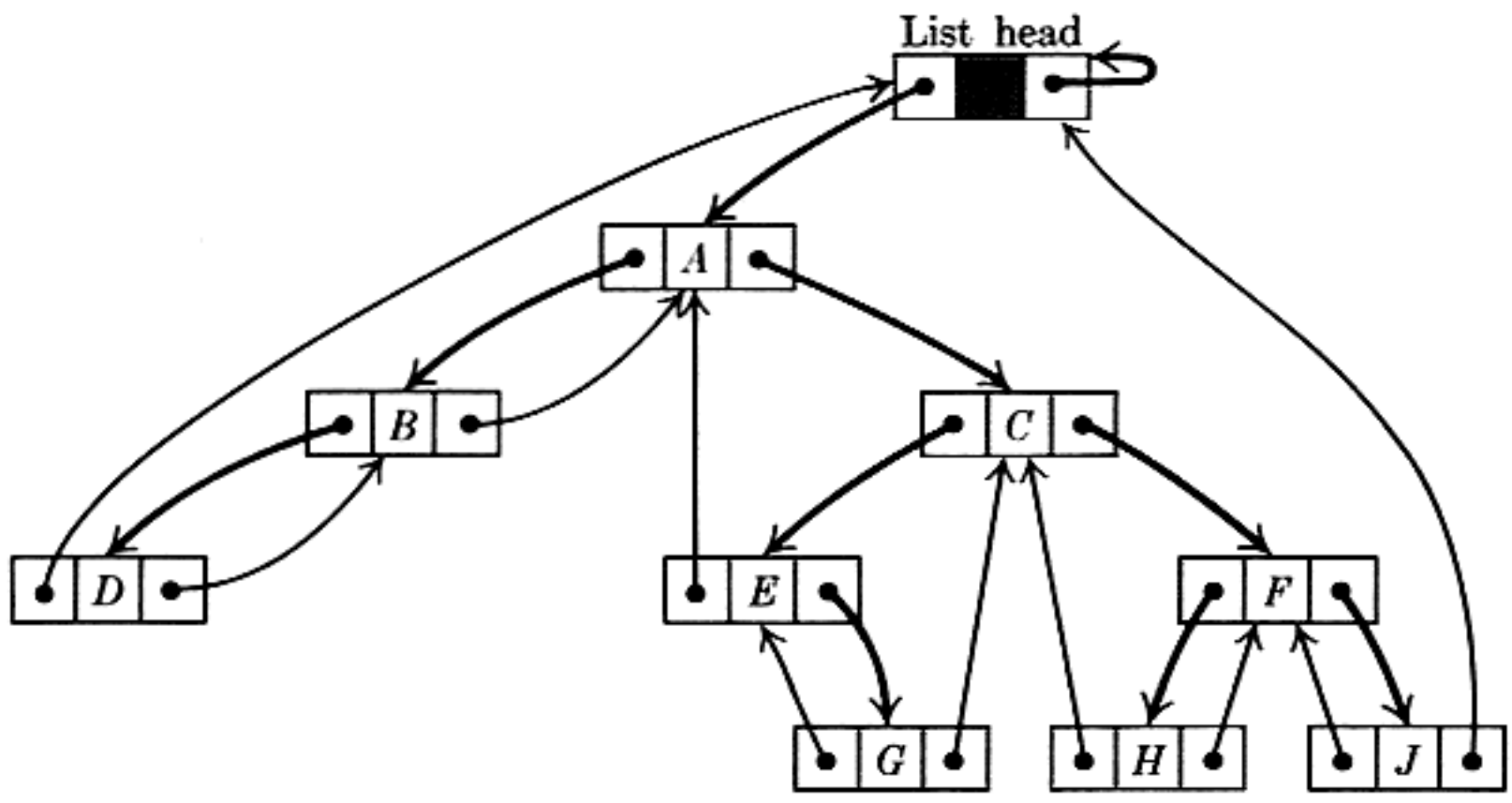
As usual, we should be careful to set up all of our algorithms so that they work properly with empty trees; and if T is the pointer to the tree, we would like to have LOC(T)* and LOC(T)$ be the *first* nodes in prefix or symmetric order, respectively. For threaded trees, it turns out that things will work nicely if NODE(LOC(T)) is made into a "list head" for the tree, with

$$\texttt{LLINKT(HEAD)} \equiv \texttt{T}, \quad \texttt{RLINK(HEAD)} = \texttt{HEAD}, \quad \texttt{RTAG(HEAD)} = \text{“+”}. \tag{7}$$

(HEAD denotes LOC(T), the address of the list head.) An empty threaded tree will satisfy the conditions

$$\texttt{LLINK(HEAD)} = \texttt{HEAD}, \quad \texttt{LTAG(HEAD)} = \text{“}-\text{”}. \tag{8}$$

The tree grows by having nodes inserted to the *left* of the list head. (These initial conditions are primarily dictated by the algorithm to compute P*, which appears in exercise 17.) In accordance with these conventions, the computer representation for tree (1), as a threaded tree, is



(9)

With these preliminaries out of the way, we are now ready to consider MIX programs for Algorithms S and T. The following two programs traverse a binary tree in symmetric order (i.e., postorder), jumping to location VISIT periodically with index register 5 pointing to the node that is currently of interest.

Program T. In this implementation of Algorithm T, the stack is kept in locations A + 1, A + 2, . . . , A + MAX; and OVERFLOW occurs if the stack grows too large. rI6 is the stack pointer and rI5 ≡ P. The program has been rearranged slightly from Algorithm T (step T2 appears twice) so that the test for an empty stack need not be made when going directly from T3 to T2 to T4.

```
01   LLINK  EQU   1:2
02   RLINK  EQU   1:2
03   T1     LD5   HEAD(LLINK)     1     T1. Initialize. Set P ← T.
04          J5Z   DONE            1     Stop if P = Λ.
05          ENT6  0               1
06   T3     DEC6  MAX             n     T3. Stack ⇐ P.
07          J6NN  OVERFLOW        n     Has stack reached capacity?
08          INC6  MAX+1           n     If not, increase stack pointer.
09          ST5   A,6             n     Store P in stack.
10          LD5   0,5(LLINK)      n     P ← LLINK(P).
11   T2A    J5NZ  T3              n     To T3 if P ≠ Λ.
12   T4     LD5   A,6             n     T4. P ⇐ Stack.
13          DEC6  1               n     Decrease stack pointer.
14   T5     JMP   VISIT           n     T5. Visit P.
15          LD5   1,5(RLINK)      n     P ← RLINK(P).
16   T2     J5NZ  T3              n     T2. P = Λ?
17          J6NZ  T4              a     Test if stack is empty.
18   DONE   ...                         ∎
```

Program S. Algorithm S has been augmented with initialization and termination conditions to make this program comparable to Program T.

```
01   LLINKT EQU   0:2
02   RLINKT EQU   0:2
03   S0     ENT6  HEAD            1       S0. Initialize. Set Q ← HEAD.
04          JMP   S2              1
05   S3     JMP   VISIT           n       S3. Visit P.
06   S1     LD5N  1,5(RLINKT)     n       S1. RLINK(P) a thread?
07          J5NN  1F              n       Jump if RTAG(P) = "—".
08          ENN6  0,5             n − a   Otherwise set Q ← RLINK(P).
09   S2     ENT5  0,6             n + 1   Set P ← Q.
10          LD6   0,5(LLINKT)     n + 1   Q ← LLINKT(P).
11          J6P   *-2             n + 1   If LTAG(P) = "+", repeat.
12   1H     ENT6  -HEAD,5         n + 1
13          J6NZ  S3              n + 1   Visit unless P = HEAD. ∎
```

An analysis of the running time appears with the above code. These quantities are easy to determine using Kirchhoff's law and the facts that

i) In Program T, the number of insertions onto the stack must equal the number of deletions;

ii) In Program S, the LLINK and RLINK of each node is examined precisely once;

iii) The number of "visits" is the number of nodes in the tree.

The analysis tells us Program T takes $15n + a + 3$ units of time, and Program S takes $11n - a + 8$ units, where n is the number of nodes in the tree, and a is the number of terminal right links (nodes with no right subtree). The quantity a can be as low as 1, assuming that $n \neq 0$, and it can be as high as n; and if left and right are symmetrical, the average value of a is $(n + 1)/2$ as a consequence of facts proved in exercise 14.

The principal conclusions we may reach on the basis of this analysis are that

i) Step S2 of Algorithm S is performed only *once* on the average per execution of that algorithm, if P is a random node of the tree.

ii) Traversal is slightly faster for threaded trees; in the case of MIX the ratio of execution times is about 3 to 2.

iii) Algorithm T needs more memory space than Algorithm S because of the auxiliary stack required. In Program T we kept the stack in consecutive memory locations, and, consequently, it was necessary to put an arbitrary bound on its size. It would be very embarrassing if this bound were exceeded, so it must be set reasonably large (see exercise 10); thus the memory requirement of Program T is significantly more than Program S. Not infrequently a complex computer application will be independently traversing several trees at once, and a separate stack will be needed for each tree under Program T. This suggests that Program T might use linked allocation for its stack (see exercise 20); its execution time then becomes $30n + a + 3$ units, roughly twice as slow as before, although this may not be terribly important when the execution time for the other coroutine is added in. Still another alternative is to keep the stack links within the tree itself in a very tricky way, as discussed in exercise 21.

iv) Algorithm S is, of course, more general than Algorithm T, since it allows us to go from P to P$ when we are not necessarily traversing the entire binary tree.

So a threaded binary tree is decidedly superior to an unthreaded one, with respect to traversal. These advantages are offset in some applications by the slightly increased time needed to insert and delete nodes in a threaded tree. It is also sometimes possible to save memory space by "sharing" common subtrees with an unthreaded representation, while threaded trees require adherence to a strict tree structure with no overlapping of subtrees.

Thread links can also be used to compute P*, $P, and #P with efficiency comparable to that of Algorithm S. The functions *P and P# are slightly harder to compute, just as they are for unthreaded tree representations. The reader is urged to work exercise 17.

Most of the usefulness of threaded trees would disappear if it were hard to set up the thread links in the first place. The fact which makes the idea really work is that threaded trees grow almost as easily as ordinary ones do. We have the following algorithm:

Algorithm I (*Insertion into a threaded binary tree*). This algorithm attaches a single node, NODE(Q), as the right subtree of NODE(P), if the right subtree is empty (i.e., if RTAG(P) = "−"), and it inserts NODE(Q) between NODE(P) and NODE(RLINK(P)) otherwise. The binary tree in which the insertion takes place is assumed to be threaded as in (9); for a modification, see exercise 23.

I1. [Adjust tags and links.] Set RLINK(Q) ← RLINK(P), RTAG(Q) ← RTAG(P), RLINK(P) ← Q, RTAG(P) ← "+", LLINK(Q) ← P, LTAG(Q) ← "−".

I2. [Was RLINK(P) a thread?] If RTAG(Q) = "+", set LLINK(Q$) ← Q. (Here Q$ is determined by Algorithm S, which will work properly even though LLINK(Q$) now points to NODE(P) instead of NODE(Q). This step is necessary only when inserting into the midst of a threaded tree instead of merely inserting a "new leaf.") ▮

By reversing the roles of left and right (in particular, by replacing Q$ by $Q in step I2), we obtain an algorithm which inserts to the left in a similar way.

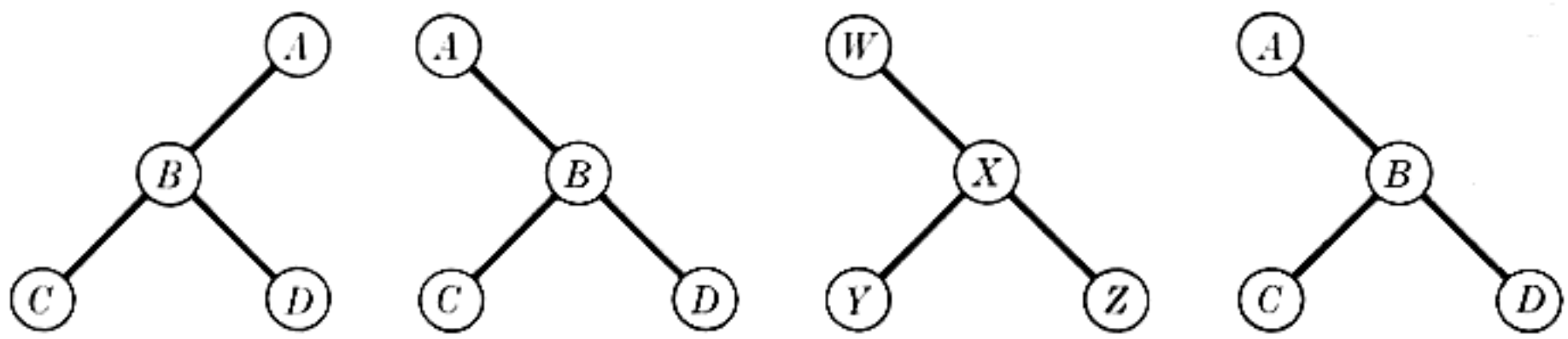
Our discussion of threaded binary trees so far has made use of thread links both to the left and to the right. There is an important middle ground between the completely unthreaded and completely threaded methods of representation: A *right-threaded binary tree* combines the two approaches by making use of threaded RLINKs, but representing empty left subtrees by LLINK = Λ. (Similarly, a left-threaded binary tree threads only the null LLINKs.) Algorithm S does not make essential use of threaded LLINKs; if we change the test "LTAG = +" in step S2 to "LLINK ≠ Λ", we obtain an algorithm for traversing right-threaded binary trees in post-order. Program S works without change in the right-threaded case. A great many applications of binary tree structures require only a left-to-right traversal of trees using the functions P$ and/or P*, and for these applications there is no need to thread the LLINKs. We have described threading in both the left and right directions in order to indicate the symmetry and possibilities of the situation, but in practice one-sided threading is much more common.

Let us now consider an important concept about binary trees, and its connection to traversal. Two binary trees T and T' are said to be *similar* if they have the same structure; formally, this means either they are both empty, or they are both nonempty and their left and right subtrees are respectively similar. Similarity means, informally, that the diagrams of T and T' have the same "shape." Another way to phrase similarity is to say there is a one-to-one correspondence between the nodes of T and T' which preserves the structure: if nodes u_1 and u_2 in T correspond respectively to u_1' and u_2' in T', then u_1 is

in the left subtree of u_2 if and only if u_1' is in the left subtree of u_2', and the same is true for right subtrees.

The binary trees T and T' are said to be *equivalent* if they are similar and, moreover, if corresponding nodes contain the same information. Formally, let info (u) denote the information contained in a node u; the trees are equivalent if and only if they are both empty, or they are both nonempty and info (root(T)) = info (root (T')) and their left and right subtrees are respectively equivalent.

As examples of these definitions, consider the four binary trees



in which the first two are not similar (although they are equivalent as *trees* instead of as binary trees); the second, third, and fourth are similar and, in fact, the second and fourth are equivalent.

Computer applications involving tree structures often require an algorithm to decide whether two binary trees are similar or equivalent. The following theorem is useful in this regard:

Theorem A. *Let the nodes of binary trees T and T' be respectively*

$$u_1, u_2, \ldots, u_n \qquad \textit{and} \qquad u_1', u_2', \ldots, u_{n'}'$$

in preorder. For any node u let

$$\begin{array}{llll} l(u) = 1 & \text{if } u \text{ has a nonempty left subtree,} & l(u) = 0 & \text{otherwise;} \\ r(u) = 1 & \text{if } u \text{ has a nonempty right subtree,} & r(u) = 0 & \text{otherwise.} \end{array} \tag{10}$$

Then T and T' are similar if and only if $n = n'$ and

$$l(u_j) = l(u_j'), \qquad r(u_j) = r(u_j') \qquad \text{for} \qquad 1 \le j \le n. \tag{11}$$

T and T' are equivalent if and only if in addition

$$\text{info } (u_j) = \text{info } (u_j') \qquad \text{for} \qquad 1 \le j \le n. \tag{12}$$

Note that l and r are essentially the contents of the LTAG and RTAG in a threaded tree, with 1 and 0 substituted for "+" and "−". This theorem characterizes any binary tree structure in terms of two sequences of 0's and 1's.

Proof. It is clear that the condition for equivalence of binary trees will follow immediately if we prove the condition for similarity; furthermore $n = n'$ and

(11) is certainly necessary, since corresponding nodes of similar trees must have the same position in preorder. Therefore it suffices to prove that the conditions (11) and $n = n'$ will guarantee the similarity of T and T'. The proof is by induction on n, using the following auxiliary result:

Lemma P. *Let the nodes of a nonempty binary tree be $u_1, u_2, \ldots, u_n$ in preorder, and let $f(u) = l(u) + r(u) - 1$. Then*

$$\begin{aligned} f(u_1) + f(u_2) + \cdots + f(u_n) &= -1, \qquad \text{and} \\ f(u_1) + \cdots + f(u_k) \geq 0 \qquad &\text{for} \qquad 1 \leq k < n. \end{aligned} \tag{13}$$

Proof. The result is clear for $n = 1$. If $n > 1$, the binary tree consists of its root u_1 and further nodes. If $f(u_1) = 0$, then either the left subtree or the right subtree is empty, so the condition is obviously true by induction. If $f(u_1) = 1$, let the left subtree have n_l nodes; by induction we have

$$\begin{aligned} f(u_1) + \cdots + f(u_k) > 0 \qquad &\text{for} \qquad 1 \leq k \leq n_l, \\ f(u_1) + \cdots + f(u_{n_l+1}) &= 0, \end{aligned} \tag{14}$$

and the condition (13) is again evident. ∎

(For other theorems analogous to Lemma P, see the discussion of "Polish notation" in Chapter 11.)

To complete the proof of Theorem A, note that it is clearly true when $n = 0$. If $n > 0$, the definition of preorder implies that u_1 and u_1' are the respective roots of their trees, and there are integers n_l, n_l' (the sizes of the left subtrees) such that

$u_2, \ldots, u_{n_l+1}$ and $u_2', \ldots, u_{n_l'+1}'$ are the left subtrees of T and T';
$u_{n_l+2}, \ldots, u_n$ and $u_{n_l'+2}', \ldots, u_n'$ are the right subtrees of T and T'.

The proof by induction will be complete if we can show $n_l = n_l'$. There are three cases:

if $l(u_1) = 0$, then $n_l = 0 = n_l'$;
if $l(u_1) = 1$, $r(u_1) = 0$, then $n_l = n - 1 = n_l'$;
if $l(u_1) = r(u_1) = 1$, then by Lemma P we can find the least $k > 0$ such that $f(u_1) + \cdots + f(u_k) = 0$; and $n_l = k - 1 = n_l'$ (cf. Eqs. 14). ∎

As a consequence of Theorem A, we can test two threaded binary trees for equivalence or similarity by simply traversing them in preorder and checking the `INFO` and `TAG` fields.

We conclude this section by giving a typical, yet basic, algorithm for binary trees, one that makes a copy of a binary tree into different memory locations:

Algorithm C (*Copy a binary tree*). Let `HEAD` be the address of the list head of a binary tree T (i.e., T is the left subtree of `HEAD`; `LLINK(HEAD)` is a pointer to the tree). Let `NODE(U)` be a node with an empty left subtree. This algorithm makes a copy of T and the copy becomes the left subtree of `NODE(U)`. In partic-

ular, if NODE(U) is the list head of an empty binary tree, this algorithm changes the empty tree into a copy of T.

C1. [Initialize.] Set P ← HEAD, Q ← U. Go to C4.

C2. [Anything to right?] If NODE(P) has a nonempty right subtree, set R ⇐ AVAIL, and attach NODE(R) to the right of NODE(Q). (At the beginning of step C2, the right subtree of NODE(Q) is empty.)

C3. [Copy INFO.] Set INFO(Q) ← INFO(P). (Here INFO denotes all parts of the node that are to be copied.)

C4. [Anything to left?] If NODE(P) has a nonempty left subtree, set R ⇐ AVAIL, and attach NODE(R) to the left of NODE(Q). (At the beginning of step C4, the left subtree of NODE(Q) is empty.)

C5. [Advance.] Set P ← P*, Q ← Q*.

C6. [Test if complete.] If P = HEAD (or equivalently if Q = RLINK(U), assuming NODE(U) has a nonempty right subtree), the algorithm terminates; otherwise go to step C2. ▮

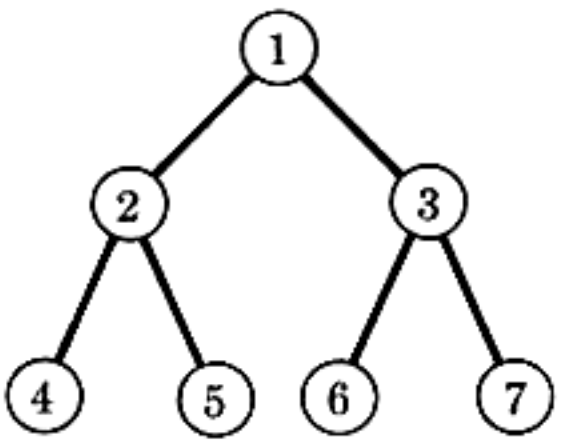
This simple algorithm shows a typical application of tree traversal; the description here applies to threaded, unthreaded, or partially threaded trees. Step C5 requires the calculation of preorder successors P* and Q*; for unthreaded trees, this generally is done with an auxiliary stack. A proof of the validity of Algorithm C appears in exercise 29; a MIX program corresponding to this algorithm in the case of a right-threaded binary tree appears in exercise 2.3.2–13. For threaded trees, the "attaching" in steps C2 and C4 is done using Algorithm I.

The exercises which follow include quite a few topics of interest relating to the material of this section.

EXERCISES

1. [*01*] In the binary tree (2), let `INFO(P)` denote the letter stored in `NODE(P)`. What is `INFO(LLINK(RLINK(RLINK(T))))`?

2. [*11*] List the nodes of the following binary tree in (a) preorder; (b) symmetric order; (c) endorder:



3. [*20*] Is the following statement true or false? "The terminal nodes of a binary tree occur in the same relative position in preorder, postorder, and endorder."

▶ **4.** [*20*] The text defines three basic orders for traversing a binary tree; another alternative would be to proceed in three steps as follows:

a) Visit the root,
b) traverse the right subtree,
c) traverse the left subtree,

using the same rule recursively on all nonempty subtrees. Does this new order bear any simple relation to the three orders already discussed?

5. [*22*] Nodes of a binary tree may be identified by a sequence of zeros and ones, in a notation analogous to "Dewey decimal notation" for trees, as follows: The root (if present) is represented by the sequence "1". Roots (if present) of the left and right subtrees of the node represented by α are respectively represented by $\alpha 0$ and $\alpha 1$. For example, the node H in (1) would have the representation "1110". (Cf. exercise 2.3–15.)

Show that preorder, postorder, and endorder may be conveniently described in terms of this notation.

6. [*M22*] Suppose that a binary tree has n nodes which are $u_1\, u_2 \ldots u_n$ in preorder and $u_{p_1}\, u_{p_2} \ldots u_{p_n}$ in postorder. Show that the permutation $p_1\, p_2 \ldots p_n$ can be obtained by passing $1\, 2 \ldots n$ through a stack, in the sense of exercise 2.2.1–2. Conversely, show that any permutation $p_1\, p_2 \ldots p_n$ obtainable with a stack corresponds to some binary tree in this way.

7. [*22*] Show that if we are given the preorder and the postorder of the nodes of a binary tree, the binary tree structure may be constructed. Does the same result hold true if we are given the preorder and endorder (instead of postorder)? Or if we are given the postorder and endorder?

8. [*20*] Find all binary trees whose nodes appear in exactly the same sequence in both (a) preorder and postorder; (b) preorder and endorder; (c) postorder and endorder.

9. [*M20*] When a binary tree having n nodes is traversed using Algorithm T, state how many times each of the steps T1, T2, T3, T4, and T5 is performed (as a function of n).

▶ **10.** [*20*] What is the largest number of entries that can be in the stack at once, during the execution of Algorithm T, if the binary tree has n nodes? (The answer to this question is very important for storage allocation, if the stack is being stored consecutively.)

11. [*M48*] Analyze the *expected* value of the largest stack size occurring during the execution of Algorithm T as a function of n, given that all binary trees with n nodes are considered equally probable.

12. [*22*] Design an algorithm analogous to Algorithm T which traverses a binary tree in *preorder*, and prove that your algorithm is correct.

▶ **13.** [*24*] Design an algorithm analogous to Algorithm T which traverses a binary tree in *endorder*.

14. [*22*] Show that if a binary tree with n nodes is represented as in (2), the total number of Λ links in the representation can be expressed as a simple function of n; this quantity does not depend on the shape of the tree.

15. [*19*] Note that in a threaded-tree representation like (9), each node except the list head has exactly one link pointing to it from above, namely the link from its "father."

In addition, some of the nodes have further links pointing to them from below; for example, the node containing "*C*" has two pointers coming up from below, node "*E*" has one. Is there any simple connection between the number of links pointing to a node and some other basic property of that node? (It is, of course, necessary to know how many links point to a given node when alterations to the tree structure are being considered.)

▶ **16.** [*22*] The diagrams in Fig. 24 help to give an intuitive characterization of the position of NODE(Q$) in a binary tree, in terms of the structure near NODE(Q): If NODE(Q) has a nonempty right subtree, consider Q = P, Q = P in the upper diagrams; NODE(Q$) is the "leftmost" node of that right subtree. If NODE(Q) has an empty right subtree, consider Q = P in the lower diagrams; NODE(Q$) is located by proceeding upward in the tree until after the first upward step to the right.

Give a similar "intuitive" rule for finding the position of NODE(Q*) in a binary tree in terms of the structure near NODE(Q).

▶ **17.** [*22*] Give an algorithm analogous to Algorithm S for determining P* in a threaded binary tree. Assume that the tree has a list head as in (7), (8), (9).

18. [*24*] Many algorithms dealing with trees like to visit each node *twice* instead of once, using a combination of preorder and postorder called the *dual order*. Traversal of a binary tree in dual order is defined as follows: If the binary tree is empty, do nothing; otherwise

a) visit the root, for the first time;
b) traverse the left subtree, in dual order;
c) visit the root, for the second time;
d) traverse the right subtree, in dual order.

For example, traversal of (1) in dual order gives the sequence

$$A_1B_1D_1D_2B_2A_2C_1E_1E_2G_1G_2C_2F_1H_1H_2F_2J_1J_2$$

where A_1 means "*A*" is being visited for the first time, etc.

If P points to a node of the tree and if $d = 1$ or 2, define $(\mathrm{P}, d)^\Delta = (\mathrm{Q}, e)$ if either the next step in dual order after visiting NODE(P) the dth time is to visit NODE(Q) the eth time, or if (P, d) was the last step in dual order and (Q, e) = (HEAD, 2), where HEAD is the address of the list head. We also define $(\mathrm{HEAD}, 1)^\Delta$ as the first step in dual order.

Design an algorithm analogous to Algorithm T which traverses a binary tree in dual order, and also design an algorithm analogous to Algorithm S which computes $(\mathrm{P}, d)^\Delta$. Discuss the relation between these algorithms and exercises 12 and 17.

19. [*24*] Design an algorithm analogous to Algorithm S for the calculation of P# in a threaded binary tree.

20. [*23*] Modify Program T so that it keeps the stack in a linked list, not in consecutive memory locations.

21. [*32*] Design an algorithm which traverses an unthreaded binary tree in postorder *without using any auxiliary stack*. It is permissible to alter the LLINK and RLINK fields of the tree nodes during this algorithm in any manner whatever, subject only to the condition that the binary tree has its conventional representation [as in (2), for

example] both before and after your algorithm has traversed the tree. You may also use an RTAG field (one bit only) in each node for temporary storage.

22. [*25*] Write a MIX program for the algorithm given in exercise 21 and compare its execution time to Programs S and T.

23. [*22*] Design algorithms analogous to Algorithm I for insertion to the right and insertion to the left in a *right-threaded* binary tree. Assume that the nodes have the fields LLINK, RLINK, and RTAG.

24. [*M20*] Is Theorem A still valid if the nodes of T and T' are given in postorder instead of preorder?

25. [*M24*] Let $\mathfrak{I}$ be a set of binary trees and let $N(\mathfrak{I})$ be the set $\{\text{info}\,(u) \mid u$ is a node of T for some T in $\mathfrak{I}\}$. Suppose that a linear ordering relation "$\preceq$" (cf. exercise 2.2.3–14) has been defined on $N(\mathfrak{I})$. Given any trees T, T' in $\mathfrak{I}$, let us now define $T \preceq T'$ if and only if

1) T is empty; or
2) T and T' are not empty, and info (root (T)) $\prec$ info (root (T')); or
3) T and T' are not empty, info (root (T)) $=$ info (root (T')), leftsubtree (T) $\preceq$ leftsubtree (T'), and leftsubtree (T) is not equivalent to leftsubtree (T'); or
4) T and T' are not empty, info (root (T)) $=$ info (root (T')), leftsubtree (T) is equivalent to leftsubtree (T'), and rightsubtree (T) $\preceq$ rightsubtree (T').

Prove that (a) $T \preceq T'$ and $T' \preceq T''$ implies $T \preceq T''$; (b) T is equivalent to T' if and only if $T \preceq T'$ and $T' \preceq T$; (c) for any T, T' in $\mathfrak{I}$ we have either $T \preceq T'$ or $T' \preceq T$. [Thus, if equivalent trees in $\mathfrak{I}$ are regarded as equal, the relation $\preceq$ induces a linear ordering on $\mathfrak{I}$. This ordering has many applications (for example, in the simplification of algebraic expressions). When $N(\mathfrak{I})$ has only one element, i.e., if the "info" of each node is the same, we have the special case that equivalence is the same as similarity.]

26. [*M24*] Consider the ordering $T \preceq T'$ defined in the preceding exercise. Prove a theorem analogous to Theorem A, giving a necessary and sufficient condition that $T \preceq T'$, and making use of the dual order as defined in exercise 18.

▸ **27.** [*28*] Design an algorithm which tests two given trees T and T' to see whether $T \prec T'$, $T \succ T'$, or T equivalent to T', in terms of the relation defined in exercise 25, assuming that both binary trees are right-threaded. Assume that each node has the fields LLINK, RLINK, RTAG, INFO; use no auxiliary stack.

28. [*00*] After Algorithm C has been used to make a copy of a tree, is the new binary tree *equivalent* to the original, or *similar* to it?

29. [*M25*] Prove as rigorously as possible that Algorithm C is valid.

▸ **30.** [*22*] Design an algorithm that threads an unthreaded tree [e.g., transforms (2) into (9)]. *Note:* Always use the notation P*, P$, etc. when possible instead of repeating the steps for traversal algorithms like Algorithm T.

31. [*23*] Design an algorithm which "erases" a right-threaded binary tree, i.e., returns all of its nodes except the list head to the AVAIL list and makes the list head signify an empty binary tree. Assume that each node has the fields LLINK, RLINK, RTAG; use no auxiliary stack.

32. [*21*] Suppose that each node of a binary tree has four link fields: LLINK and RLINK, which point to left and right subtrees or Λ, as in an unthreaded tree; and SUC and PRED, which point to the successor and predecessor of the node in symmetric order. (Thus, SUC(P) = P$ and PRED(P) = $P. Such a tree contains more information than a threaded tree.) Design an algorithm like Algorithm I for insertion into such a tree.

▶ **33.** [*30*] There is more than one way to thread a tree! Consider the following representation, using three fields LTAG, LLINK, RLINK in each node:

LTAG(P): defined the same as in a threaded binary tree;
LLINK(P): always equal to P*;
RLINK(P): defined the same as in an unthreaded binary tree.

Discuss insertion algorithms for such a representation, and write out the copying algorithm, Algorithm C, in detail for this representation.

34. [*22*] Let P point to a node in some binary tree, and let HEAD be the address of the list head of an empty binary tree. Give an algorithm which removes NODE(P) and all of its subtrees from whatever tree it was in, and which attaches the subtree having NODE(P) as its root to HEAD. Assume that all the binary trees in question are *right-threaded*, with fields LLINK, RTAG, RLINK in each node.

35. [*40*] Define a *ternary tree* (and, more generally, a t-ary tree for any $t \geq 2$) in a manner analogous to our definition of a binary tree, and explore the topics discussed in this section (including topics found in the exercises above) which can be generalized to t-ary trees in a meaningful way.

36. [*M23*] Exercise 1.2.1–15 shows that lexicographic order extends a well-ordering of a set S to a well-ordering of the n-tuples of elements of S. Exercise 25 above shows that a linear ordering of the information in tree nodes can be extended to a linear ordering of trees, using a similar definition. If the relation $\prec$ well-orders $N(\mathfrak{I})$, is the extended relation of exercise 25 a well-ordering of $\mathfrak{I}$?

▶ **37.** [*24*] (D. Ferguson.) If two computer words are necessary to contain two link fields and an INFO field, representation (2) requires $2n$ words of memory for a tree with n nodes. Design a representation scheme for binary trees which uses less space, assuming that *one* link and an INFO field will fit in a single computer word.

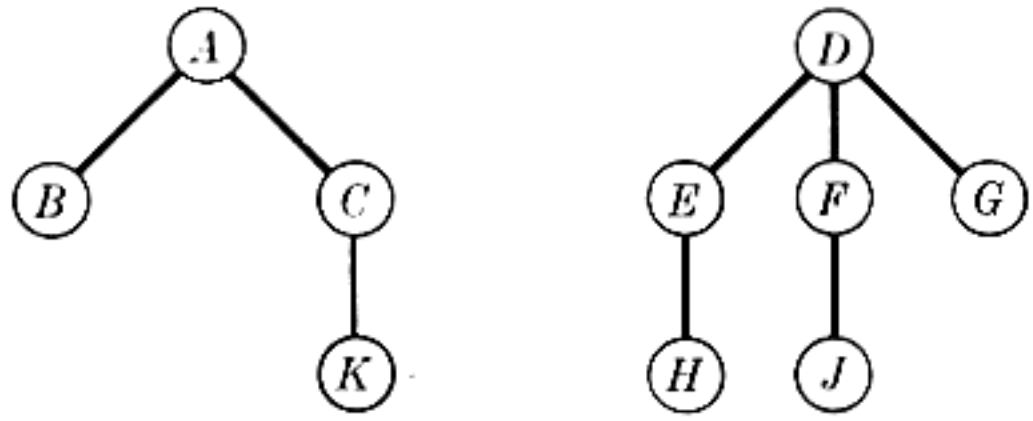
2.3.2. Binary Tree Representation of Trees

We turn now from binary trees to just plain trees. Let us recall the basic differences between trees and binary trees as we have defined them:

1) A tree is never empty, i.e., it always has at least one node; and each node of a tree can have 0, 1, 2, 3, . . . sons.
2) A binary tree can be empty, and each of its nodes can have 0, 1, or 2 sons; in the case of a single son we distinguish between a "left" son and a "right" son.

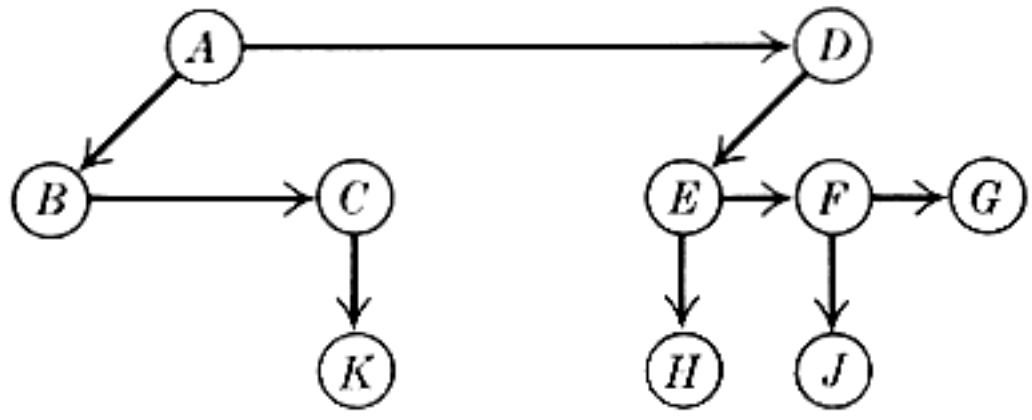
Recall also that a "forest" is an ordered set of zero or more trees. The subtrees immediately below any node of a tree form a forest.

There is a natural way to represent any forest as a binary tree. Consider the following forest of two trees:



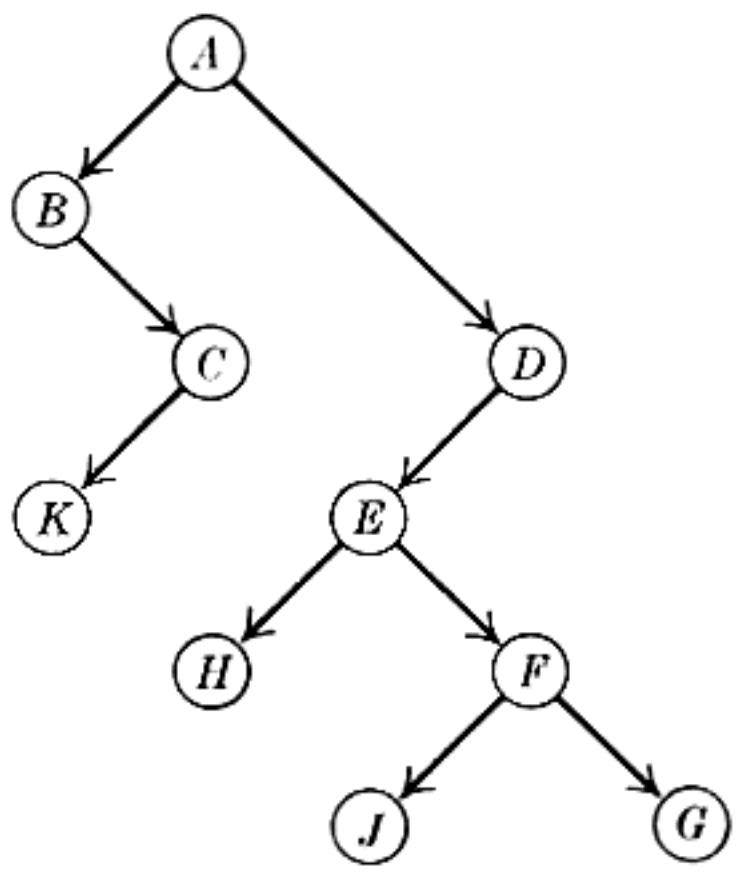
(1)

The corresponding binary tree is obtained by linking together the sons of each family and removing vertical links except from a father to his first son:



(2)

Then, tilt the diagram 45° and we have a binary tree:



(3)

Conversely, it is easy to see that any binary tree corresponds to a unique forest of trees by reversing the process.

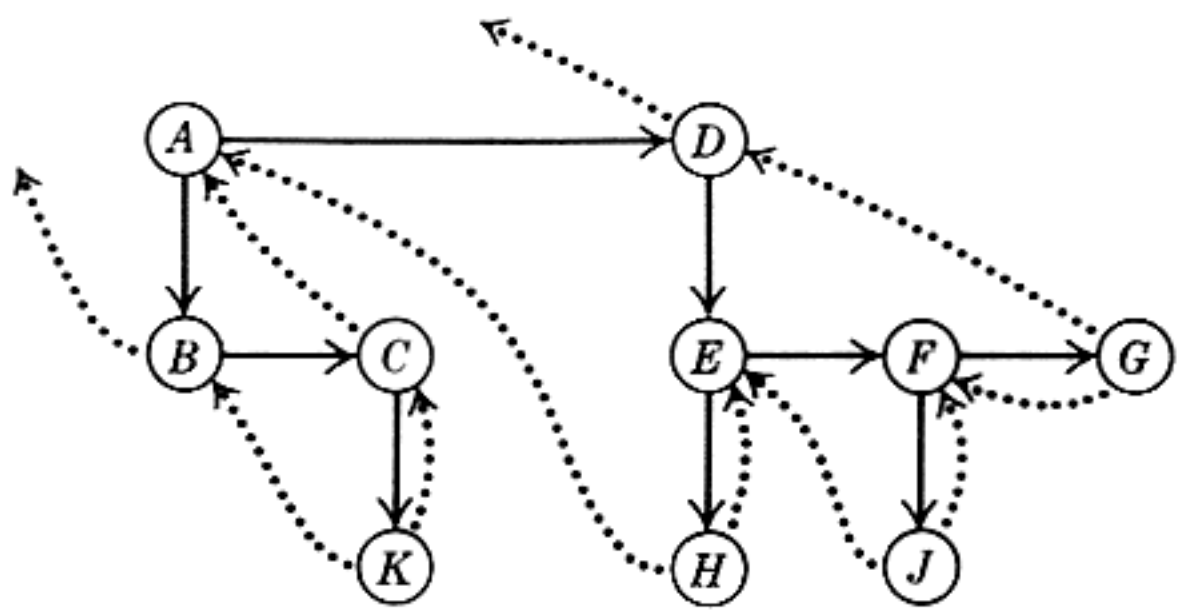
The above transformation is extremely important; it is called the *natural correspondence* between forests and binary trees. (In particular, it gives a correspondence between trees and those binary trees which have a root but no right subtree. We may also change things a little and let the list head of a binary tree correspond to the root of a tree, thus obtaining a one-to-one correspondence between trees and binary trees.)

Let $F = (T_1, T_2, \ldots, T_n)$ be a forest of trees. The binary tree $B(F)$ corresponding to F can be defined rigorously as follows:

a) If $n = 0$, $B(F)$ is empty.
b) If $n > 0$, the root of $B(F)$ is root (T_1); the left subtree of $B(F)$ is $B(T_{11}, T_{12}, \ldots, T_{1m})$, where $T_{11}, T_{12}, \ldots, T_{1m}$ are the subtrees of root (T_1); and the right subtree of $B(F)$ is $B(T_2, \ldots, T_n)$.

These rules specify the transformation from (1) to (3) precisely.

It will occasionally be convenient to draw our binary tree diagrams as in (2), without the 45° rotation. The *threaded* binary tree corresponding to (1) is



(4)

(compare with Fig. 24, giving the latter a 45° change in orientation). Note that *right thread links go from the rightmost son of a family to the father.* Left thread links do not have such a natural interpretation, due to the lack of symmetry between left and right.

The ideas about traversal expressed in the previous section can now be recast in terms of forests (and, therefore, trees) under the natural correspondence. The three orders of traversal take the following form:

Preorder	*Postorder*	*Endorder*
a) Visit the root of the first tree;	a) Traverse the subtrees of the first tree (in postorder);	a) Traverse the subtrees of the first tree (in endorder);
b) traverse the subtrees of the first tree (in preorder);	b) visit the root of the first tree;	b) traverse the remaining trees (in endorder);
c) traverse the remaining trees (in preorder).	c) traverse the remaining trees (in postorder).	c) visit the root of the first tree.

The algorithms developed in the previous section may be applied without change.

In order to understand the significance of these three methods of traversal, consider the following notation for expressing tree structure by nested parentheses:

$$(A(B, C(K)),\ D(E(H), F(J), G)). \tag{5}$$

This notation corresponds to the forest (1): we represent a tree by the information written in its root, followed by a representation of its subtrees; the representation of a nonempty forest is a parenthesized list of the representations of its trees, separated by commas.

If (1) is traversed in preorder, we visit the nodes in the sequence $A\ B\ C\ K\ D\ E\ H\ F\ J\ G$; this is simply (5) with the parentheses and commas removed. Preorder is a natural way to list the nodes of a tree: we list the root first, then the descendants. If a tree structure is represented by indentation as in Fig. 20(c), the rows appear in preorder. The section numbers of this book itself (see Fig. 21) appear in preorder; thus, for example, Section 2.3.2 follows Section 2.3.1, which follows Section 2.3, which follows Section 2.2.6.

Postorder for the nodes in (1) is $B\ K\ C\ A\ H\ E\ J\ F\ G\ D$; this is analogous to preorder, except that it corresponds to the similar parenthesis notation

$$((B, (K)C)A,\ ((H)E, (J)F, G)D), \tag{6}$$

in which a node appears just *after* its descendants instead of just before. The words "preorder" and "postorder" are meant to imply the difference between (5) and (6).

It is interesting to note that preorder is a time-honored concept which might meaningfully be called *dynastic order*. At the death of a king, duke, or earl, etc., the title passes to his first son, then to descendants of the first son, and finally if these all die out it passes to other sons of the family in the same way. (English custom also includes daughters in a family on the same basis as sons, except they come after all the sons.) In theory, we could take a lineal chart of all the aristocracy and write out the nodes in preorder; then if we consider only the people presently living, we would obtain the *order of succession to the throne* (except as modified by Acts of Abdication).

The nodes of (1) in endorder are

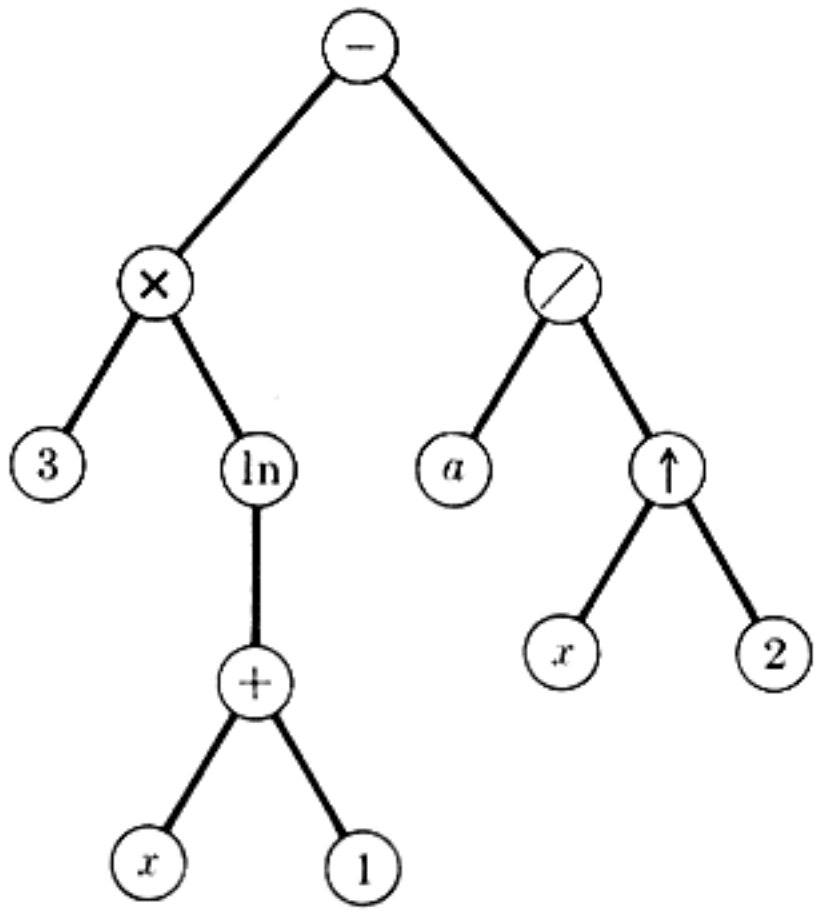
$$K\ C\ B\ H\ J\ G\ F\ E\ D\ A;$$

this may look a little strange at first, but it has a simple interpretation: Skip from left to right in either (5) or (6) until hitting the first right parenthesis; then go to the left, visiting nodes, until encountering a left parentheses; then erase the entire parenthesis group from the expression and go to the next right parenthesis, etc., until the whole expression has been erased. We will see in the next section that the reverse of endorder, for example, $A\ D\ E\ F\ G\ J\ H\ B\ C\ K$, is also important.

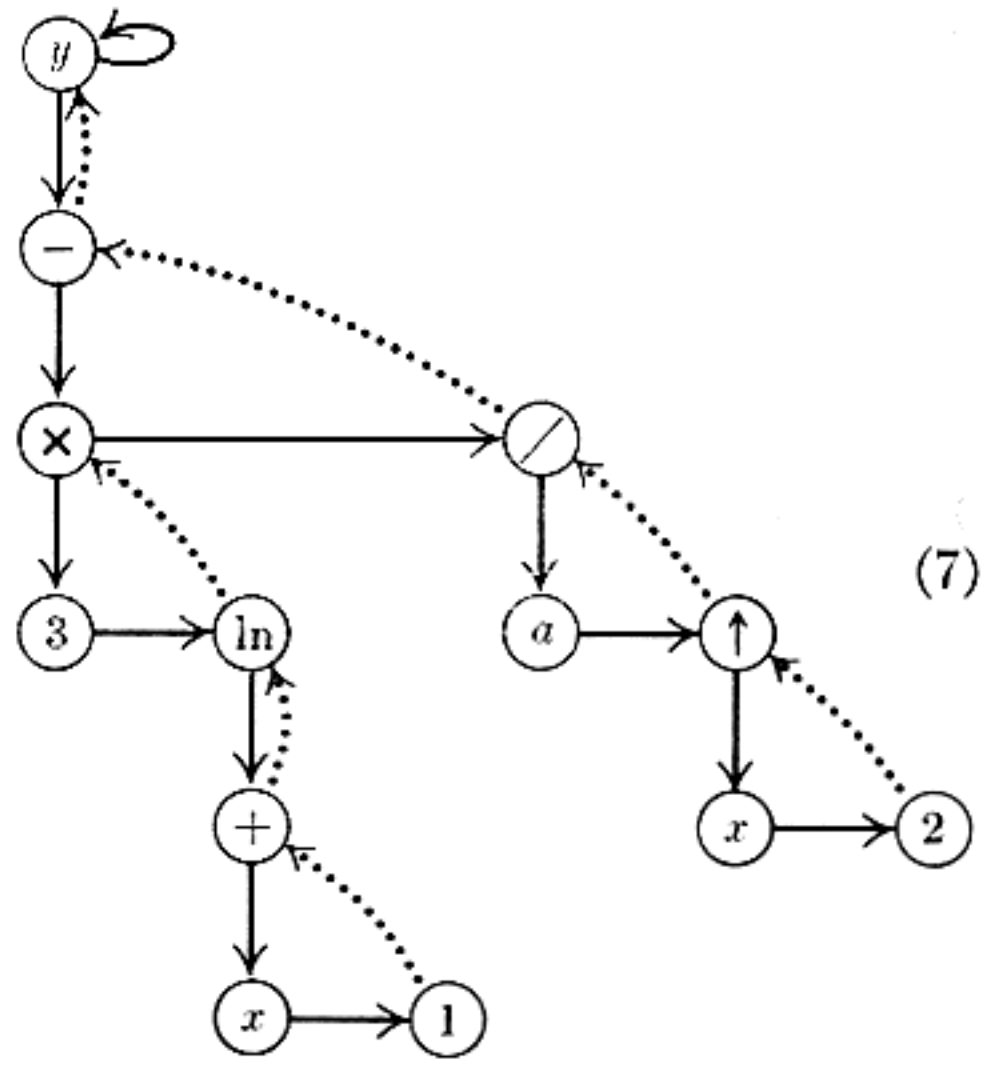
As an example of the application of these methods to a practical problem, let us consider the manipulation of algebraic formulas. Such formulas are most properly regarded as representations of tree structures, not as one- or two-dimensional configurations of symbols. The formula

$$y = 3 \ln (x + 1) - a/x^2$$

has the tree representation



i.e.,



(7)

Here the left-hand diagram gives the conventional tree representation, like Fig. 22, in which the binary operators $+$, $-$, $\times$, $/$, and $\uparrow$ (the latter denotes exponentiation) have two subtrees corresponding to their operands; the unary operators "ln" and "neg" (the latter does not appear in this tree; it denotes negation as in "$y = -x$") have one subtree, and variables and constants are terminal nodes. In the right-hand diagram, we have shown the equivalent right-threaded binary tree, including an additional node y which is a list head for the tree. The list head has the form described in Eq. 2.3.1-(7).

The nodes of this tree are

$$- \quad \times \quad 3 \quad \ln \quad + \quad x \quad 1 \quad / \quad a \quad \uparrow \quad x \quad 2 \qquad \text{in preorder;} \tag{8}$$

$$3 \quad x \quad 1 \quad + \quad \ln \quad \times \quad a \quad x \quad 2 \quad \uparrow \quad / \quad - \qquad \text{in postorder.} \tag{9}$$

Algebraic expressions like (8) and (9) are very important, and they are known as "Polish notations" because form (8) was introduced by the Polish logician, Łukasiewicz. Expression (8) is the *prefix notation* for formula (7), and (9) is the corresponding *postfix notation*. We will return to the interesting topic of Polish notation in later chapters; for now let us be content with the knowledge that Polish notation is directly related to the basic orders of tree traversal.

It is important to note that, even though the left-hand tree in (7) bears a superficial resemblance to a binary tree, we are treating it here as a *tree*, and representing it by a quite different binary tree, shown at the right in (7). Although we could develop routines for algebraic manipulations based directly on binary tree structures—these are the so-called "three-address code" representations of algebraic formulas—the author has found that several simplifications occur in practice if we use the general tree representation of algebraic formulas, as in (7), although the tree need not necessarily be threaded.

Let us assume that tree structures for the algebraic formulas with which we will be dealing have nodes of the following form:

RTAG	RLINK	TYPE	LLINK
INFO			

(10)

Here RLINK and LLINK have the usual significance, and RTAG is negative for thread links. The TYPE field is used to distinguish different kinds of nodes: TYPE $= 0$ means the node represents a constant, and INFO is the value of the constant. TYPE $= 1$ means the node represents a variable, and INFO is the five-letter alphabetic name of this variable. TYPE ≥ 2 means the node represents an operator; INFO is the alphabetic name of the operator and the value TYPE $= 2, 3, 4, \ldots$ is used to distinguish the different operators $+$, $-$, $\times$, $/$, etc. We will not concern ourselves here with how the tree structure has been set up inside the computer memory in the first place, since this topic is analyzed in great detail in Chapter 10; let us merely assume that the tree already appears in our computer memory, and questions of input and output will be deferred until later.

We will now discuss the "classical" example of algebraic manipulation, finding the *derivative* of a formula with respect to the variable x. Programs for algebraic differentiation were among the first symbol-manipulation routines ever written for computers; they were written as early as 1952. The process of differentiation illustrates many of the techniques of algebraic manipulation, and it is of significant practical value in scientific applications.

Readers who are not familiar with mathematical calculus may consider this problem as an abstract exercise in formula manipulation, defined by the following rules:

$$D(x) = 1 \tag{11}$$

$$D(a) = 0, \quad \text{if } a \text{ is a constant or a variable} \neq x. \tag{12}$$

$$D(\ln u) = D(u)/u \tag{13}$$

$$D(-u) = -D(u) \tag{14}$$

$$D(u+v) = D(u) + D(v) \tag{15}$$

$$D(u-v) = D(u) - D(v) \tag{16}$$

$$D(u \times v) = D(u) \times v + u \times D(v) \tag{17}$$

$$D(u / v) = D(u)/v - (u \times D(v))/(v \uparrow 2) \tag{18}$$

$$D(u \uparrow v) = D(u) \times (v \times (u \uparrow (v-1))) + ((\ln u) \times D(v)) \times (u \uparrow v) \tag{19}$$

These rules allow us to evaluate the derivative $D(y)$ for any formula y composed of the above operators.

(Our main interest in this algorithm is, as usual, in the details of how the process is carried out inside a computer. There are many higher-level languages and special routines available at most computer installations which have built-in

facilities that greatly simplify algebraic manipulations like these; but the purpose of the present example is to gain more experience in fundamental tree operations.)

The idea behind the following algorithm is to traverse the tree in postorder, forming the derivative of each node as we go, until eventually the entire derivative has been calculated. Using postorder means that we arrive at an operator node (like "+") *after* its operands have been differentiated. Rules (11) through (19) imply that every subformula of the original formula will have to be differentiated, sooner or later, so we might as well do the differentiations in postorder. By using a right-threaded tree, we avoid the need for a stack during the operation of the algorithm. On the other hand, a threaded tree representation has the disadvantage that it is necessary to make copies of subtrees (for example, in the rule for $D(u \uparrow v)$ we may need to copy u and v each three times), when in many circumstances we could use a List representation instead of a tree and avoid this copying; see Section 2.3.5.

Algorithm D (*Differentiation*). If `Y` is the address of a list head which points to a formula represented as described above, and if `DY` is the address of the list head for an empty tree, this algorithm makes `NODE(DY)` point to a tree representing the analytic derivative of `Y` with respect to the variable "`X`".

D1. [Initialize.] Set `P` ← `Y$` (i.e., the first node of the tree, in postorder).

D2. [Differentiate.] Set `P1` ← `LLINK(P)` and if `P1` ≠ Λ, also set `Q1` ← `RLINK(P1)`. Then perform the routine `DIFF[TYPE(P)]`, described below. (The routines `DIFF[0]`, `DIFF[1]`, etc., will form the derivative of the tree with root `P`, and will set pointer variable `Q` to the address of the root of the derivative. The variables `P1` and `Q1` are set up first, in order to simplify the specification of the `DIFF` routines.)

D3. [Adjust link.] If `TYPE(P)` denotes a binary operator, set `RLINK(P1)` ← `P2`. (See the next step for an explanation.)

D4. [Advance to `P$`.] Set `P2` ← `P`, `P` ← `P$`. Now if `RTAG(P2)` = "+", i.e., if `NODE(P2)` has a brother on his right, set `RLINK(P2)` ← `Q`. (This is the tricky part of the algorithm: we temporarily destroy the structure of tree `Y`, so that a link to the derivative of `P2` is saved for future use. The missing link is reinserted in step D3. See exercise 21 for further discussion of this trick.)

D5. [Done?] If `P` ≠ `Y`, return to step D2. Otherwise set `LLINK(DY)` ← `Q` and `RLINK(Q)` ← `DY`, `RTAG(Q)` ← `"-"`. ▮

The procedure described in Algorithm D is just the background routine for the differentiation operations which are performed by the processing routines `DIFF[0]`, `DIFF[1]`, . . . , called in step D2. In many ways, Algorithm D is like the control routine for an interpretive system or machine simulator, as discussed in Section 1.4.3, but it traverses a tree instead of a simple sequence of instructions.

Let us now consider the routines which do the actual differentiation. In the following discussion, the statement "`P` points to a tree" means that `NODE(P)`

is the root of a tree stored in the conventional manner, and both RLINK(P) and RTAG(P) are meaningless so far as this tree is concerned. We will make use of a *tree construction function* which makes new trees by joining smaller ones together: Let x denote some kind of node, either a constant, variable, or operator, and let U and V denote pointers to trees; then we have

TREE(x, U, V) makes a new tree with x in its root node and with U and V the subtrees of the root: W ⇐ AVAIL, INFO(W) ← x, LLINK(W) ← U, RLINK(U) ← V, RTAG(U) ← "+", RLINK(V) ← W, RTAG(V) ← "−".

TREE(x, U) similarly makes a new tree with only one subtree: W ⇐ AVAIL, INFO(W) ← x, LLINK(W) ← U, RLINK(U) ← W, RTAG(U) ← "−".

TREE(x) makes a new tree with x as a terminal root node: W ⇐ AVAIL, INFO(W) ← x, LLINK(W) ← Λ.

In all cases, the value of TREE is W, that is, a pointer to the tree just constructed. Another function, COPY(U), makes a copy of the tree pointed to by U and has as its value a pointer to the tree thereby created.

The basic functions TREE and COPY make it easy to build up a tree for the derivative of a formula, step by step. Before we look at the DIFF routines, however, let us consider what happens if we blindly apply rules (11) through (19) to a rather simple formula like

$$y = 3 \ln (x + 1) - a/x^2;$$

we get

$$D(y) = (0 \cdot \ln (x + 1) + 3((1 + 0)/(x + 1)) \\ - (0/x^2 - (a(1(2x^{2-1}) + ((\ln x) \cdot 0)x^2))/(x^2)^2), \quad (20)$$

which is completely unsatisfactory. To avoid so many redundant operations in the answer, we must make our routines more complicated, so that they recognize the special cases of adding or multiplying by zero, multiplying by one, or raising to the first power. These simplifications reduce (20) to

$$D(y) = 3(1/(x + 1)) - ((-(a(2x)))/(x^2)^2), \quad (21)$$

and this is acceptable but not wholly satisfactory. The concept of a really satisfactory answer is not well-defined, because different mathematicians will prefer formulas to be expressed in different ways; however, it is clear that (21) is not as simple as it could be. In order to make substantial progress over formula (21), it is necessary to develop algebraic simplification routines (see exercise 17), which would reduce (21) to, for example,

$$D(y) = 3(x + 1)^{-1} + 2ax^{-3}. \quad (22)$$

We will content ourselves here with routines which can produce (21), not (22).

Nullary operators (*constants and variables*). For these operations, NODE(P) is a terminal node, and the values of P1, P2, Q1, and Q before the operation are irrelevant.

DIFF[0]: (NODE(P) is a constant.) Set Q ← TREE("0").

DIFF[1]: (NODE(P) is a variable.) If INFO(P) = "X", set Q ← TREE("1"); otherwise set Q ← TREE("0").

Unary operators (*logarithm and negation*). For these operations, NODE(P) has one son, U, pointed to by P1, and Q points to $D(U)$. The values of P2 and Q1 before the operation are irrelevant.

DIFF[2]: (NODE(P) is "ln".) If INFO(Q) $\neq 0$, set Q ← TREE("/",Q,COPY(P1)).

DIFF[3]: (NODE(P) is "neg".) If INFO(Q) $\neq 0$, set Q ← TREE("neg", Q).

Binary operators (*addition, subtraction, etc.*). For these operations, NODE(P) has two sons, U and V, pointed to respectively by P1 and P2; Q1 and Q point respectively to $D(U)$, $D(V)$.

DIFF[4]: ("+" operation.) If INFO(Q1) = 0, set AVAIL ⇐ Q1. Otherwise if INFO(Q) = 0, set AVAIL ⇐ Q and Q ← Q1; otherwise set Q ← TREE("+",Q1,Q).

DIFF[5]: ("−" operation.) If INFO(Q) = 0, set AVAIL ⇐ Q and Q ← Q1. Otherwise if INFO(Q1) = 0, set AVAIL ⇐ Q1 and set Q ← TREE("neg", Q); otherwise set Q ← TREE("−",Q1,Q).

DIFF[6]: ("×" operation.) If INFO(Q1) $\neq 0$, set Q1 ← MULT(Q1,COPY(P2)). Then if INFO(Q) $\neq 0$, set Q ← MULT(COPY(P1),Q). Then go to DIFF[4].

Here MULT(U,V) is a new function which constructs a tree for U × V but also makes a test to see if U or V is equal to "1":

if INFO(U) = 1 and TYPE(U) = 0, set AVAIL ⇐ U and MULT(U,V) ← V;
if INFO(V) = 1 and TYPE(V) = 0, set AVAIL ⇐ V and MULT(U,V) ← U;
otherwise set MULT(U;V) ← TREE("×",U,V).

DIFF[7]: ("/" operation.) If INFO(Q1) $\neq 0$, set

Q1 ← TREE("/",Q1,COPY(P2)).

Then if INFO(Q) $\neq 0$, set

Q ← TREE("/",MULT(COPY(P1),Q),TREE("↑",COPY(P2),TREE("2"))).

Then go to DIFF[5].

DIFF[8]: ("↑" operation.) See exercise 12.

We conclude this section by showing how all of the above operations are readily transformed into a computer program, starting "from scratch" with only MIX machine language as a basis.

Program D (*Differentiation*). The following MIXAL program performs Algorithm D, with rI2 ≡ P, rI3 ≡ P2, rI4 ≡ P1, rI5 ≡ Q, rI6 ≡ Q1. The order of computations has been rearranged a little, for convenience.

```
01   * DIFFERENTIATION IN A RIGHT-THREADED TREE
02   LLINK     EQU   4:5          Definition of fields, see (10)
03   RLINK     EQU   1:2
04   RLINKT    EQU   0:2
05   TYPE      EQU   3:3
06   * MAIN CONTROL ROUTINE       D1. Initialize.
07   D1        STJ   9F           Treat whole procedure as a subroutine.
08             LD4   Y(LLINK)     P1 ← LLINK(Y), prepare to find Y$.
09   1H        ENT2  0,4          P ← P1.
10   2H        LD4   0,2(LLINK)   P1 ← LLINK(P).
11             J4NZ  1B           If P1 ≠ Λ, repeat.
12   D2        LD1   0,2(TYPE)    D2. Differentiate.
13             JMP   *+1,1        Jump to DIFF[TYPE(P)].
14             JMP   CONSTANT     Switch table entry for DIFF[0].
15             JMP   VARIABLE                             DIFF[1].
16             JMP   LN                                      .
17             JMP   NEG                                     .
18             JMP   ADD                                     .
19             JMP   SUB                                     .
20             JMP   MUL                                     .
21             JMP   DIV                                     .
22             JMP   PWR                                  DIFF[8].
23   D3        ST3   0,4(RLINK)   D3. Adjust link. RLINK(P1) ← P2.
24   D4        ENT3  0,2          D4. Advance to P$. P2 ← P.
25             LD2   0,2(RLINKT)  P ← RLINKT(P).
26             J2N   1F           Jump if RTAG(P) = "—";
27             ST5   0,3(RLINK)   otherwise set RLINK(P2) ← Q.
28             JMP   2B           Note that NODE(P$) will be terminal.
29   1H        ENN2  0,2
30   D5        ENT1  -Y,2         D5. Done?
31             LD4   0,2(LLINK)   P1 ← LLINK(P), prepare for step D2.
32             LD6   0,4(RLINK)   Q1 ← RLINK(P1).
33             J1NZ  D2           Jump to D2 if P ≠ Y;
34             ST5   DY(LLINK)    otherwise set LLINK(DY) ← Q.
34a            ENNA  DY
34b            STA   0,5(RLINKT)  RLINK(Q) ← DY,RTAG(Q) ← "-".
35   9H        JMP   *            Exit from differentiation subroutine. ▮
```

The next part of the program contains the basic subroutines TREE and COPY. The former has three entrances, TREE0, TREE1, and TREE2, according to the number of subtrees of the tree being constructed. Regardless of which entrance to the subroutine is used, rA will contain the address of a special constant indicating what type of node forms the root of the tree being constructed; these special constants appear in lines 103–122.

```
* BASIC SUBROUTINES FOR TREE CONSTRUCTION
TREE0     STJ   9F              TREE(rA) function
          JMP   2F
TREE1     ST1   3F(0:2)         TREE(rA, rI1) function
          JSJ   1F
TREE2     STX   3F(0:2)         TREE(rA, rX, rI1) function
3H        ST1   *(RLINKT)       RLINK(rX) ← rI1, RTAG(rX) ← "+".
1H        STJ   9F
          LDXN  AVAIL
          JXZ   OVERFLOW
          STX   0,1(RLINKT)     RLINK(rI1) ← AVAIL, RTAG(rI1) ← "−".
          LDX   3B(0:2)
          STA   *+1(0:2)
          STX   *(LLINK)        Set LLINK of next root node.
2H        LD1   AVAIL           rI1 ⇐ AVAIL.
          J1Z   OVERFLOW
          LDX   0,1(LLINK)
          STX   AVAIL
          STA   *+1(0:2)        Move root node to available space.
          MOVE  *(2)
          DEC1  2               Reset rI1 to point to root node.
9H        JMP   *               Exit from TREE, result in rI1
COPYP1    ENT1  0,4             COPY(P1), special entrance to COPY
          JSJ   COPY
COPYP2    ENT1  0,3             COPY(P2), special entrance to COPY
COPY      STJ   9F              COPY(rI1) function
  ⋮         ⋮                   (see exercise 13)
9H        JMP   *               Exit from COPY, rI1 points to new tree.
CON0      CON   0               Node representing constant "0"
          CON   0
CON1      CON   0               Node representing "1"
          CON   1
CON2      CON   0               Node representing "2"
          CON   2
LOG       CON   2(TYPE)         Node representing "ln"
          ALF     LN
NEGOP     CON   3(TYPE)         Node representing "neg"
          ALF    NEG
PLUS      CON   4(TYPE)         Node representing "+"
          ALF     +
MINUS     CON   5(TYPE)         Node representing "−"
          ALF     -
TIMES     CON   6(TYPE)         Node representing "×"
          ALF     *
SLASH     CON   7(TYPE)         Node representing "/"
          ALF     /
UPARROW   CON   8(TYPE)         Node representing "↑"
          ALF    **
```

The remaining portion of the program corresponds to the differentiation routines DIFF[0], DIFF[1], . . . ; these routines are written to return control to step D3 after processing a binary operator, otherwise control is to return to step D4.

```
123   * DIFFERENTIATION ROUTINES
124   VARIABLE  LDX   1,2
125             ENTA  CON1
126             CMPX  2F          Is INFO(P) = "X"?
127             JE    *+2         If so, call TREE("1").
128   CONSTANT  ENTA  CON0        Call TREE("0").
129             JMP   TREE0
130   1H        ENT5  0,1         Q ← location of new tree.
131             JMP   D4          Return to control routine.
132   2H        ALF       X
133   LN        LDA   1,5
134             JAZ   D4          Return to control routine if INFO(Q) = 0;
135             JMP   COPYP1      otherwise set rI1 ← COPY(P1).
136             ENTX  0,5
137             ENTA  SLASH
138             JMP   TREE2       rI1 ← TREE("/", Q, rI1).
139             JMP   1B          Q ← rI1, return to control.
140   NEG       LDA   1,5
141             JAZ   D4          Return if INFO(Q) = 0.
142             ENTA  NEGOP
143             ENT1  0,5
144             JMP   TREE1       TREE("neg", Q)
145             JMP   1B          → Q, return to control.
146   ADD       LDA   1,6
147             JANZ  1F          Jump unless INFO(Q1) = 0.
148   3H        LDA   AVAIL       AVAIL ⇐ Q1.
149             STA   0,6(LLINK)
150             ST6   AVAIL
151             JMP   D3          Return to control, binary operator.
152   1H        LDA   1,5
153             JANZ  1F          Jump unless INFO(Q) = 0.
154   2H        LDA   AVAIL       AVAIL ⇐ Q.
155             STA   0,5(LLINK)
156             ST5   AVAIL
157             ENT5  0,6         Q ← Q1.
158             JMP   D3          Return to control.
159   1H        ENTA  PLUS        Prepare to call TREE("+", Q1, Q).
160   4H        ENTX  0,6
161             ENT1  0,5
162             JMP   TREE2
163             ENT5  0,1         Q ← TREE("±", Q1, Q).
164             JMP   D3          Return to control.
```

```
165  SUB     LDA   1,5
166          JAZ   2B          Jump if INFO(Q) = 0.
167          LDA   1,6
168          JANZ  1F          Jump unless INFO(Q1) = 0.
169          ENTA  NEGOP
170          ENT1  0,5
171          JMP   TREE1
172  .       ENT5  0,1         Q ← TREE("neg", Q).
173          JMP   3B          AVAIL ⇐ Q1 and return.
174  1H      ENTA  MINUS       Prepare to call TREE("—", Q1, Q).
175          JMP   4B
176  MUL     LDA   1,6
177          JAZ   1F          Jump if INFO(Q1) = 0;
178          JMP   COPYP2      otherwise set rI1 ← COPY(P2).
179          ENTA  0,6
180          JMP   MULT        MULT(Q1, COPY(P2))
181          ENT6  0,1         → Q1.
182  1H      LDA   1,5
183          JAZ   ADD         Jump if INFO(Q) = 0;
184          JMP   COPYP1      otherwise set rI1 ← COPY(P1).
185          ENTA  0,1
186          ENT1  0,5
187          JMP   MULT        MULT(COPY(P1), Q)
188          ENT5  0,1         → Q.
189          JMP   ADD
190  MULT    STJ   9F          MULT(rA, rI1) subroutine
191          STA   1F(0:2)     Let rA ≡ U, rI1 ≡ V.
192          ST2   8F(0:2)     Save rI2.
193  1H      ENT2  *           rI2 ← U.
194          LDA   1,2         Test if INFO(U) = 1
195          DECA  1
196          JANZ  1F
197          LDA   0,2(TYPE)   and if TYPE(U) = 0.
198          JAZ   2F
199  1H      LDA   1,1         If not, test if INFO(V) = 1
200          DECA  1
201          JANZ  1F
202          LDA   0,1(TYPE)   and if TYPE(V) = 0.
203          JANZ  1F
204          ST1   *+2(0:2)    If so, interchange U ↔ V.
205          ENT1  0,2
206          ENT2  *
207  2H      LDA   AVAIL       AVAIL ⇐ U.
208          STA   0,2(LLINK)
209          ST2   AVAIL
210          JMP   8F          Result is V.
211  1H      ENTA  TIMES
212          ENTX  0,2
```

```
213          JMP   TREE2     Result is TREE("×", U, V).
214   8H     ENT2  *         Restore rI2 setting.
215   9H     JMP   *         Exit MULT with result in rI1. ▌
```

The other two routines DIV and PWR are similar and they have been left as exercises (see exercises 15 and 16).

EXERCISES

▶ **1.** [*20*] The text gives a formal definition of $B(F)$, the binary tree corresponding to a forest F. Give a formal definition which reverses the process, i.e., define $F(B)$, the forest corresponding to a binary tree B.

▶ **2.** [*20*] We defined Dewey decimal notation for forests in Section 2.3, and for binary trees in exercise 2.3.1–5. Thus the node "J" in (1) is represented by "2.2.1", and in the equivalent binary tree (3) it is represented by "11010". If possible, give a rule that directly expresses the natural correspondence between trees and binary trees as a correspondence between the Dewey decimal notations.

3. [*22*] What is the relation between Dewey decimal notation for the nodes of a forest and the preorder and postorder of these nodes?

4. [*21*] Is the following statement true or false? "The terminal nodes of a tree occur in the same relative position in preorder, postorder, and endorder."

5. [*24*] The text gives various "intuitive" interpretations of preorder, postorder, and endorder. Give an "intuitive" interpretation of *reverse* endorder [for example, $A\ D\ E\ F\ G\ J\ H\ B\ C\ K$ in the forest (1)].

▶ **6.** [*25*] Let T be a nonempty binary tree in which each node has 0 or 2 sons. If we regard T as an ordinary tree, it corresponds (via the natural correspondence) to *another* binary tree T'. Is there any simple relation between preorder, postorder, and endorder of the nodes of T (as defined for binary trees) and the same three orders for the nodes of T'?

7. [*M20*] A forest may be regarded as a partial ordering, if we say that each node precedes its descendants in the tree. Are the nodes topologically sorted (as defined in Section 2.2.3) when they are listed in (a) preorder? (b) postorder? (c) endorder? (d) reverse preorder? (e) reverse postorder? (f) reverse endorder?

8. [*M20*] Exercise 2.3.1–25 shows how an ordering between the information stored in the individual nodes of a binary tree may be extended to a linear ordering of all binary trees. The same construction leads to an ordering of all trees, under the natural correspondence. Reformulate the definition of that exercise, in terms of trees.

9. [*M21*] Show that the total number of nonterminal nodes in a forest has a simple relation to the total number of right links equal to Λ in the corresponding binary tree.

10. [*M23*] Let F be a forest of trees whose nodes in preorder are $u_1, u_2, \ldots, u_n$, and let F' be a forest whose nodes in preorder are $u'_1, u'_2, \ldots, u'_{n'}$. Let $S(u)$ denote the degree (the number of sons) of node u. In terms of these ideas, formulate and prove a theorem analogous to Theorem 2.3.1A.

11. [*20*] Draw trees analogous to those shown in (7), corresponding to the formula $y = e^{-x^2}$.

12. [*M21*] Give specifications for the routine DIFF[8] (the "↑" operation), which was omitted from the algorithm in the text.

▶ **13.** [*26*] Write a MIX program for the COPY subroutine (which fits in the program of the text between lines 61–102). [*Hint:* Adapt Algorithm 2.3.1C to the case of right-threaded binary trees, with suitable initial conditions.]

▶ **14.** [*M21*] How long does it take the program of exercise 13 to copy a tree with n nodes?

15. [*23*] Write a MIX program for the DIV routine, corresponding to DIFF[7] as specified in the text. (This routine should be added to the program in the text after line 215.)

16. [*24*] Write a MIX program for the PWR routine, corresponding to DIFF[8] as specified in exercise 12. (This routine should be added to the program in the text after the solution to exercise 15.)

17. [*M40*] Write a program to do algebraic simplification capable of reducing, for example, (20) or (21) to (22). [*Hints:* Include a new field with each node, representing its coefficient (for summands) or its exponent (for factors in a product). Apply algebraic identities, like replacing $\ln(u \uparrow v)$ by $v \ln u$, and remove the operations $-$, $/$, $\uparrow$, and neg when possible by using equivalent addition or multiplication operations. Make $+$ and $\times$ into n-ary instead of binary operators; collect like terms by sorting their operands in tree order (exercise 8); some sums and products will now reduce to zero or unity, presenting perhaps further simplifications. Other adjustments, like replacing a sum of logarithms by the logarithm of a product, also suggest themselves.] For references, see the survey article by J. Sammet, *CACM* **9** (1966), 555–569.

18. [*M40*] Consider algebraic formulas which are composed of the operators of symbolic logic (AND, OR, and NOT, say). Try several different algorithms for deciding whether or not such a formula is a *tautology*, i.e., is true for all combinations of truth values of its variables. [For example,

((X AND Y) OR NOT X) OR NOT(Y AND Z)

is a tautology.] For each algorithm considered, analyze its computational efficiency. For examples of algorithms to try, see H. Wang, *IBM J. Res. Dev.* **4** (1960), 2–22, and *CACM* **3** (1960), 220–234; see also Dunham, Fridshal and Sward, *Information Processing* (Proceedings of the International Congress of Information Processing, UNESCO, Paris, 1959), 282–285. Another idea is to use the Boolean operations of a binary computer on 2^n-bit quantities, where n is the number of variables.

19. [*M35*] A *free lattice* is a mathematical system, which (for the purposes of this exercise) can be simply defined as the set of all formulas composed of variables and two abstract binary operators "$\cup$" and "$\cap$". A relation "$X \supseteq Y$" is defined between certain formulas X and Y in the free lattice by the following rules:

a) $X \cup Y \supseteq Z$ if and only if $X \supseteq Z$ or $Y \supseteq Z$

b) $X \cap Y \supseteq Z$ if and only if $X \supseteq Z$ and $Y \supseteq Z$

c) $X \supseteq Y \cup Z$ if and only if $X \supseteq Y$ and $X \supseteq Z$

d) $X \supseteq Y \cap Z$ if and only if $X \supseteq Y$ or $X \supseteq Z$

e) $v_1 \supseteq v_2$ if and only if $v_1 = v_2$, for variables v_1, v_2.

For example, we find $a \cap (b \cup c) \supseteq (a \cap b) \cup (a \cap c) \not\supseteq a \cap (b \cup c)$.

Design an algorithm which tests whether or not $X \supseteq Y$, given two formulas X and Y in the free lattice.

▶ **20.** [*M22*] Prove that if u and v are nodes of a tree, u is an ancestor of v if and only if u precedes v in preorder and u follows v in postorder.

21. [*25*] Algorithm D controls the differentiation activity for binary operators, unary operators, and "nullary" operators, i.e., for trees whose nodes have degree 2, 1, or 0; but it does not indicate explicitly how the control would be handled for ternary operators and nodes of higher degree. (For example, exercise 17 suggests making addition and multiplication into operators with any number of operands.) Is it possible to extend Algorithm D in a simple way so that it will handle operators of degree more than 2?

▶ **22.** [*M26*] If T and T' are trees, let us say T *can be embedded in* T', written $T \subseteq T'$, if there is a one-to-one function f from the nodes of T into the nodes of T' such that f preserves both preorder and postorder. (In other words, u precedes v in preorder for T if and only if $f(u)$ precedes $f(v)$ in preorder for T', and the same holds for postorder. See Fig. 25.)

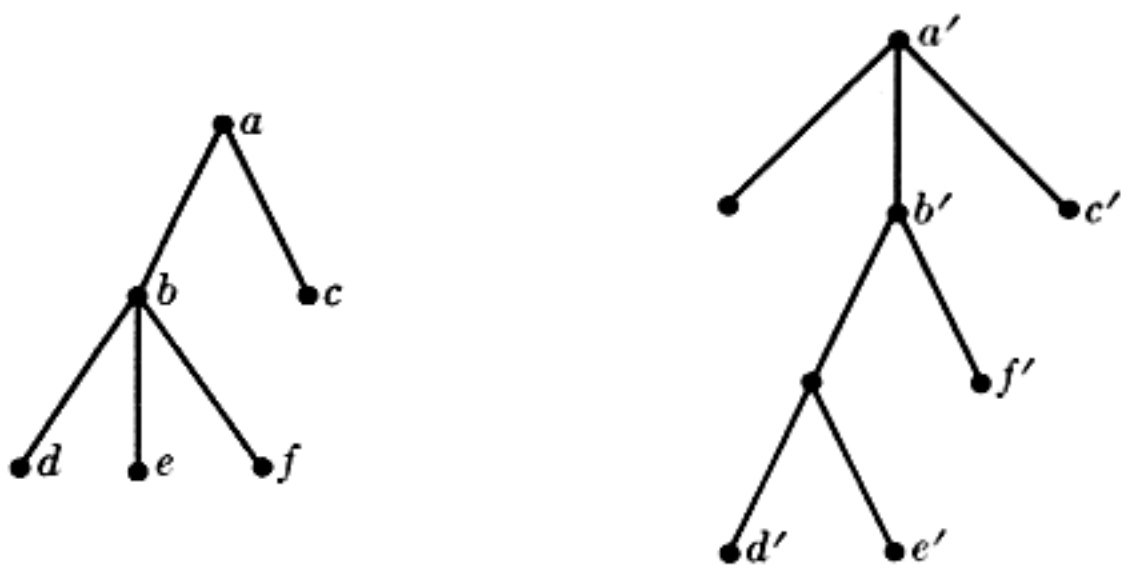


Fig. 25. One tree embedded in another (see exercise 22).

If T has more than one node, let $l(T)$ be the leftmost subtree of root (T) and let $r(T)$ be the rest of T, that is, T with $l(T)$ deleted. Prove that T can be embedded in T' if either T has just one node, or both T and T' have more than one node and either $T \subseteq l(T')$, or $T \subseteq r(T')$, or $l(T) \subseteq l(T')$ and $r(T) \subseteq r(T')$. Does the converse hold?

2.3.3. Other Representations of Trees

There are many ways to represent tree structures inside a computer besides the LLINK–RLINK (left son–right sibling) method given in the previous section. As usual, the proper choice of representation depends heavily on what kind of operations we want to perform on the trees. In this section we will consider a few of the possible tree representation methods that have proved to be useful.

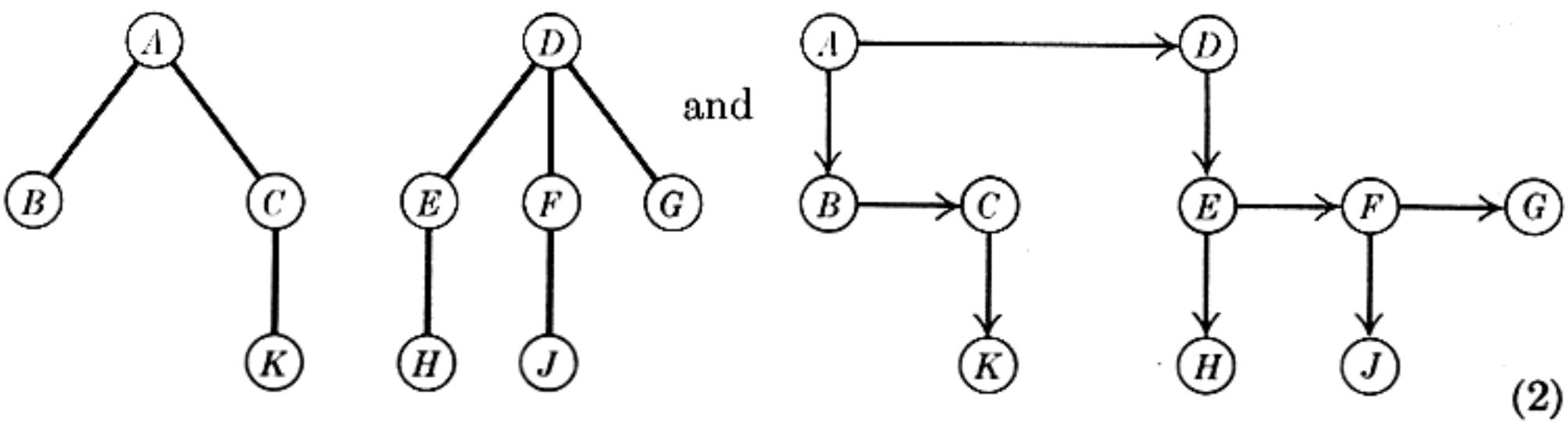
First we can use *sequential* memory techniques. As in the case of linear lists, this mode of allocation is most suitable when we want a compact representation

of a tree structure that is not going to be subject to radical dynamic changes in size or shape during program execution. There are many situations in which we need essentially constant tables of tree structures for reference within a program, and the desired form of these trees in memory depends on the way in which these tables are to be examined.

The two most common sequential representations of trees (and forests) correspond essentially to omission of `LLINK` or of `RLINK`; consecutive storage of the nodes takes the place of one type of link. Let us consider first the case that `LLINK`s are omitted, and look at the forest

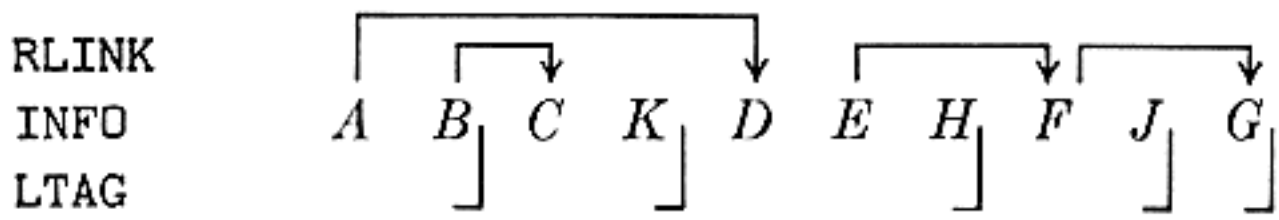
$$(A(B, C(K)),\ D(E(H), F(J), G)) \tag{1}$$

considered in the previous section, which has the tree diagrams



(2)

The *preorder sequential representation* has the nodes appearing in preorder, with the fields `INFO`, `RLINK`, and `LTAG` in each node:



(3)

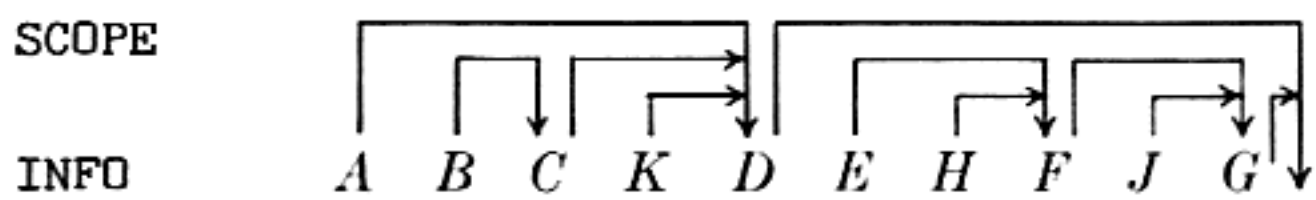
Here nonnull `RLINK`s have been indicated by arrows, and `LTAG` = "—" (for terminal nodes) is indicated by "⌋". `LLINK` is unnecessary, since it would either be null or it would point to the next item in sequence.

This representation has several interesting properties. In the first place, all subtrees of a node appear immediately after that node, so that all subtrees within the original forest appear in consecutive blocks. [Compare this with the "nested parentheses" in (1) and in Fig. 20(b).] In the second place, note that the `RLINK` arrows never cross each other in (3); this will be true in general, for in a binary tree all nodes between `X` and `RLINK(X)` in preorder lie in the left subtree of `X`, and so no outward arrows will emerge from that part of the tree. In the third place, we may observe that the `LTAG` field, which indicates whether a node is terminal or not, is redundant, since "⌋" occurs only at the end of the forest and just *preceding* every downward pointing arrow. Thus if we would proceed from left to right in (3), maintaining a suitable stack to remember the locations of `RLINK`s that have not yet been encountered, we could tell what the

value of LTAG must be. (This observation is interesting from a theoretical point of view, but in practice it is usually advantageous to include LTAG or some equivalent, since it takes only one bit of a node anyway.) The reader will find it interesting to compare (1) with (3).

Note that representation (3) involves some wasted space, since over half of the RLINK fields are equal to Λ for this particular forest. There are two common ways to make use of the wasted space:

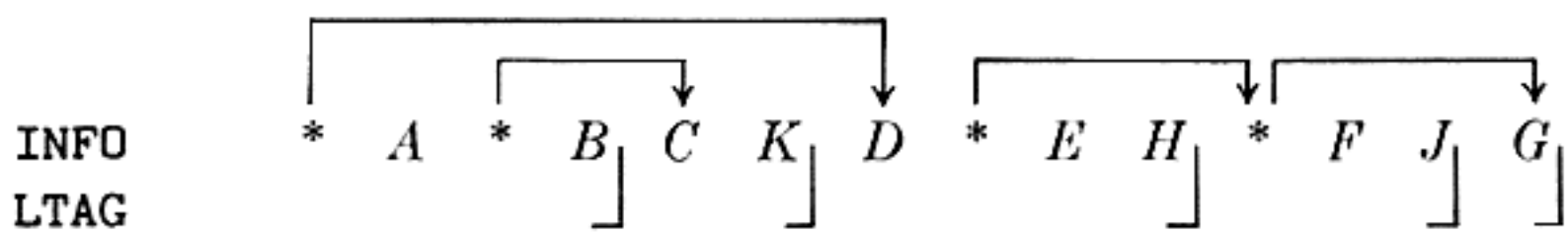
1) Fill in RLINK of each node to the address following the subtree below that node. The field is now often called "SCOPE" instead of RLINK, since it indicates the right boundary of the "influence" (descendants) of each node. Instead of (3), we would have



(4)

The arrows still do not cross each other. Furthermore, LTAG(X) = "—" is characterized by the condition SCOPE(X) = X + c, given that c is the number of words per node. One example of the use of this SCOPE idea appears in exercise 2.4–12.

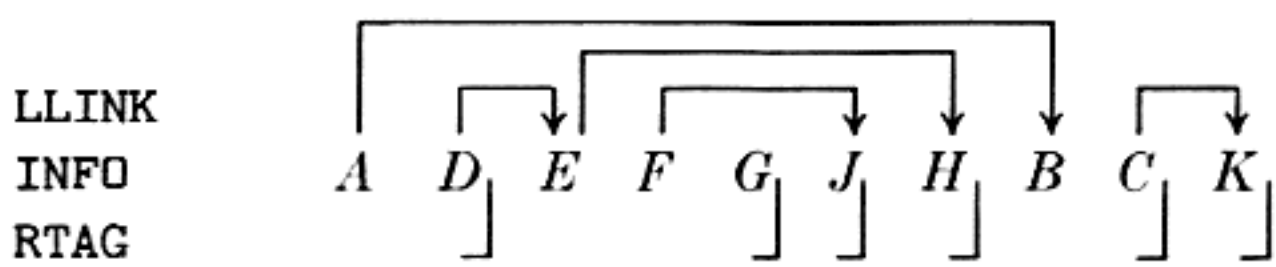
2) Decrease the size of each node by removing the RLINK field, and add special "link" nodes just before nodes that formerly had a nonnull RLINK:



(5)

Here "*" indicates the special link nodes, whose INFO somehow characterizes them as links pointing as shown by the arrows. If the INFO and RLINK fields of (3) occupy roughly the same amount of space, the net effect of the change to (5) is to consume less memory, since the number of "*" nodes is always less than the number of non-"*" nodes. Representation (5) is somewhat analogous to a sequence of instructions in a one-address computer like MIX, with the "*" nodes corresponding to conditional jump instructions.

The *family-order sequential representation* of a tree differs from the preorder representation in that it omits RLINK but retains LLINK. It has the nodes appearing in the *reverse* of endorder, which in the case of trees (2) would be



(6)

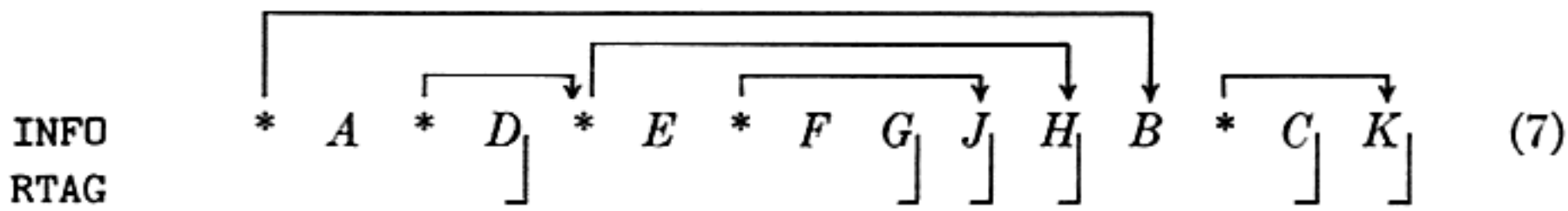
Reverse endorder may be called family-order because all sons of the same family are adjacent; thus, for example, E F G appear together in (6). (A similar

but not identical ordering has been called *level order* by G. Salton; see *CACM* **5** (1962), 103–114.) Family-order for any forest may be recursively defined as follows:

a) visit the root of the first tree;
b) traverse the remaining trees, in family-order;
c) traverse the subtrees of the root of the first tree, in family-order.

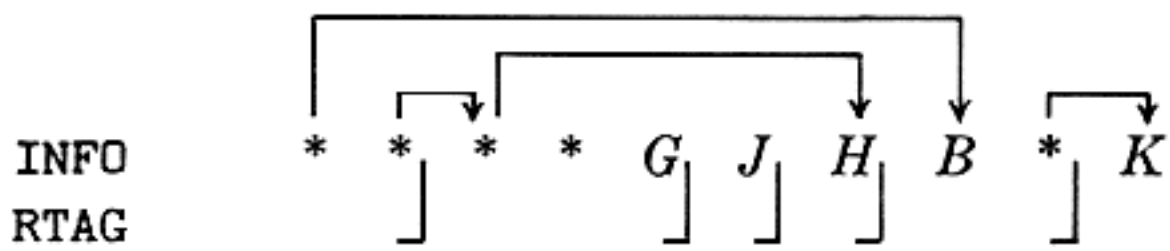
(Compare with the definition of endorder in the previous section.)

Representation (6) is in a sense the "natural" analog, for trees, of the sequential representation of linear lists, if we think of a forest as a list of trees, and if we think of each tree as one item of INFO plus a list of subtrees. Each family is a sequentially-stored list. The family-order representation has interesting properties analogous to those of the preorder representation: (a) The arrows never cross each other (see exercise 1). (b) The RTAG field, which marks the end of each family, is redundant, just as LTAG was redundant in the case of preorder. (c) To compress memory in certain circumstances, a representation analogous to (5) may be used:



(7)

In this case there are often situations where nonterminal nodes carry no information, so the node following each "*" node may be deleted. Thus, if the names A, D, E, F, and C do not need to be stored, (7) becomes



(8)

which is like (6) with only INFO and RTAG fields in each node.

The reader will easily see how to design algorithms that traverse and analyze trees represented sequentially as above, since the LLINK and RLINK information is essentially available just as though we had a fully linked tree structure.

A third method, *postorder sequential representation*, is somewhat different from the above two. We list the nodes in postorder and give the *degree* of each node instead of links:

DEGREE	0	0	1	2	0	1	0	1	0	3
INFO	B	K	C	A	H	E	J	F	G	D

(9)

For a proof that this is sufficient to characterize the tree structure, see exercise 2.3.2–10. This order is useful for the evaluation of certain functions defined on the nodes of a tree.

Algorithm F (*Evaluate a locally defined function in a tree*). Suppose f is a function of the nodes of a tree, such that the value of f at a node x depends only on x and the values of f on the sons of x. The following algorithm, using an auxiliary stack, evaluates f at each node of a nonempty forest.

F1. [Initialize.] Set the stack empty, and let P point to the first node of the forest in postorder.

F2. [Evaluate f.] Set $d \leftarrow$ DEGREE(P). (The first time this step is reached, d will be zero. In general, when we get to this point, it will always be true that the top d items of the stack are $f(x_d), \ldots, f(x_1)$—from the top of the stack downward—where $x_1, \ldots, x_d$ are the sons of NODE(P) from left to right.) Evaluate f(NODE(P)), using the values of $f(x_d), \ldots, f(x_1)$ found on the stack.

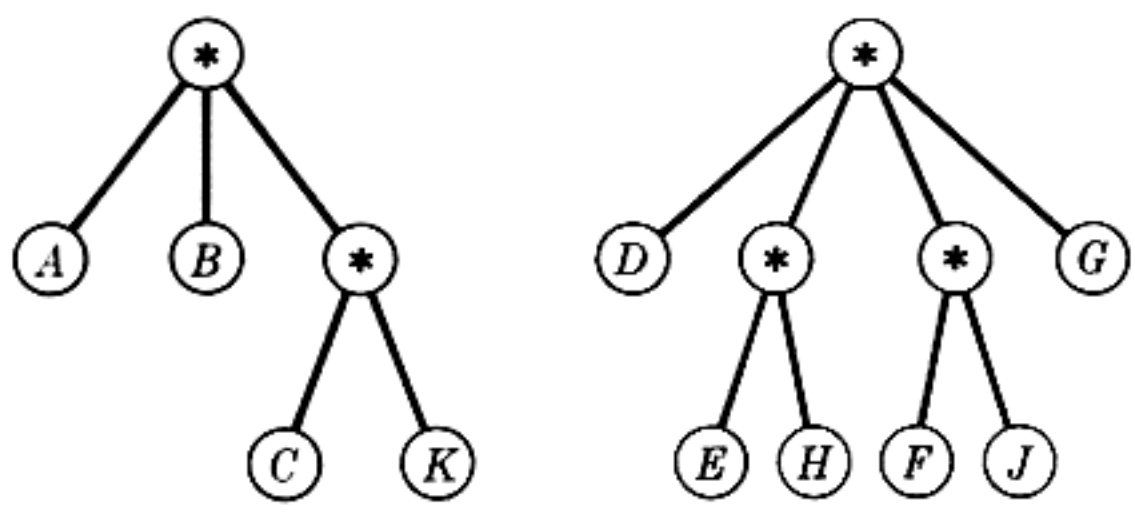
F3. [Update the stack.] Remove the top d items of the stack, and then put the value f(NODE(P)) on top of the stack.

F4. [Advance.] If P is the last node in postorder, terminate the algorithm. (Then the stack contains $f(\text{root}\,(T_m)), \ldots, f(\text{root}\,(T_1))$, from top to bottom, where $T_1, \ldots, T_m$ are the trees of the given forest.) Otherwise set P $\leftarrow$ P\$ [this would be simply P $\leftarrow$ P + 1 in the representation (9)], and return to step F2. ∎

The validity of Algorithm F follows by induction on the size of the trees processed (see exercise 17). This algorithm bears a striking similarity to the differentiation algorithm (2.3.2D) of the previous section, which evaluates a function of a closely related type. (See exercise 3.) The same idea is used in many interpretive routines in connection with the evaluation of arithmetic expressions in postfix notation; we will return to this topic in Chapter 8. See also exercise 18, which gives another important procedure similar to Algorithm F.

We have now seen that sequential representation of trees and forests is possible and meaningful in any of the three basic orders for the tree nodes: preorder, postorder, and (in reverse) endorder. There are also a number of linked forms of representation, which we will now consider.

The first idea is related to the transformation that takes (6) into (7): we remove the INFO fields from all nonterminal nodes and put this information as a new terminal node below the previous node. For example, the trees (2) would become



(10)

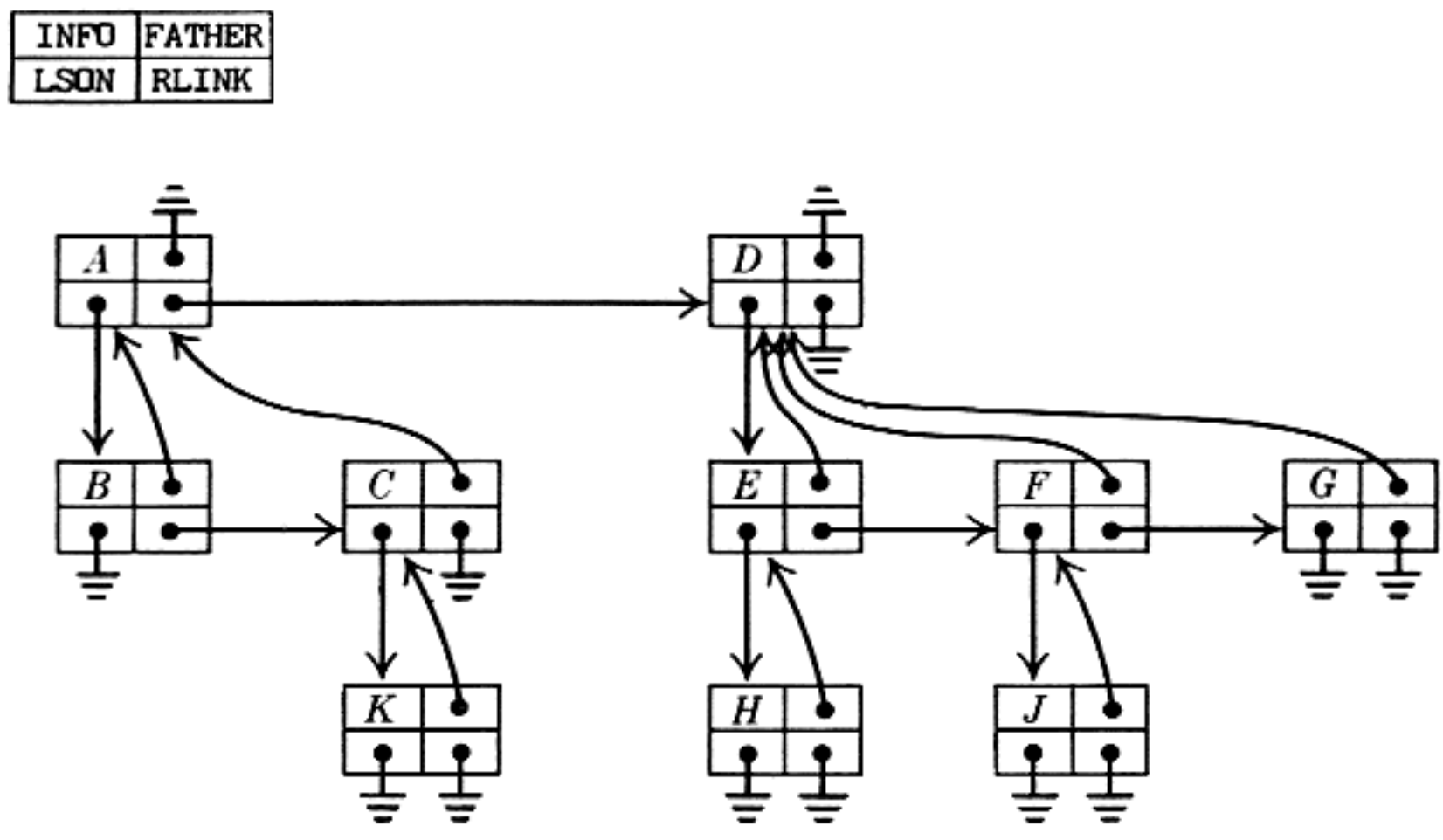
This new form shows that we may assume (without loss of generality) that all INFO in a tree structure appears in its terminal nodes. Therefore in the natural binary tree representation of Section 2.3.2, the LLINK and INFO fields are mutually exclusive and they can share the same field in each node. A node might have the fields

LTAG	LLINK or INFO	RLINK

where the sign LTAG tells whether the second field is a link or not. [Compare this representation with, for example, the two-word format of (10) in Section 2.3.2.] By cutting INFO down from 6 bytes to 3, we can fit each node into one word. However, note that there are now 15 nodes instead of 10; the forest (10) takes 15 words of memory while (2) takes 20, yet the latter has 60 bytes of INFO compared to 30 in the other. There is no real gain in memory space in (10) unless the excess INFO space was going to waste; the LLINKs replaced in (10) are removed at the expense of about the same number of new RLINKs in the added nodes. Precise details of the difference between the two representations are discussed in exercise 4.

In the standard binary tree representation of a tree, the LLINK field might be more accurately called the "LSON" field, since it points from a father node to his leftmost son. The leftmost son is usually the "youngest" of the sons in the tree, since it is easier to insert a node at the left of a family than at the right; so the abbreviation "LSON" may also be thought of as the "last son" or "least son."

Many applications of tree structures require rather frequent references upward in the tree as well as downward. A threaded tree gives us the ability to go upward, but not with great speed; occasionally, it is preferable to have a third link, FATHER, in each node. This leads to a *triply linked tree*, where each node has LSON, RLINK, and FATHER links. Figure 26 shows a triply linked tree representation of (2). For an example of the use of triply linked trees, see Section 2.4.



It is clear that the FATHER link all by itself is enough to specify any *oriented* tree (or forest) completely. For we can draw the diagram of the tree if we know all the upward links. Every node except the root has just one father, but there may be several sons; so it is simpler to give upward links than downward ones. Why then haven't we considered upward links much earlier in our discussion? The answer, of course, is that upward links by themselves are hardly adequate in most situations, since it is very difficult to tell quickly if a node is terminal or not, or to locate any of its sons, etc. There is, however, a very important application in which only upward links are sufficient: We now turn to a brief study of an elegant algorithm for dealing with equivalence relations, which is due to M. J. Fischer and B. A. Galler.

An *equivalence relation* "$\equiv$" is a relation between the elements of a set of objects S satisfying the following three properties for any objects x, y, and z (not necessarily distinct) in S:

i) If $x \equiv y$ and $y \equiv z$, then $x \equiv z$. (Transitivity.)
ii) If $x \equiv y$, then $y \equiv x$. (Symmetry.)
iii) $x \equiv x$. (Reflexivity.)

(Compare this with the definition of a "partial ordering" relation in Section 2.2.3; equivalence relations are quite different from partial orderings, in spite of the fact that two of the three defining properties are the same.) Examples of equivalence relations are the relation "$=$", the relation of congruence (modulo m) for integers, the relation of similarity between trees, as defined in Section 2.3.1, etc.

The equivalence problem is to read in pairs of equivalences and to determine later whether two particular elements can be proved equivalent or not on the basis of the given pairs. For example, suppose that S is the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and suppose that we are given the pairs

$$1 \equiv 5, \qquad 6 \equiv 8, \qquad 7 \equiv 2, \qquad 9 \equiv 8, \qquad 3 \equiv 7, \qquad 4 \equiv 2, \qquad 9 \equiv 3. \tag{11}$$

It follows that, for example, $2 \equiv 6$, since $2 \equiv 7 \equiv 3 \equiv 9 \equiv 8 \equiv 6$. But we cannot show that $1 \equiv 6$. In fact, the pairs (11) divide S into two classes

$$\{1, 5\} \qquad \text{and} \qquad \{2, 3, 4, 6, 7, 8, 9\}, \tag{12}$$

such that two elements are equivalent if and only if they belong to the same class. It is not difficult to prove that *any* equivalence relation partitions its set S into disjoint classes (called the "equivalence classes"), such that two elements are equivalent if and only if they belong to the same class.

Therefore to solve the equivalence problem it is a matter of keeping track of equivalence classes like (12). We may start with each element alone in its class, thus:

$$\{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}\ \{7\}\ \{8\}\ \{9\} \tag{13}$$

Now if we are given the relation $1 \equiv 5$, we put $\{1, 5\}$ together in a class. After processing the first three relations $1 \equiv 5$, $6 \equiv 8$, and $7 \equiv 2$, we will have changed (13) to

$$\{1, 5\} \quad \{2, 7\} \quad \{3\} \quad \{4\} \quad \{6, 8\} \quad \{9\}. \tag{14}$$

Now the pair $9 \equiv 8$ puts $\{6, 8, 9\}$ together, etc.

The problem is to find a good way to represent situations like (12), (13), and (14) within a computer so that we can efficiently perform the operations of merging classes together and of testing whether two given elements are in the same class. The algorithm below uses tree structures for this purpose: The elements of S become nodes of a forest, and two nodes are equivalent as a consequence of the pairs of equivalences read so far *if and only if they belong to the same tree.* This test is easy to make, since two elements are in the same tree if and only if they are below the same root element. Furthermore, it is easy to merge two trees together into one, by simply attaching one as a new subtree of the root of the other.

Algorithm E (*Process equivalence relations*). Let S be the set of numbers $\{1, 2, \ldots, n\}$, and let FATHER[1], FATHER[2], ..., FATHER[n] be integer variables. This algorithm inputs a set of relations such as (11) and adjusts the FATHER table to represent a set of trees, so that two elements are equivalent as a consequence of the given relations if and only if they belong to the same tree. (*Note:* In a more general situation, the elements of S would be symbolic names instead of simply the numbers from 1 to n; then a search routine, as in Chapter 6, would locate nodes corresponding to the elements of S, and FATHER would be a field in each node. The modifications for this more general case are straightforward.)

E1. [Initialize.] Set FATHER[k] $\leftarrow 0$ for $1 \le k \le n$. [This means all trees initially consist of a root alone, as in (13).]

E2. [Input new pair.] Get the next pair of equivalent elements "$j \equiv k$" from the input. If the input is exhausted, the algorithm terminates.

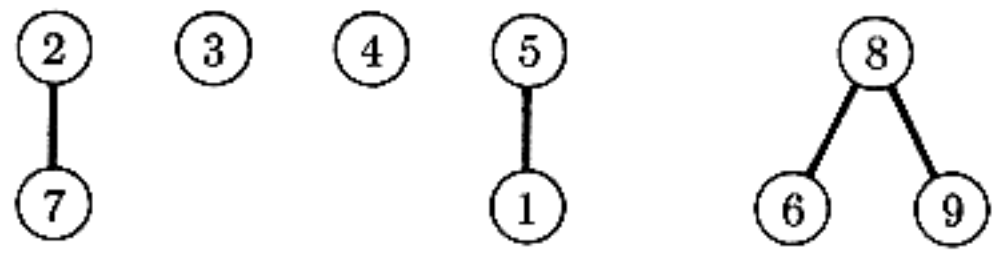
E3. [Find roots.] If FATHER[j] $\neq 0$, set $j \leftarrow$ FATHER[j] and repeat this step. If FATHER[k] $\neq 0$, set $k \leftarrow$ FATHER[k] and repeat this step. (After this operation, j and k have moved up to the roots of two trees which are to be made equivalent. The input relation $j \equiv k$ was redundant if and only if we now have $j = k$.)

E4. [Merge trees.] If $j \neq k$, set FATHER[j] $\leftarrow k$. Go back to step E2. ▮

The reader should try this algorithm on the input (11). After processing $1 \equiv 5$, $6 \equiv 8$, $7 \equiv 2$, and $9 \equiv 8$, we will have

$$\begin{array}{rccccccccc} \text{FATHER}[k]: & 5 & 0 & 0 & 0 & 0 & 8 & 2 & 0 & 8 \\ k: & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \tag{15}$$

which represents the trees



(16)

After this point, the remaining relations of (11) cause some more interesting things to happen; see exercise 9.

This equivalence problem occurs in many applications. A more general version of the problem which arises when a compiler processes "equivalence declarations" in languages like FORTRAN is discussed in exercise 11.

There are still more ways to represent trees in computer memory. Recall that we discussed three principal methods for representing linear lists in Section 2.2: the "straight" representation with terminal link Λ, the "circularly" linked lists, and the "doubly" linked lists. The representation of unthreaded binary trees described in Section 2.3.1 corresponds to a "straight" representation in both LLINKs and RLINKs. It is possible to get eight other binary tree representations by independently using any of these three methods in the LLINK and RLINK directions. For example, Fig. 27 shows what we get if circular linking is used in both directions. If circular links are used throughout as in the figure, we have what is called a *ring structure;* ring structures have proved to be quite flexible in a number of applications. The proper choice of representation depends, as always, on the type of insertions, deletions, and traversals that are needed in the algorithms that manipulate these structures. A reader who has looked over the examples given so far in this chapter should have no difficulty understanding when to use and how to deal with any of these memory representations.

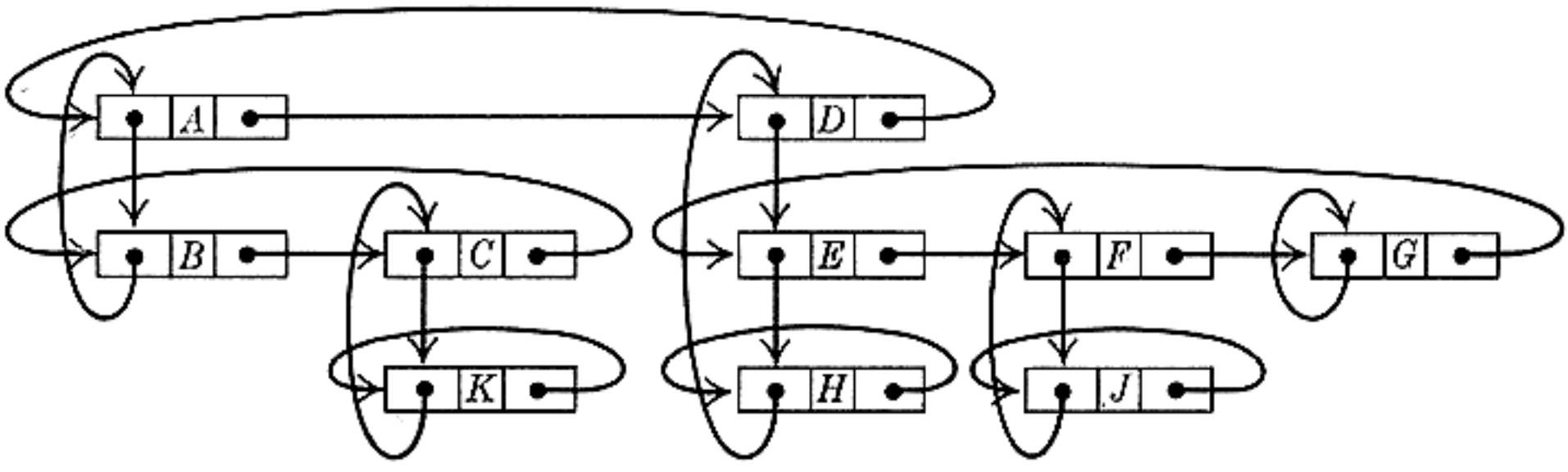


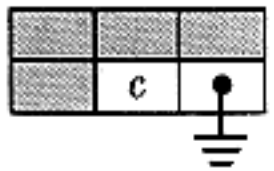
Fig. 27. A ring structure.

We close this section with an example of modified doubly linked ring structures applied to a problem we have considered before: arithmetic on polynomials. Algorithm 2.2.4A performs the addition of one polynomial to another, given that the two polynomials are expressed as circular lists, and various other algorithms in that section give other operations on polynomials; however, the polynomials are restricted to at most three variables. When multi-variable polynomials are

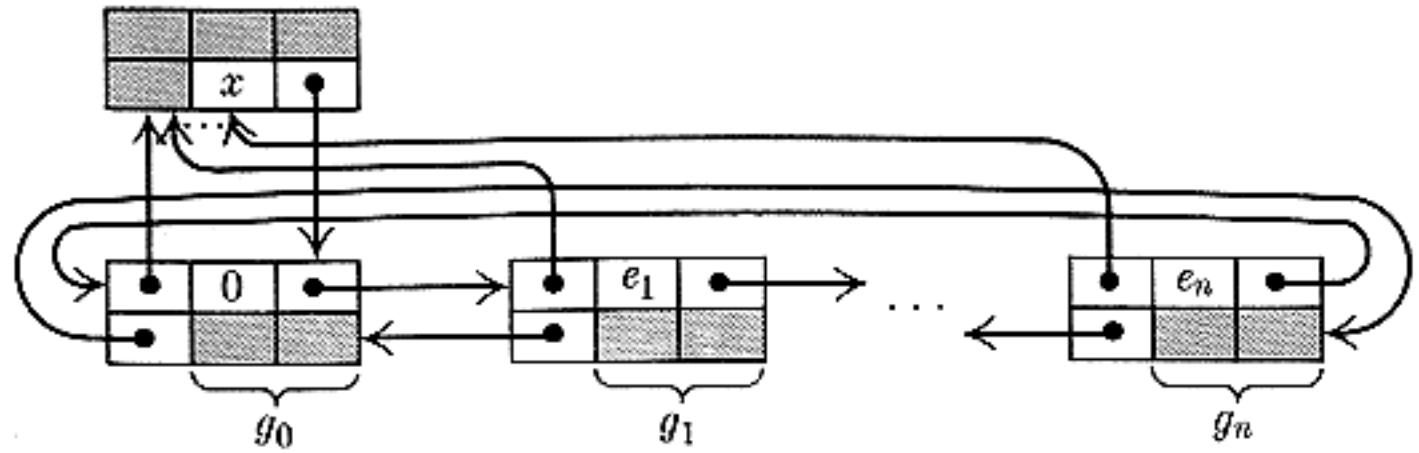
(a) Fields

UP	EXP	RIGHT
LEFT	CV	DOWN

(b) Polynomial $= c$ (constant)



(c) Polynomial $= g_0 + g_1 x^{e_1} + g_2 x^{e_2} + \cdots + g_n x^{e_n}$



(d) Example: $3 + x^2 + xyz + z^3 - 3xz^3$

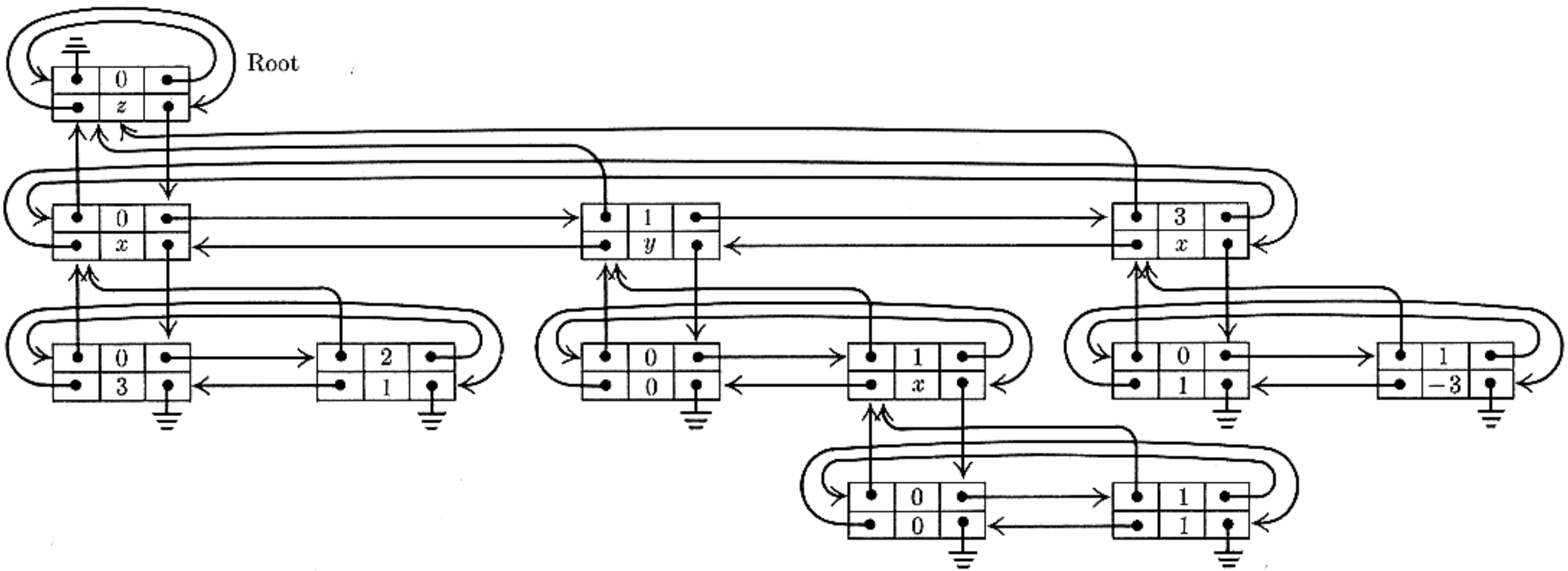
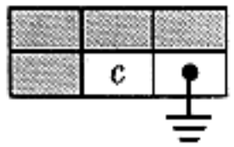


Fig. 28. Representation of polynomials using four-directional links. Shaded areas of nodes indicate information irrelevant in the context considered.

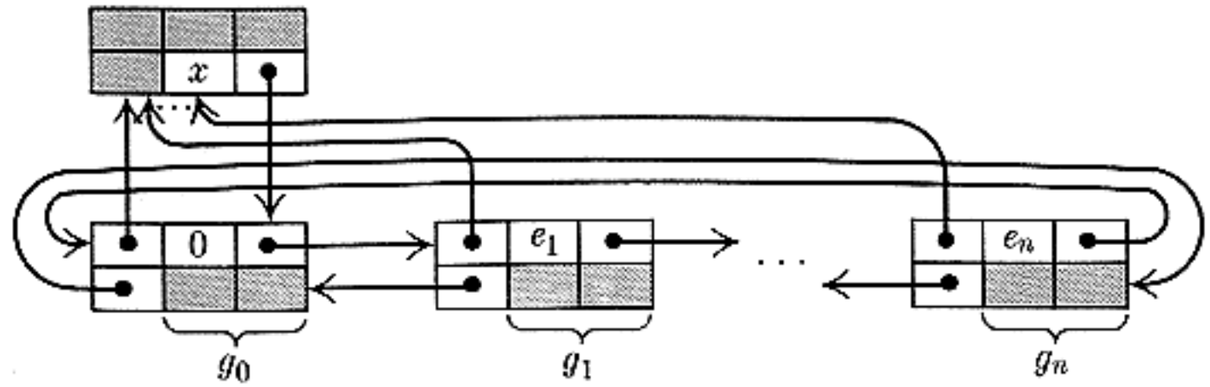
(a) Fields

UP	EXP	RIGHT
LEFT	CV	DOWN

(b) Polynomial $= c$ (constant)



(c) Polynomial $= g_0 + g_1 x^{e_1} + g_2 x^{e_2} + \cdots + g_n x^{e_n}$



(d) Example: $3 + x^2 + xyz + z^3 - 3xz^3$

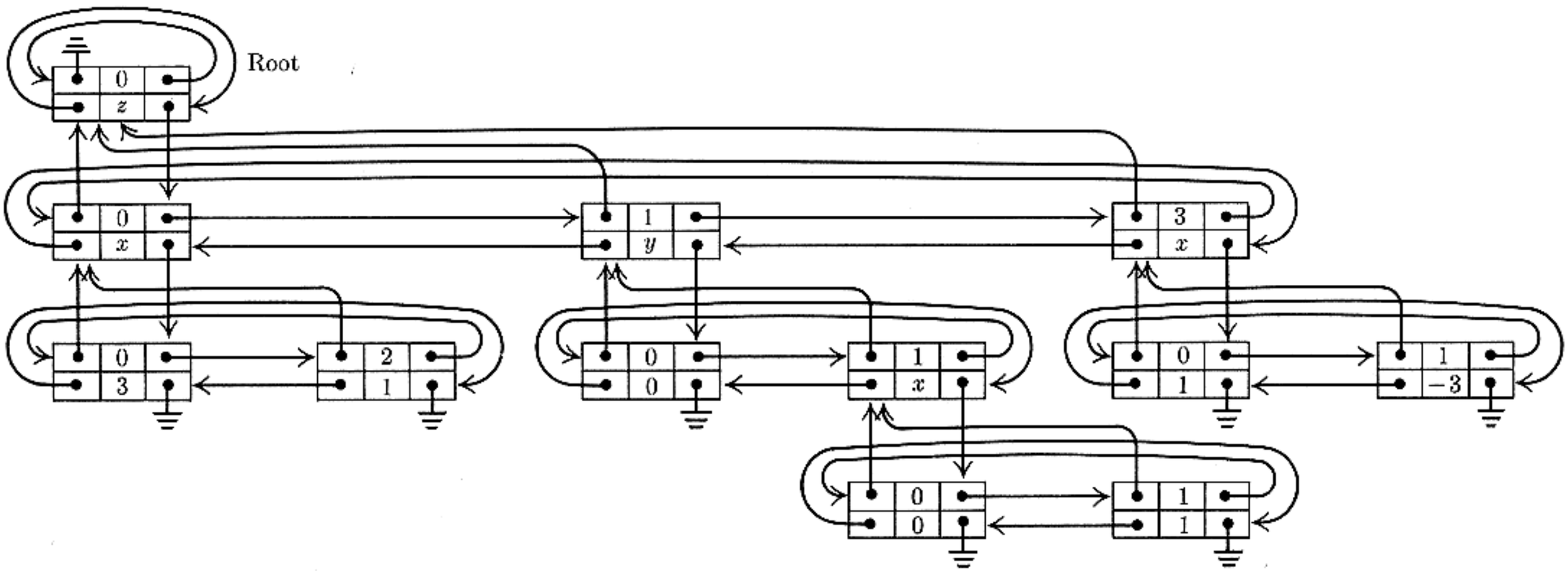


Fig. 28. Representation of polynomials using four-directional links. Shaded areas of nodes indicate information irrelevant in the context considered.

involved, it is often more appropriate to use a tree structure instead of a linear list.

A polynomial either is a constant or has the form

$$\sum_{0 \le j \le n} g_j x^{e_j},$$

where x is a variable, $n > 0$, $0 = e_0 < e_1 < \cdots < e_n$, and $g_0, \ldots, g_n$ are polynomials involving only variables alphabetically less than x; $g_1, \ldots, g_n$ are not zero. This definition of polynomials lends itself to tree representation as indicated in Fig. 28. Nodes have six fields, which in the case of MIX might fit in three words:

+	0	LEFT	RIGHT
+	EXP	UP	DOWN
CV			

(17)

Here LEFT, RIGHT, UP, and DOWN are links, EXP is an integer representing an exponent, and CV is either a constant (coefficient) or the alphabetic name of a variable.

The following algorithm illustrates traversal, insertion, and deletion in such a four-way-linked tree, so it bears careful study.

Algorithm A (*Addition of polynomials*). This algorithm adds polynomial(P) to polynomial(Q), assuming that P and Q are pointer variables which link to the roots of polynomial trees having the form shown in Fig. 28. At the conclusion of the algorithm, polynomial(P) will be unchanged, and polynomial(Q) will contain the sum.

A1. [Test type of polynomial.] If DOWN(P) $=\Lambda$ (i.e., if P points to a constant), then set Q $\leftarrow$ DOWN(Q) zero or more times until DOWN(Q) $=\Lambda$ and go to A3. If DOWN(P) $\neq \Lambda$, then if DOWN(Q) $=\Lambda$ or if CV(Q) $<$ CV(P), go to A2. Otherwise if CV(Q) $=$ CV(P), set P $\leftarrow$ DOWN(P), Q $\leftarrow$ DOWN(Q) and repeat this step; if CV(Q) $>$ CV(P), set Q $\leftarrow$ DOWN(Q) and repeat this step. (Step A1 either finds two matching terms of the polynomials or else determines that an insertion of a new variable must be made into this part of polynomial(Q).)

A2. [Downward insertion.] Set R $\Leftarrow$ AVAIL, S $\leftarrow$ DOWN(Q). If S $\neq \Lambda$, set UP(S) $\leftarrow$ R, S $\leftarrow$ RIGHT(S), and if EXP(S) $\neq 0$, repeat this operation until ultimately EXP(S) $= 0$. Set UP(R) $\leftarrow$ Q, DOWN(R) $\leftarrow$ DOWN(Q), LEFT(R) $\leftarrow$ R, RIGHT(R) $\leftarrow$ R, CV(R) $\leftarrow$ CV(Q), and EXP(R) $\leftarrow 0$. Finally, set CV(Q) $\leftarrow$ CV(P) and DOWN(Q) $\leftarrow$ R, and return to A1. (We have inserted a zero polynomial just below NODE(Q), to obtain a match with a corresponding polynomial found within P's tree. The link manipulations done in this step

are straightforward and may be derived easily using "before-and-after" diagrams, as explained in Section 2.2.3.)

A3. [Match found.] (At this point, P and Q point to corresponding terms of the given polynomials, so addition is ready to proceed.) Set CV(Q) $\leftarrow$ CV(Q) + CV(P). If this sum is zero and if EXP(Q) $\neq 0$, go to step A8.

A4. [Advance to left.] (After successfully adding a term, we look for the next term to add.) Set P $\leftarrow$ LEFT(P). If EXP(P) $= 0$, go to A6. Otherwise set Q $\leftarrow$ LEFT(Q) one or more times until EXP(Q) $\leq$ EXP(P). If then EXP(Q) $=$ EXP(P), return to step A1.

A5. [Insert to right.] Set R $\Leftarrow$ AVAIL. Set UP(R) $\leftarrow$ UP(Q), DOWN(R) $\leftarrow \Lambda$, LEFT(R) $\leftarrow$ Q, RIGHT(R) $\leftarrow$ RIGHT(Q), LEFT(RIGHT(R)) $\leftarrow$ R, RIGHT(Q) $\leftarrow$ R, EXP(R) $\leftarrow$ EXP(P), CV(R) $\leftarrow 0$, and Q $\leftarrow$ R. Return to step A1. (It was found necessary to insert a new term in the current row, just to the right of NODE(Q), in order to match a corresponding exponent in polynomial(P). As in step A2, a "before-and-after" diagram makes the above operations clear.)

A6. [Return upward.] (A row of polynomial(P) has now been completely traversed.) If UP(P) $= \Lambda$, the algorithm terminates; otherwise set P $\leftarrow$ UP(P).

A7. [Move Q up to right level.] If UP(P) $= \Lambda$, the algorithm terminates; otherwise set Q $\leftarrow$ UP(Q) zero or more times until CV(UP(Q)) $=$ CV(UP(P)). Return to step A4.

A8. [Delete zero term.] Set R $\leftarrow$ Q, Q $\leftarrow$ RIGHT(Q), LEFT(Q) $\leftarrow$ LEFT(R), and AVAIL $\Leftarrow$ R. (Cancellation occurred, so a row element of polynomial(Q) is deleted.) If now EXP(LEFT(P)) $= 0$ and Q $=$ LEFT(Q), go to A9; otherwise return to A4.

A9. [Delete constant polynomial.] (Cancellation has caused a polynomial to reduce to a constant, so a row of polynomial(Q) is deleted.) Set R $\leftarrow$ Q, Q $\leftarrow$ UP(Q), DOWN(Q) $\leftarrow$ DOWN(R), CV(Q) $\leftarrow$ CV(R), and AVAIL $\Leftarrow$ R. Set S $\leftarrow$ DOWN(Q); if S $\neq \Lambda$, set UP(S) $\leftarrow$ Q, S $\leftarrow$ RIGHT(S), and if EXP(S) $\neq 0$, repeat this operation until ultimately EXP(S) $= 0$.

A10. [Zero detected?] If DOWN(Q) $= \Lambda$, CV(Q) $= 0$, and EXP(Q) $\neq 0$, set P $\leftarrow$ UP(P) and go to A8; otherwise go to A6. ▮

This algorithm will actually run much faster than Algorithm 2.2.3A if polynomial(P) has few terms and polynomial(Q) has many, since it is not necessary to pass over all of polynomial(Q) during the addition process. The reader will find it instructive to simulate Algorithm A by hand, adding the polynomial $xy - x^2 - xyz - z^3 + 3xz^3$ to the polynomial shown in Fig. 28. (This case does not demonstrate the efficiency of the algorithm, but it makes the algorithm go through all of its paces by showing the difficult situations which must be handled.) For further commentary on Algorithm A, see exercises 12 and 13.

No claim is being made here that the representation shown in Fig. 28 is the "best" for polynomials in several variables; in Chapter 8 we will consider another format for polynomial representation, together with arithmetic algorithms using an auxiliary stack, which have significant advantages of conceptual simplicity when compared to Algorithm A. Our main interest in Algorithm A is the way it typifies manipulations on trees with many links.

EXERCISES

▶ **1.** [*20*] Show that for any family-order sequential representation, as in (6), the LLINK arrows never cross each other.

2. [*22*] (Burks, Warren, and Wright, *Math. Comp.* **8** (1954), 46–50.) The trees (2) stored in *preorder* with degrees would be

DEGREE	2	0	1	0	3	1	0	1	0	0
INFO	*A*	*B*	*C*	*K*	*D*	*E*	*H*	*F*	*J*	*G*

[cf. (9) where postorder was used]. Design an algorithm analogous to Algorithm F to evaluate a locally defined function of the nodes by going from right to left in this representation.

▶ **3.** [*24*] Modify Algorithm 2.3.2D so that it follows the ideas of Algorithm F, placing the derivatives it computes as intermediate results on a stack, instead of recording their locations in an anomalous fashion as is done in step D3. (Cf. exercise 2.3.2–21.) The stack may be maintained by using the RLINK field in the root of each derivative.

4. [*18*] The trees (2) contain 10 nodes, five of which are terminal. Representation of these trees in the normal binary-tree fashion involves 10 LLINK fields and 10 RLINK fields (one for each node). Representation of these trees in the form (10), where LLINK and INFO share the same space in a node, requires 5 LLINKs and 15 RLINKs. There are 10 INFO fields in each case.

Given a forest with n nodes, m of which are terminal, compare the total number of LLINKs and RLINKs that must be stored using these two methods of tree representation.

5. [*16*] A triply linked tree, as shown in Fig. 26, contains FATHER, LSON, and RLINK fields in each node, with liberal use of Λ-links when there is no appropriate node to mention in the FATHER, LSON, or RLINK field. Would it be a good idea to extend this representation to a *threaded* tree, by putting "thread" links in place of the null LSON and RLINK entries, as we did in Section 2.3.1?

▶ **6.** [*24*] Suppose that the nodes of an *oriented* forest have three link fields, FATHER, LSON, and RLINK, but only the FATHER link has been set up to indicate the tree structure. The LSON field of each node is Λ and the RLINK fields are set as a linear list which simply links the nodes together in some order. The link variable FIRST points to the first node, and the last node has RLINK = Λ.

Design an algorithm which goes through these nodes and fills in the LSON and RLINK fields compatible with the FATHER links, so that a triply linked tree representation like that in Fig. 26 is obtained. Also, reset FIRST so that it now points to the root of the first tree in this representation.

7. [*15*] What classes would appear in (12) if the relation $9 \equiv 3$ had not been given in (11)?

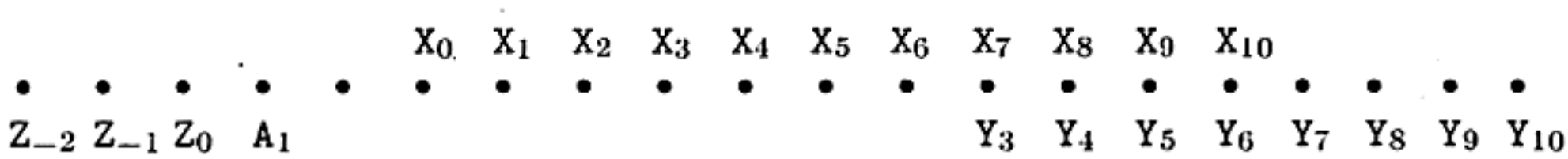
8. [*20*] Algorithm E sets up a tree structure which represents the given pairs of equivalences, but the text does not mention explicitly how the result of Algorithm E can be used. Design an algorithm which answers the question, "Is $j \equiv k$?", assuming that $1 \le j \le n$, $1 \le k \le n$, and that Algorithm E has set up the FATHER table for some set of equivalences.

9. [*21*] Give a table analogous to (15) and a diagram analogous to (16) which shows the trees present after Algorithm E has processed all of the pairs of equivalences in (11) from left to right.

10. [*M48*] Perform an analysis of the efficiency of Algorithm E when it is given random pairs of equivalences in random order. In particular, what is the average level of the nodes in the trees, after Algorithm E has been in operation?

▶ **11.** [*24*] ("Equivalence declarations.") Several compiler languages provide a facility for overlapping the memory locations assigned to sequentially stored tables. The programmer gives the compiler pairs of relations of the form "X[j] $\equiv$ Y[k]" which means variable X[$j + s$] is to be assigned to the same location as variable Y[$k + s$] for all s. Each variable is further given a range of allowable subscripts: "ARRAY X[ℓ:u]" means that space is to be set aside in memory for the table entries X[ℓ], X[$\ell + 1$], . . . , X[u]. For each equivalence class of variables, the compiler reserves as small a block of consecutive memory locations as possible, to contain all the table entries for the allowable subscript values of these variables.

For example, suppose we have ARRAY X[0:10], ARRAY Y[3:10], ARRAY A[1:1], and ARRAY Z[−2:0], plus the equivalences X[7] $\equiv$ Y[3], Z[0] $\equiv$ A[0], and Y[1] $\equiv$ A[8]. We must set aside 20 consecutive locations



for these variables. Note that the location following A[1] is not an "allowable" subscript value for any of the arrays.

The object of this exercise is to modify Algorithm E so that it applies to the more general situation just described. Assume that we are writing a compiler for such a language, and the tables inside our compiler program itself have one node for each array, containing the fields NAME, FATHER, DELTA, LBD, and UBD. Assume that the compiler program has previously processed all the ARRAY declarations, so that if "ARRAY X[ℓ:u]" has appeared and if P points to the node for X, then

$$\text{NAME(P)} = \text{"X"}, \quad \text{FATHER(P)} = \Lambda, \quad \text{DELTA(P)} = 0,$$
$$\text{LBD(P)} = \ell, \quad \text{UBD(P)} = u.$$

The problem is to design an algorithm which processes the equivalence declarations, so that after this algorithm has been performed

FATHER(P) = Λ means that locations X[LBD(P)], . . . , X[UBD(P)] are to be reserved in memory for this equivalence class;

FATHER(P) = Q $\neq \Lambda$ means location X[k] equals location Y[k + DELTA(P)], where NAME(Q) = "Y".

For example, before the equivalences listed above we might have the nodes

P	NAME(P)	FATHER(P)	DELTA(P)	LBD(P)	UBD(P)
α	X	Λ	0	0	10
β	Y	Λ	0	3	10
γ	A	Λ	0	1	1
δ	Z	Λ	0	−2	0

After the equivalences are processed, the nodes might appear thus:

α	X	Λ	*	−5	14
β	Y	α	4	*	*
γ	A	δ	0	*	*
δ	Z	α	−3	*	*

("*" denotes irrelevant information).

Design an algorithm which makes this transformation. Assume that inputs to your algorithm have the form (P, j, Q, k), denoting "X[j] $\equiv$ Y[k]", where NAME(P) = "X" and NAME(Q) = "Y". Be sure to check whether the equivalences are contradictory; e.g., X[1] $\equiv$ Y[2] contradicts X[2] $\equiv$ Y[1].

12. [*21*] At the beginning of Algorithm A, the variables P and Q point to the roots of two trees. Let P_0 and Q_0 denote the values of P and Q before execution of Algorithm A. (a) After the algorithm terminates, is Q_0 always the address of the root of the sum of the two given polynomials? (b) After the algorithm terminates, have P and Q returned to their original values P_0, Q_0?

▶ **13.** [*M29*] Give an informal proof that at the beginning of step A8 of Algorithm A we always have EXP(P) = EXP(Q) and CV(UP(P)) = CV(UP(Q)). (This fact is important to the proper understanding of that algorithm.)

14. [*M32*] Give a formal proof (or disproof) of the validity of Algorithm A.

15. [*40*] Design an algorithm to compute the product of two polynomials represented as in Fig. 28.

▶ **16.** [*28*] Design an algorithm which, given tables INFO1[j], RLINK[j] for $1 \le j \le n$, corresponding to preorder sequential representation, forms tables INFO2[j], DEGREE[j] for $1 \le j \le n$, corresponding to postorder sequential representation. For example, your algorithm should transform

j	1	2	3	4	5	6	7	8	9	10
INFO1[j]	A	B	C	K	D	E	H	F	J	G
RLINK[j]	5	3	0	0	0	8	0	10	0	0

[cf. (3)] into

INFO2[j]	B	K	C	A	H	E	J	F	G	D
DEGREE[j]	0	0	1	2	0	1	0	1	0	3

[cf. (9)].

17. [*M24*] Prove the validity of Algorithm F.

18. [*25*] Algorithm F evaluates a "bottom-up" locally-defined function, namely, one which should be evaluated at the sons of a node before it is evaluated at the node. A "top-down" locally-defined function f is one in which the value of f at a node x depends only on x and the value of f at the *father* of x. Using an auxiliary stack, design an algorithm analogous to Algorithm F which evaluates a "top-down" function f at each node of a tree. (Like Algorithm F, your algorithm should work efficiently on trees which have been stored in *postorder* with degrees, as in (9).)

*2.3.4. Basic Mathematical Properties of Trees

Tree structures have been the object of extensive mathematical investigations for many years, long before the advent of computers, and many interesting facts have been discovered about them. In this section we will survey the mathematical theory of trees, which not only gives us more insight into the nature of tree structures but also has important applications to computer algorithms.

The material which follows comes mostly from a larger area of mathematics known as the theory of graphs. Unfortunately, there is as yet no standard terminology in this field, and so the author has followed the usual practice of contemporary books on graph theory, namely to use words that are similar but not identical to the terms used in all *other* books on graph theory. An attempt has been made in the following subsections (and, indeed, throughout this book) to choose short, descriptive words for the important concepts, selected from those which are in reasonably common use and which do not sharply conflict with other common terminology. The nomenclature used here is also biased towards computer applications; thus, an electrical engineer may prefer to call a "tree" what we call a "free tree," but we want the shorter term "tree" to stand for the concept which is generally used in the computer literature and which is so much more important in computer applications. If we were to follow the terminology of some authors on graph theory, we would have to say "finite labeled rooted ordered tree" instead of just "tree," and "topological bifurcating arborescence" instead of "binary tree"!

2.3.4.1. Free trees. A *graph* is generally defined to be a set of points (called *vertices*) together with a set of lines (called *edges*) joining certain pairs of distinct vertices. There is at most one edge joining any pair of vertices. Two vertices are called *adjacent* if there is an edge joining them. If V and V' are vertices and if $n \geq 0$, we say that $(V_0, V_1, \ldots, V_n)$ is a *path* of length n from V to V' if $V = V_0$, V_k is adjacent to V_{k+1} for $0 \leq k < n$, and $V_n = V'$. The path is *simple* if $V_0, V_1, \ldots, V_{n-1}$ are distinct and if $V_1, \ldots, V_{n-1}, V_n$ are distinct. A graph is *connected* if there is a path between any two vertices of the graph. A *cycle* is a simple path of length three or more from a vertex to itself.

These definitions are illustrated in Fig. 29, which shows a connected graph with five vertices and six edges. Vertex C is adjacent to A but not to B; there are two paths of length two from B to C, namely (B, A, C) and (B, D, C). There are several cycles, including (B, D, E, B).

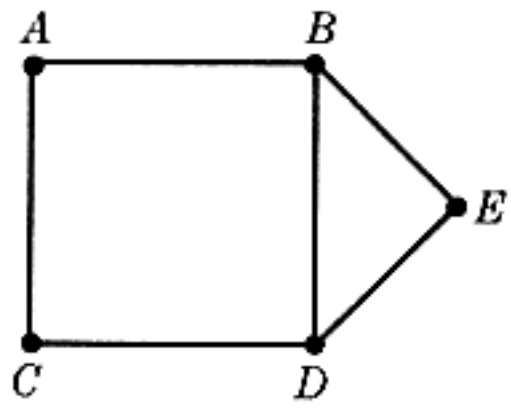


Fig. 29. A graph.

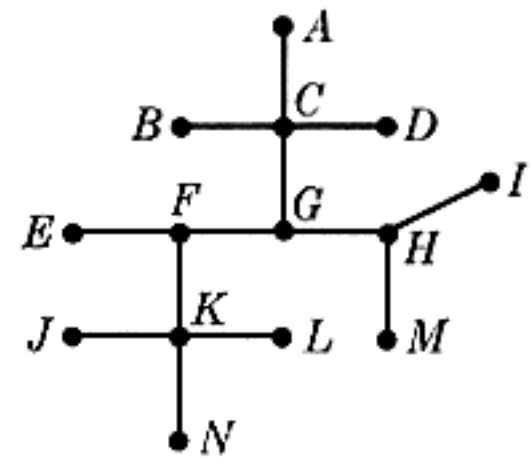


Fig. 30. A free tree.

A *free tree* or "unrooted tree" (Fig. 30) is defined to be a connected graph with no cycles. This definition applies to infinite graphs as well as finite ones, although for computer applications we naturally are most concerned with finite trees. There are many equivalent ways to define a free tree; some of these appear in the following well-known theorem:

Theorem A. *If G is a graph, the following statements are equivalent:*

a) *G is a free tree;*

b) *G is connected, but if any edge is deleted, the resulting graph is no longer connected.*

c) *If V and V' are distinct vertices of G, there is exactly one simple path from V to V'.*

Furthermore, if G is finite, containing exactly $n > 0$ vertices, the following properties are also equivalent to (a), (b), and (c):

d) *G contains no cycles and has $n - 1$ edges.*

e) *G is connected and has $n - 1$ edges.*

Proof. (a) implies (b), for if the edge VV' is deleted but G is still connected, there must be a simple path $(V, V_1, \ldots, V')$—see exercise 2—and then $(V, V_1, \ldots, V', V)$ would be a cycle in G.

(b) implies (c), for there is at least one simple path from V to V'. And if there were two such paths $(V, V_1, \ldots, V')$ and $(V, V_1', \ldots, V')$, we could find the smallest k for which $V_k \neq V_k'$; deleting the edge $V_{k-1}V_k$ would not disconnect the graph, since there would still be a path $(V_{k-1}, \ldots, V, V_1', \ldots, V', \ldots, V_k)$ from V_{k-1} to V_k which does not use the deleted edge.

(c) implies (a), for if G contains a cycle $(V, V_1, \ldots, V)$, there are two simple paths from V to V_1.

To show that (d) and (e) are also equivalent to (a), (b), and (c), let us first prove an auxiliary result: If G is any finite graph which has no cycles and at least one edge, then there is at least one vertex which is adjacent to exactly one other vertex. For we take an arbitrary vertex V_1 and an adjacent vertex V_2; for $k \geq 2$ either V_k is adjacent to V_{k-1} and no other, or it is adjacent to a vertex which we may call $V_{k+1} \neq V_{k-1}$. Since there are no cycles, $V_1, V_2, \ldots, V_{k+1}$ must be distinct vertices, so this process must ultimately terminate.

Now assume G is a tree with $n > 1$ vertices, and let V_n be a vertex which is adjacent to only one other vertex, namely V_{n-1}. If we delete V_n and the edge $V_{n-1}V_n$, the remaining graph G' is a tree, since V_n appears in no simple path of G except as the first or the last element. This argument proves (by induction on n) that G has $n - 1$ edges, i.e., (a) implies (d).

Assume that G satisfies (d) and let V_n, V_{n-1}, G' be as in the preceding paragraph. Then the graph G is connected, since V_n is connected to V_{n-1} which (by induction on n) is connected to all other vertices of G'. Thus (d) implies (e).

Finally assume that G satisfies (e). If G contains a cycle, we can delete any edge appearing in that cycle and G would still be connected. We can therefore continue deleting edges in this way until we obtain a connected graph G' with $n - 1 - k$ edges and no cycles. But since (a) implies (d), we must have $k = 0$, that is, $G = G'$. ∎

The idea of a free tree can be applied directly to the analysis of computer algorithms. In Section 1.3.3, we discussed the application of Kirchhoff's law to the problem of counting the number of times each step of an algorithm is performed; we found that Kirchhoff's law does not completely determine the number of times each step is executed, but it reduces the number of unknowns that must be specially interpreted. The theory of trees tells us how many independent unknowns will remain, and it gives us a systematic way to find them.

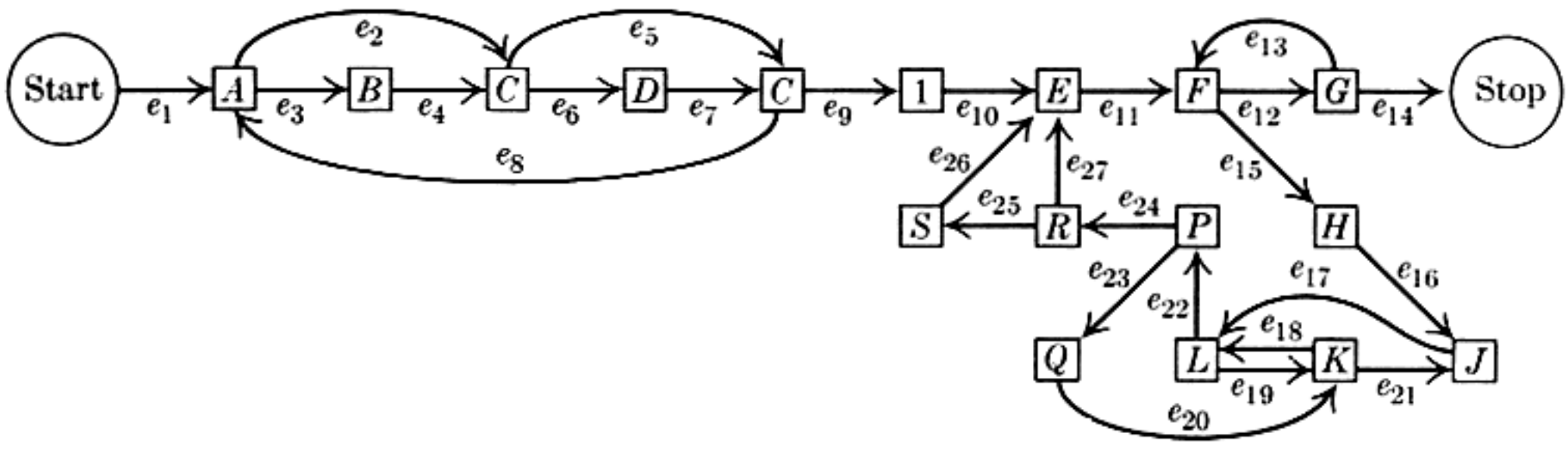


Fig. 31. Abstracted flow chart of Program 1.3.3A.

It is easier to understand the method which follows if an example is studied, so we will work an example as the theory is being developed. Figure 31 shows an abstracted flow chart for Program 1.3.3A, which was subjected to a "Kirchhoff's law" analysis in Section 1.3.3. Each box in Fig. 31 represents part of the computation, and the letter or number inside the box denotes the number of times that computation will be performed during one run of the program, using the notation of Section 1.3.3. An arrow between boxes represents a possible jump in the program. The arrows have been labeled $e_1, e_2, \ldots, e_{27}$. Our goal is to find all relations between the quantities A, B, C, D, E, F, G, H, J, K, L, P, Q, R, and S that are implied by Kirchhoff's law, and at the same time we hope to gain some insight into the general problem. (*Note:* Some simplifications have

already been made in Fig. 31, e.g., the box between C and E has been labeled "1", and this is in fact a consequence of Kirchhoff's law.)

Let E_j denote the number of times branch e_j is taken during the execution of the program being studied; Kirchhoff's law is

$$\text{"sum of } E\text{'s into box} = \text{value in box} = \text{sum of } E\text{'s leaving box"}; \tag{1}$$

e.g., in the case of the box marked K we have

$$E_{19} + E_{20} \quad = \quad K \quad = \quad E_{18} + E_{21}. \tag{2}$$

In the discussion which follows, we will regard $E_1, E_2, \ldots, E_{27}$ as the unknowns, instead of $A, B, \ldots, S$.

The flow chart in Fig. 31 may be further abstracted so that it becomes a graph G as in Fig. 32. The boxes have shrunk to vertices, and the arrows $e_1, e_2, \ldots$ now represent edges of the graph. (A graph, strictly speaking, has no implied direction in its edges, and when we refer to graph-theoretical properties of G, the direction of the arrows should be ignored. The application to Kirchhoff's law, however, makes use of the arrows, as we will see later.) For convenience an extra edge e_0 has been drawn from the "stop" vertex to the "start" vertex, so that Kirchhoff's law applies uniformly to all parts of the graph. Figure 32 also includes some other minor changes from Fig. 31: an extra vertex and edge have been added to divide e_{13} into two parts e'_{13} and e''_{13}, so that the basic definition of a graph (no two edges join the same two vertices) is valid; e_{19} has also been split up in this way. A similar modification would have been made if we had any vertex with an arrow leading back to itself.

Some of the edges in Fig. 32 have been drawn much heavier than the others. These edges form a *free subtree* of the graph, connecting all the vertices. It is

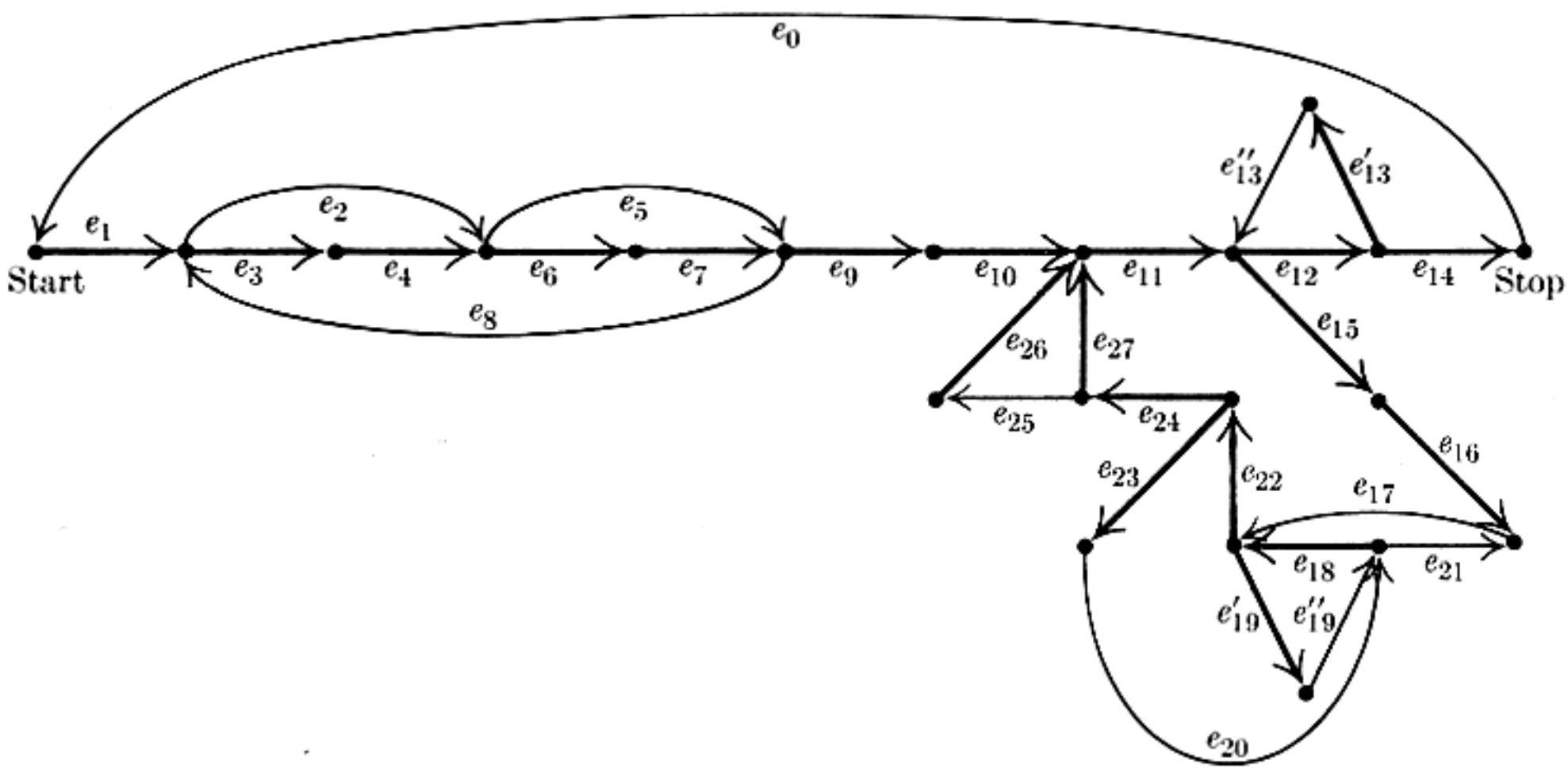


Fig. 32. Graph corresponding to Fig. 31, including a free subtree.

always possible to find a free subtree of the graphs arising from flow charts, because the graphs must be connected and, by part (b) of Theorem A, if G is connected and not a free tree, we can delete some edge and still have the resulting graph connected; this process can be iterated until we reach a subtree. Another algorithm for finding a free subtree appears in exercise 9. We can in fact always discard the edge e_0 (which went from the "stop" to the "start" vertex) first, so that we may assume e_0 does not appear in the subtree chosen.

Let G' be a free subtree of the graph G found in this way, and consider any edge VV' of G that is *not* in G'. We may now note an important consequence of Theorem A: G' plus this new edge VV' contains a cycle, and in fact there is *exactly one* cycle having the form $(V, V', \ldots, V)$, since there is a unique simple path from V' to V in G'. For example, if G' is the free subtree shown in Fig. 32, and if we add the edge e_2, we obtain a cycle which goes along e_2 and then (in the direction opposite to the arrows) along e_4 and e_3. This cycle may be written algebraically as "$e_2 - e_3 - e_4$", using plus signs and minus signs to indicate whether the cycle goes in the direction of the arrows or not.

If we carry out this process for each edge not in the free subtree, we obtain the so-called *fundamental cycles*, which in the case of Fig. 32 are

$$\begin{aligned}
C_0&: \quad e_0 + e_1 + e_3 + e_4 + e_6 + e_7 + e_9 + e_{10} + e_{11} + e_{12} + e_{14},\\
C_2&: \quad e_2 - e_3 - e_4,\\
C_5&: \quad e_5 - e_6 - e_7,\\
C_8&: \quad e_8 + e_3 + e_4 + e_6 + e_7,\\
C''_{13}&: \quad e''_{13} + e_{12} + e'_{13},\\
C_{17}&: \quad e_{17} + e_{22} + e_{24} + e_{27} + e_{11} + e_{15} + e_{16},\\
C''_{19}&: \quad e''_{19} + e_{18} + e'_{19},\\
C_{20}&: \quad e_{20} + e_{18} + e_{22} + e_{23},\\
C_{21}&: \quad e_{21} - e_{16} - e_{15} - e_{11} - e_{27} - e_{24} - e_{22} - e_{18},\\
C_{25}&: \quad e_{25} + e_{26} - e_{27}.
\end{aligned} \tag{3}$$

Obviously an edge e_j which is not in the free subtree will appear in only one of the fundamental cycles, namely C_j.

We are now approaching the climax of this construction. Each fundamental cycle represents a solution to Kirchhoff's equations; for example, the solution corresponding to C_2 is to let $E_2 = +1$, $E_3 = -1$, $E_4 = -1$, and all other E's $= 0$. It is clear that flow around a cycle in a graph always satisfies the condition (1) of Kirchhoff's law. Moreover, Kirchhoff's equations are "homogeneous," so the sum or difference of solutions to (1) yields another solution. Therefore we may conclude that the values of $E_0, E_2, E_5, \ldots, E_{25}$ are *independent* in the following sense:

If $x_0, x_2, \ldots, x_{25}$ are any real numbers (one x_j for each e_j not in the free subtree G'), *there is a solution to Kirchhoff's equations* (1) *such that $E_0 = x_0$, $E_2 = x_2, \ldots, E_{25} = x_{25}$.* (4)

Such a solution is found by going x_0 times around cycle C_0, x_2 times around cycle C_2, etc. Furthermore, we find that the values of the remaining variables $E_1, E_3, E_4, \ldots$ are completely *dependent* on the values $E_0, E_2, \ldots, E_{25}$:

The solution mentioned in statement (4) *is unique.* (5)

For if there are two solutions to Kirchhoff's equations such that $E_0 = x_0, \ldots, E_{25} = x_{25}$, we can subtract one from the other and we thereby obtain a solution in which $E_0 = E_2 = E_5 = \cdots = E_{25} = 0$. But now *all* E_j must be zero, for it is easy to see that a nonzero solution to Kirchhoff's equations is impossible when the graph is a free tree (see exercise 4). Therefore the two assumed solutions must be identical. We have now proved that all solutions of Kirchhoff's equations may be obtained as sums of multiples of the fundamental cycles.

When these remarks are applied to the graph in Fig. 32, we obtain the following general solution of Kirchhoff's equations in terms of the independent variables $E_0, E_2, \ldots, E_{25}$:

$$
\begin{aligned}
E_1 &= E_0, & E_{14} &= E_0,\\
E_3 &= E_0 - E_2 + E_8, & E_{15} &= E_{17} - E_{21},\\
E_4 &= E_0 - E_2 + E_8, & E_{16} &= E_{17} - E_{21},\\
E_6 &= E_0 - E_5 + E_8, & E_{18} &= E''_{19} + E_{20} - E_{21},\\
E_7 &= E_0 - E_5 + E_8, & E'_{19} &= E''_{19},\\
E_9 &= E_0, & E_{22} &= E_{17} + E_{20} - E_{21},\\
E_{10} &= E_0, & E_{23} &= E_{20},\\
E_{11} &= E_0 + E_{17} - E_{21}, & E_{24} &= E_{17} - E_{21},\\
E_{12} &= E_0 + E''_{13}, & E_{26} &= E_{25},\\
E'_{13} &= E''_{13}, & E_{27} &= E_{17} - E_{21} - E_{25}.
\end{aligned}
\tag{6}
$$

To obtain these equations, we merely list, for each edge e_j in the subtree, all E_k for which e_j appears in cycle C_k, with the appropriate sign. [Thus, the matrix of coefficients in (6) is just the transpose of the matrix of coefficients in (3).]

Strictly speaking, C_0 should not be called a fundamental cycle, since it involves the special edge e_0. We may call C_0 minus the edge e_0 a *fundamental path from "start" to "stop."* Our boundary condition, that the "start" and "stop" boxes in the flow chart are performed exactly once, is equivalent to the relation

$$E_0 = 1. \tag{7}$$

The preceding discussion shows how to obtain all solutions to Kirchhoff's law; the same method may be applied (as Kirchhoff himself applied it) to electrical circuits instead of program flow charts. It is natural to ask at this point whether Kirchhoff's law is the strongest possible set of equations that can be given for the case of program flow charts, or whether more can be said: Any execution of a computer program that goes from "start" to "stop" gives us a set of values $E_1, E_2, \ldots, E_{27}$ for the number of times each edge is traversed, and these values obey Kirchoff's law; but are there solutions to Kirchhoff's

equations which do not correspond to any computer program execution? (In this question, we do not assume any information about the computer program as given to us, except its flow chart.) If there are solutions which meet Kirchhoff's conditions but do not correspond to actual program execution, we can give stronger conditions than Kirchhoff's law. For the case of electrical circuits Kirchhoff himself gave a second law: the sum of the voltage drops around a fundamental cycle must be zero. This second law does not apply to our problem.

There is indeed an obvious further condition that the E's must satisfy, if they are to correspond to some actual path in the flow chart from "start" to "stop"; they must be integers, and in fact they must be *nonnegative integers.* This is not a trivial condition, for we may not simply assign arbitrary nonnegative integer values to the independent variables $E_2, E_5, \ldots, E_{25}$; for example, if we take $E_2 = 2$ and $E_8 = 0$, we find from (6), (7) that $E_3 = -1$. (Thus, no execution of the flow chart in Fig. 31 will take branch e_2 twice without taking branch e_8 at least once.) The condition that all the E's be nonnegative integers is not enough either; for example, consider the solution in which $E''_{19} = 1$, $E_2 = E_5 = \cdots = E_{17} = E_{20} = E_{21} = E_{25} = 0$; there is no way to get to e_{18} except via e_{15}. The following condition is a necessary and sufficient condition which answers the problem raised in the previous paragraph: Let $E_2, E_5, \ldots, E_{25}$ be any given values, and determine $E_1, E_3, \ldots, E_{27}$ according to (6), (7). Assume that all the E's are nonnegative integers, and assume that the graph whose edges are those e_j for which $E_j > 0$, and whose vertices are those which touch such e_j, is *connected.* Then there is a path from "start" to "stop" in which edge e_j is traversed exactly E_j times. This fact is proved in the next section (see exercise 2.3.4.2–24).

Let us now summarize the preceding discussion:

Theorem K. *If a flow chart (such as Fig. 31) contains n boxes (including "start" and "stop") and m arrows, it is possible to find $m - n + 1$ fundamental cycles and a fundamental path from "start" to "stop", such that any path from "start" to "stop" is equivalent (in terms of the number of times each edge is traversed) to one traversal of the fundamental path plus a uniquely determined number of traversals of each of these fundamental cycles.* (The fundamental path and fundamental cycles may include some edges which are to be traversed in a direction *opposite* that shown by the arrow on the edge; we conventionally say that such edges are being traversed -1 times.)

Conversely, for any traversal of the fundamental path and the fundamental cycles in which the total number of times each edge is traversed is nonnegative, and in which the vertices and edges corresponding to a positive number of traversals form a connected graph, there is at least one equivalent path from "start" to "stop." ∎

The fundamental cycles are found by picking a free subtree as in Fig. 32; if we choose a different subtree we get, in general, a different set of fundamental cycles. The fact that there are $m - n + 1$ fundamental cycles follows from Theorem A. The modifications we made to get from Fig. 31 to Fig. 32 do not

change the value of $m - n + 1$, although they may increase both m and n; the construction could have been generalized so as to avoid these trivial modifications entirely (see exercise 8).

Theorem K is encouraging because it says that Kirchhoff's law (which consists of n equations in the m unknowns $E_1, E_2, \ldots, E_m$) has just one "redundancy," i.e., these n equations allow us to eliminate $n - 1$ unknowns. Note however that throughout this discussion the unknown variables have been the number of times the *edges* have been traversed, not the number of times each *box* of the flow chart has been entered.

Algorithms for finding all cycles in a graph are discussed by J. T. Welch, Jr., in *JACM* **13** (1966), 205–210.

EXERCISES

1. [*14*] List all cycles from B to B which are present in the graph of Fig. 29.

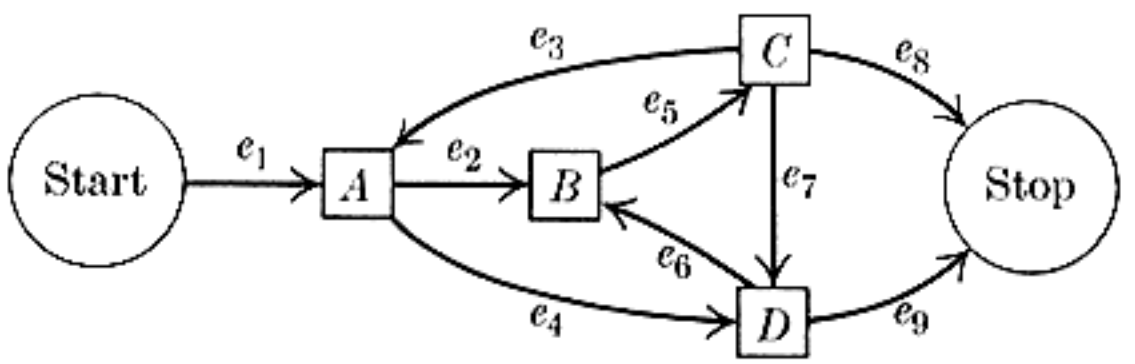
2. [*M20*] Prove that if V and V' are vertices of a graph and if there is a path from V to V', then there is a simple path from V to V'.

3. [*15*] What path from "start" to "stop" is equivalent (in the sense of Theorem K) to one traversal of the fundamental path plus one traversal of cycle C_2 in Fig. 32?

▶ 4. [*M20*] Let G' be a finite free tree in which arrows have been drawn on its edges $e_1, \ldots, e_{n-1}$; let $E_1, \ldots, E_{n-1}$ be numbers satisfying Kirchhoff's law (1) in G'. Show that $E_1 = \cdots = E_{n-1} = 0$.

5. [*20*] Using Eqs. (6), express the quantities $A, B, \ldots, S$ which appear inside the boxes of Fig. 31 in terms of the independent variables $E_2, E_5, \ldots, E_{25}$.

6. [*22*] Carry out the construction in the text for the flow chart



using the free subtree consisting of edges e_1, e_2, e_3, e_4, e_9. What are the fundamental cycles? Express E_1, E_2, E_3, E_4, E_9 in terms of E_5, E_6, E_7, and E_8.

▶ 7. [*M23*] The text shows that if we apply Kirchhoff's laws to the number of times each edge is traversed, $n - 1$ of the unknowns are eliminated (if there are n boxes in the flow chart). Discuss the effect of Kirchhoff's law on the number of independent unknowns if we let the unknowns be the number of times each *box* is entered instead of the number of times each edge is traversed. (Thus, we might take $A, B, \ldots, S$ to be the unknowns in the text's example, instead of $E_1, E_2, \ldots, E_{27}$.)

8. [*M22*] Edges e_{13} and e_{19} were split into two parts in Fig. 32, since a graph is not supposed to have two edges joining the same two vertices. However, if we look at the final result of the construction, this splitting into two parts seems quite artificial

since $E'_{13} = E''_{13}$, $E'_{19} = E''_{19}$ are two of the relations found in (6), and E''_{13}, E''_{19} are two of the independent variables. Explain how the construction could be generalized so that an artificial splitting of edges may be avoided.

▶ **9.** [*M27*] Suppose a graph has n vertices $V_1, \ldots, V_n$ and m edges $e_1, \ldots, e_m$. Each edge e_k is represented by a pair of integers (a_k, b_k) giving the numbers of the vertices which it makes adjacent. Design an algorithm which takes the input pairs $(a_1, b_1), \ldots, (a_m, b_m)$ and prints out a subset of these which forms a free tree; the algorithm reports failure if this is impossible. Try to make the algorithm as efficient as possible.

10. [*16*] An electrical engineer, designing the circuitry for a computer, finds that he has n terminals $T_1, T_2, \ldots, T_n$ which he wants to have at essentially the same voltage at all times. To achieve this, he can solder wires between any pairs of terminals; the idea is to make enough wire connections so that there is a path through the wires from any terminal to any other. Show that the minimum number of wires needed to connect all the terminals is $n - 1$, and $n - 1$ wires achieve the desired connection if and only if they form a free tree (with terminals and wires standing for vertices and edges).

11. [*M27*] (R. C. Prim, *Bell System Tech. J.* **36** (1957), 1389–1401.) Consider the wire connection problem of exercise 10 with the additional proviso that a cost $c(i, j)$ is given for each $i < j$, denoting the expense of wiring terminal T_i to terminal T_j. Show that the following algorithm gives a connection tree of minimum cost: "If $n = 1$, do nothing. Otherwise, renumber the terminals and costs so that

$$c(n-1, n) = \min_{1 \le i < n} c(i, n);$$

connect terminal T_{n-1} to T_n; then change $c(j, n-1)$ to $\min\big(c(j, n-1), c(j, n)\big)$ for $1 \le j < n - 1$, and repeat the algorithm for $n - 1$ terminals $T_1, \ldots, T_{n-1}$ using these new costs. (The algorithm is to be repeated with the understanding that whenever a connection is subsequently requested between the terminals now called T_j and T_{n-1}, the connection is actually made between terminals now called T_j and T_n if it is cheaper; thus T_{n-1} and T_n are being regarded as though they were one terminal in the remainder of the algorithm.)" This algorithm may also be stated as follows: "Choose a particular terminal to start with; then repeatedly make the cheapest possible connection from an unchosen terminal to a chosen one, until all have been chosen."

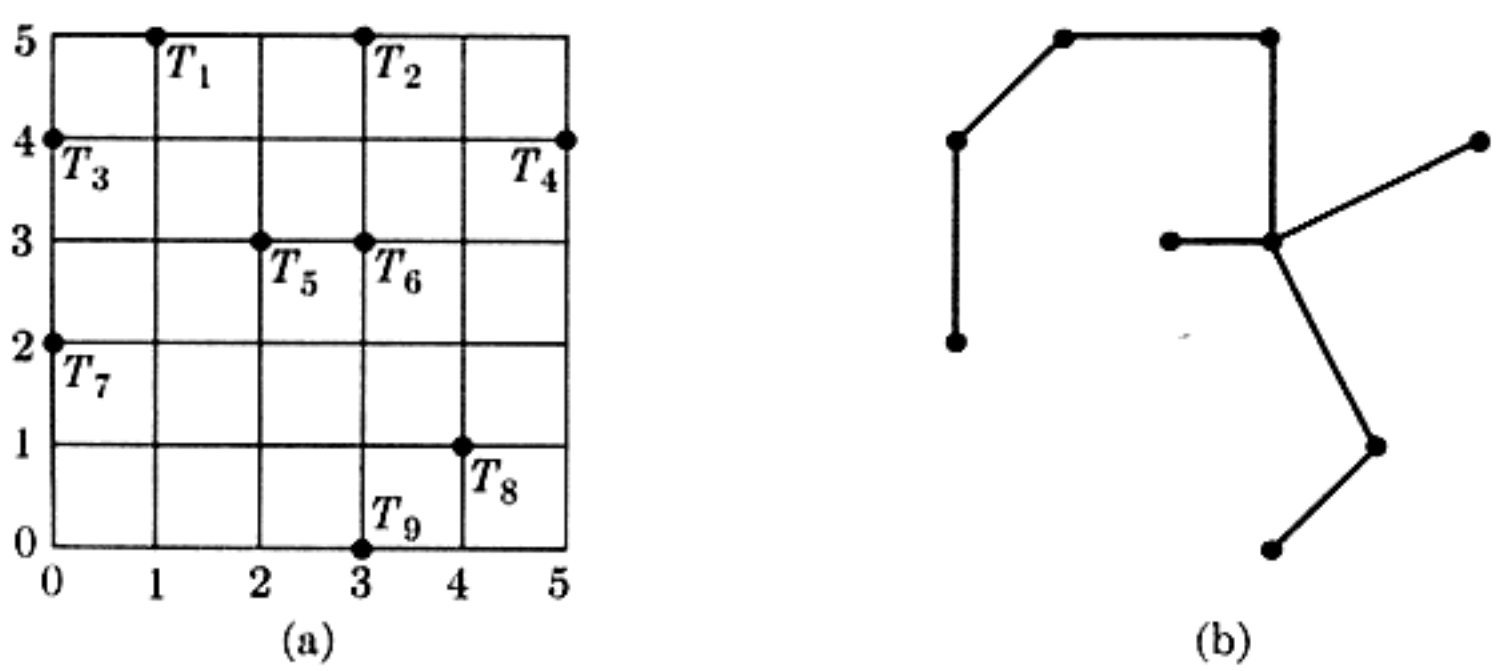


Fig. 33. Free tree of minimum cost. (See exercise 11.)

For example, consider Fig. 33(a), which shows nine terminals on a grid; let the cost of connecting two terminals be the wire length, i.e., the distance between them. (The reader may wish to try to find a minimal cost tree by hand, using intuition instead of the above algorithm.) The algorithm would first connect T_8 to T_9, then T_6 to T_8, T_5 to T_6, T_2 to T_6, T_1 to T_2, T_3 to T_1, T_7 to T_3, and finally T_4 to either T_2 or T_6. A minimum cost tree (with total wire length $7 + 2\sqrt{2} + 2\sqrt{3}$) is shown in Fig. 33(b).

▶ **12.** [*29*] The algorithm of exercise 11 is not stated in a fashion suitable for direct computer implementation. Reformulate that algorithm, specifying in more detail the operations that are to be done, in such a way that a computer program can carry out the process with reasonable efficiency.

13. [*M20*] Let G be a graph (possibly infinite) which contains no cycles. Assume that if any edge not already present in G is added to G, a cycle is formed. Prove that G is a free tree.

14. [*M24*] Consider a graph with n vertices and m edges, in the notation of exercise 9. Show that it is possible to write any permutation of the integers $\{1, 2, \ldots, n\}$ as a product of transpositions $(a_{k_1}b_{k_1})(a_{k_2}b_{k_2}) \cdots (a_{k_t}b_{k_t})$ if and only if the graph is connected. (Hence there are sets of $n - 1$ transpositions which generate all permutations on n elements, but no set of $n - 2$ will do so.)

2.3.4.2. Oriented trees. In the previous section, we saw that an abstracted flow chart may be regarded as a graph, if we ignore the direction of the arrows on its edges; the graph-theoretic ideas of "cycle" and "free subtree," etc., were shown to be relevant in the study of flow charts. There is a good deal more that can be said when the direction of each edge is given more significance, and in this case we have what is called a "directed graph" or "digraph."

Let us define a *directed graph* formally as a set of vertices and a set of *arcs*, each arc leading from a vertex V to a vertex V'. If e is an arc from V to V' we say V is the *initial* vertex of e, and V' is the *final* vertex, and we write $V = \text{init}(e)$, $V' = \text{fin}(e)$. The case that $\text{init}(e) = \text{fin}(e)$ is not excluded (although it was excluded from the definition of edge in an ordinary graph), and several different arcs may have the same initial and final vertices. The *out-degree* of a vertex V is the number of arcs leading out from it, i.e., the number of arcs e such that $\text{init}(e) = V$; similarly, the *in-degree* of V is the number of arcs with $\text{fin}(e) = V$.

The concepts of paths, cycles, etc. are defined for directed graphs in a manner similar to the corresponding definitions for ordinary graphs, but there are some important new technicalities that must be considered. If $e_1, e_2, \ldots, e_n$ are arcs (with $n \geq 1$), we say $(e_1, e_2, \ldots, e_n)$ is an *oriented path* of length n from V to V' if $V = \text{init}(e_1)$, $V' = \text{fin}(e_n)$, and $\text{fin}(e_k) = \text{init}(e_{k+1})$ for $1 \leq k < n$. An oriented path $(e_1, e_2, \ldots, e_n)$ is called *simple* if $\text{init}(e_1), \ldots, \text{init}(e_n)$ are distinct and $\text{fin}(e_1), \ldots, \text{fin}(e_n)$ are distinct. An *oriented cycle* is a simple oriented path from a vertex to itself. (Note that an oriented cycle can have length 1 or 2, but this was excluded from our definition of "cycle" in the previous section. Can the reader see why this makes sense?)

As examples of these straightforward definitions, we may refer to Fig. 31 in the previous section. The box labeled "J" is a vertex with in-degree 2 (because of the arcs e_{16}, e_{21}) and out-degree 1. The sequence $(e_{17}, e_{19}, e_{18}, e_{22})$ is an oriented path of length 4 from J to P; this path is not simple since, for example, init $(e_{19}) = L =$ init (e_{22}). The diagram contains no oriented cycles of length 1, but (e_{18}, e_{19}) is an oriented cycle of length 2.

A directed graph is said to be *strongly connected* if there is an oriented path from V to V' for any two vertices $V \neq V'$. It is said to be *rooted* if there is at least one "root," i.e., at least one vertex R such that there is an oriented path from V to R for all $V \neq R$. ["Strongly connected" always implies "rooted," but the converse does not hold. A flow chart such as Fig. 31 in the previous section is an example of a rooted digraph, with R the "stop" vertex; with the additional arc from "stop" to "start" (Fig. 32) it becomes strongly connected.]

Every directed graph G corresponds in an obvious manner to an ordinary graph G_0, where G_0 has an edge from V to V' if and only if $V \neq V'$ and G has an arc from V to V' or from V' to V. We can speak of (unoriented) *paths* and *cycles* in G with the understanding that these are paths and cycles of G_0; we can say that G is *connected* (this is a much weaker property than "strongly connected," even weaker than "rooted") if the corresponding graph G_0 is connected, etc.

An *oriented tree* (see Fig. 34), sometimes called a "rooted tree" by other authors, is a directed graph with a specified vertex R such that:

a) Each vertex $V \neq R$ is the initial vertex of exactly one arc, denoted by $e[V]$;

b) R is the initial vertex of no arc;

c) R is a root in the sense defined above (i.e., for each vertex $V \neq R$ there is an oriented path from V to R).

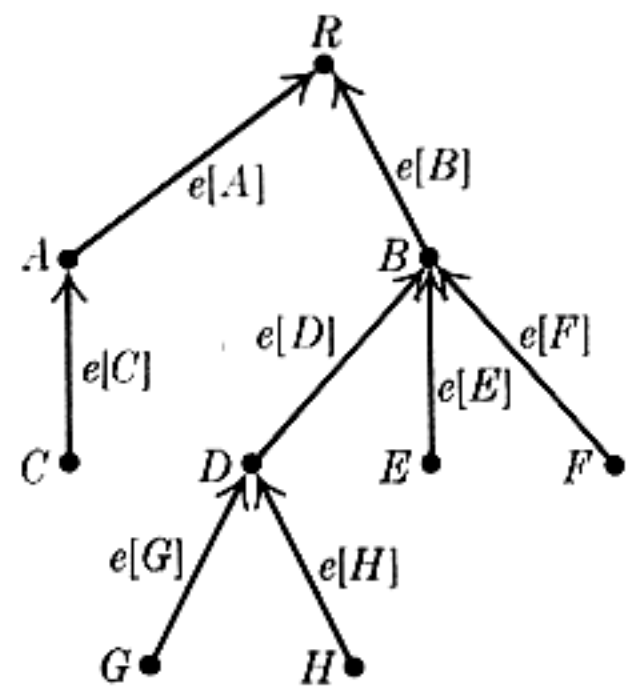


Fig. 34. An oriented tree.

It follows immediately that for each vertex $V \neq R$ there is a *unique* oriented path from V to R; and hence there are no oriented cycles.

Our previous definition of "oriented tree" (at the beginning of Section 2.3) is easily seen to be compatible with the new definition just given, when there are finitely many vertices; the vertices correspond to nodes, and the arc $e[V]$ is the link from V to FATHER(V).

The (undirected) graph corresponding to an oriented tree is connected [because of property (c)], and contains no cycles [since by property (a) if $(V_0, V_1, \ldots, V_n)$ is a cycle with $n \geq 3$, either

$$(e[V_0], e[V_1], \ldots, e[V_{n-1}]) \qquad \text{or} \qquad (e[V_n], \ldots, e[V_2], e[V_1])$$

is an oriented cycle]; therefore *an oriented tree is a free tree* when the direction of the arcs is neglected.

Conversely, it is important to note that we can reverse the process just described. If we start with any nonempty free tree, such as that in Fig. 30, we can choose *any* vertex as the root R, and assign directions to the edges. The intuitive idea is to "pick up" the graph at vertex R and shake it; then assign upward-pointing arrows. More formally, the rule is this:

> Change the edge VV' to an arc from V to V' if and only if the simple path from V to R leads through V', that is, if it has the form $(V_0, V_1, \ldots, V_n)$, where $n > 0$, $V_0 = V$, $V_1 = V'$, $V_n = R$.

To verify that such a construction is valid, we need to prove that each edge VV' is assigned the direction $V \rightarrow V'$ or the direction $V' \leftarrow V$; and this is easy to prove, for if $(V, V_1, \ldots, R)$ and $(V', V_1', \ldots, R)$ are paths, there is a cycle unless $V = V_1'$ or $V_1 = V'$. It is a consequence of this construction that the directions of the arcs in an oriented tree are completely determined by knowing which vertex is the root, so they need not be shown in diagrams when the root is explicitly indicated.

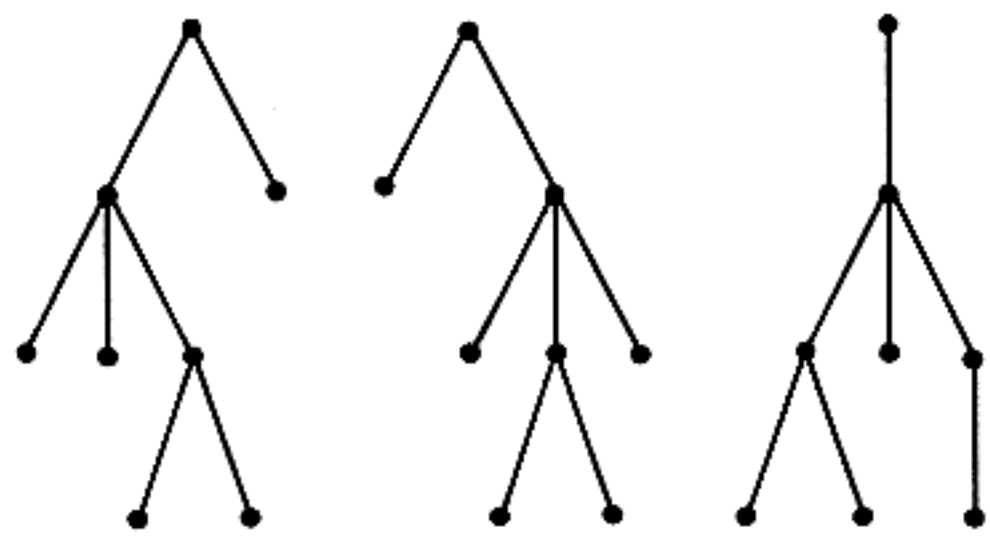


Fig. 35. Three tree structures.

We now see the relation between three types of trees: the (ordered) tree which is of principal importance in computer programs, as defined at the beginning of Section 2.3; the oriented tree (i.e., unordered tree); and the free tree. Both of the latter two types of trees arise in the study of computer algorithms, but not as often as the first type. *The essential distinction between these types of tree structure is merely the amount of information that is taken to be relevant.* For example, Fig. 35 shows three distinct trees if they are considered as ordered trees (with root at the top). As oriented trees, the first and second are identical, since the left-to-right order of subtrees is immaterial; as free trees, all three graphs in Fig. 35 are identical, since the root is immaterial.

An *Eulerian circuit* in a directed graph is an oriented path $(e_1, e_2, \ldots, e_m)$ such that *every* arc in the directed graph occurs exactly once, and fin $(e_m) =$ init (e_1). This is a "complete traversal" of the arcs of the directed graph. (Eulerian circuits get their name from Leonhard Euler's famous discussion in 1736 of the impossibility of traversing each of the seven bridges in the city of Königsberg exactly once during a Sunday stroll. He treated the analogous problem for

undirected graphs. Eulerian circuits should be distinguished from "Hamiltonian circuits," which are oriented cycles that encounter each *vertex* exactly once; see Chapter 7.)

A directed graph is said to be *balanced* (see Fig. 36) if every vertex V has the same in-degree as its out-degree, i.e., there are just as many edges with V as their initial vertex as there are with V as their final vertex. This condition is closely related to Kirchhoff's law (see exercise 24). It is obviously possible to find an Eulerian circuit in a directed graph only if the graph is connected and balanced, provided that there are no *isolated vertices*, i.e., vertices with in-degree and out-degree both equal to zero.

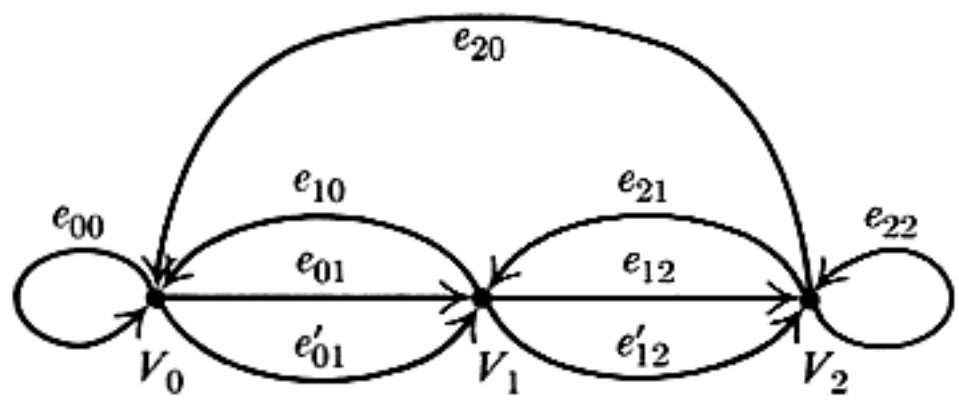


Fig. 36. A balanced directed graph.

So far in this section there have been quite a few definitions (e.g., directed graph, arc, initial vertex, final vertex, out-degree, in-degree, oriented path, simple oriented path, oriented cycle, oriented tree, Eulerian circuit, isolated vertex, and the properties of being strongly connected, rooted, and balanced), but there has been a scarcity of important results connecting these concepts. Now we are ready for meatier material. The first basic result is a theorem due to I. J. Good [*J. London Math. Soc.* **21** (1947), 167–169], who showed that Eulerian circuits are always possible unless they are obviously impossible:

Theorem G. *A finite, directed graph with no isolated vertices possesses an Eulerian circuit if and only if it is connected and balanced.*

Proof. Assume G is balanced and let

$$P = (e_1, \ldots, e_m)$$

be an oriented path of longest possible length that uses no arc twice. Then if $V = \text{fin}\,(e_m)$, and if k is the out-degree of V, all k arcs e with $\text{init}\,(e) = V$ must already appear in P, otherwise we could add e and get a longer path. But if $\text{init}\,(e_j) = V$ and $j > 1$, then $\text{fin}\,(e_{j-1}) = V$; hence, since G is balanced, we must have

$$\text{init}\,(e_1) = V = \text{fin}\,(e_m),$$

otherwise the in-degree of V would be at least $k + 1$.

Now by cyclic permutation of P it follows that any arc e not in the path has neither initial nor final vertex in common with any arc in the path; so if P is not an Eulerian circuit, G is not connected. ∎

There is an important connection between Eulerian circuits and oriented trees:

Lemma E. *Let* $(e_1, \ldots, e_m)$ *be an Eulerian circuit of a directed graph* G *having no isolated vertices. Let* $R = \text{fin}\,(e_m) = \text{init}\,(e_1)$. *For each vertex* $V \neq R$ *let* $e[V]$ *be the "last exit" from* V *in the circuit, i.e.,*

$$e[V] = e_j \quad \textit{if} \quad \text{init}\,(e_j) = V \quad \textit{and} \quad \text{init}\,(e_k) \neq V \quad \textit{for} \quad j < k \leq m. \tag{1}$$

Then the vertices of G *with the arcs* $e[V]$ *form an oriented tree with root* R.

Proof. Properties (a) and (b) of the definition of oriented tree are evidently satisfied. By exercise 7 we need only show there are no oriented cycles among the $e[V]$; but this is immediate, since if $\text{fin}\,(e[V]) = V' = \text{init}\,(e[V'])$, where $e[V] = e_j$ and $e[V'] = e_{j'}$, then $j < j'$. ∎

This lemma can perhaps be better understood if we turn things around and consider the "first entrances" to each vertex; the first entrances form an unordered tree with all arcs pointing *away* from R. Lemma E has a surprising and important converse, proved by T. van Aardenne-Ehrenfest and N. G. de Bruijn [*Simon Stevin* **28** (1951), 203–217]:

Theorem D. *Let* G *be a finite, balanced, directed graph, and let* G' *be an oriented tree consisting of the vertices of* G *plus some of the arcs of* G. *Let* R *be the root of* G' *and let* $e[V]$ *be the arc of* G' *with initial vertex* V. *Let* e_1 *be any arc of* G *with* $\text{init}\,(e_1) = R$. *Then* $P = (e_1, e_2, \ldots, e_m)$ *is an Eulerian circuit if it is an oriented path for which*

i) *no arc is used more than once; i.e.,* $e_j \neq e_k$ *when* $j \neq k$.
ii) $e[V]$ *is not used in* P *unless it is the only choice consistent with rule* (i)*; i.e., if* $e_j = e[V]$ *and if* e *is an arc with* $\text{init}\,(e) = V$, *then* $e = e_k$ *for some* $k \leq j$.
iii) P *terminates only when it cannot be continued by rule* (i)*; i.e., if* $\text{init}\,(e) = \text{fin}\,(e_m)$, *then* $e = e_k$ *for some* k.

Proof. By (iii) and the argument in the proof of Theorem G, we must have $\text{fin}\,(e_m) = \text{init}\,(e_1) = R$. Now if e is an arc not appearing in P, let $V = \text{fin}\,(e)$. Since G is balanced, it follows that V is the initial vertex of some arc not in P; and if $V \neq R$, $e[V]$ must not be in P by condition (ii). Now use the same argument with $e = e[V]$, and we ultimately find R is the initial vertex of some arc not in the path, contradicting (iii). ∎

The essence of Theorem D is that it shows us a simple way to construct an Eulerian circuit in a balanced directed graph, given any oriented subtree of the graph. (See the example in exercise 14.) In fact, Theorem D allows us to count the exact number of Eulerian circuits in a directed graph; this result and many other important consequences of the ideas developed in this section appear in the exercises which follow.

EXERCISES

1. [*M20*] Prove that if V and V' are vertices of a directed graph and if there is an oriented path from V to V', then there is a simple oriented path from V to V'.

2. [*15*] Which of the ten "fundamental cycles" listed in (3) of Section 2.3.4.1 are *oriented* cycles in the directed graph (Fig. 32) of that section?

3. [*16*] Draw the diagram for a directed graph that is connected but not rooted.

▶ **4.** [*M20*] The concept of *topological sorting* can be defined for any finite directed graph G as a linear arrangement of the vertices such that init (e) precedes fin (e) in the ordering for all edges e of G. (Cf. Section 2.2.3, Figs. 6 and 7.) Not all finite directed graphs can be topologically sorted; which ones can be? (Use the terminology of this section to give the answer.)

5. [*M21*] Let G be a directed graph which contains an oriented path $(e_1, \ldots, e_n)$ with fin (e_n) = init (e_1). Give a proof that G is not an oriented tree, using the terminology defined in this section.

6. [*M21*] True or false: A directed graph which is rooted and contains no cycles and no oriented cycles is an oriented tree.

▶ **7.** [*M22*] True or false: A directed graph satisfying properties (a) and (b) of the definition of oriented tree, and having no oriented cycles, is an oriented tree.

8. [*HM40*] Study the properties of *automorphism groups* of oriented trees, i.e., the groups consisting of those permutations π of the vertices and arcs such that init $(e\pi)$ = init $(e)\pi$, fin $(e\pi)$ = fin $(e)\pi$.

9. [*18*] By assigning directions to the edges, draw the oriented tree corresponding to the free tree in Fig. 29 of Section 2.3.4.1, with G as the root.

10. [*22*] An oriented tree with vertices $V_1, \ldots, V_n$ can be represented inside a computer by using a table $F[1], \ldots, F[n]$ as follows: If V_j is the root, $F[j] = 0$; otherwise $F[j] = k$, if the arc $e[V_j]$ goes from V_j to V_k. (Thus $F[1], \ldots, F[n]$ is the same as the "father" table used in Algorithm 2.3.3E.)

The text shows how a free tree can be converted into an oriented tree by choosing any desired vertex to be the root. Consequently, it is possible to start with an oriented tree that has root R, then to convert this into a free tree by neglecting the orientation of the arcs, and finally to assign new orientations, obtaining an oriented tree with any specified vertex as the root. Design an algorithm which performs this transformation: Starting with a table $F[1], \ldots, F[n]$, representing an oriented tree, and given an integer j, $1 \le j \le n$, design the algorithm to transform the F table so that it represents the same free tree but with V_j as the root.

▶ **11.** [*28*] Using the assumptions of exercise 2.3.4.1–9, but with (a_k, b_k) representing an edge whose arrow points from V_{a_k} to V_{b_k}, design an algorithm which not only prints out a free subtree as in that algorithm, but also prints out the fundamental cycles. [*Hint:* The algorithm given in the solution to exercise 2.3.4.1–9 can be combined with the algorithm in the preceding exercise.]

12. [*M10*] In the correspondence between oriented trees as defined here and oriented trees as defined at the beginning of Section 2.3, is the *degree* of a tree node equal to the *in-degree* or the *out-degree* of the corresponding vertex?

▶ **13.** [*M24*] Prove that if R is a root of a (possibly infinite) directed graph G, then G contains an oriented subtree with the same vertices as G and with root R. (As a consequence, it is always possible to choose the free subtree in flow charts like Fig. 32 of Section 2.3.4.1 so that it is actually an *oriented* subtree; this would be the case in that diagram if we had selected e''_{13}, e''_{19}, e_{20}, and e_{17} instead of e'_{13}, e'_{19}, e_{23}, and e_{15}.)

14. [*21*] Let G be the directed graph shown in Fig. 36, and let G' be the oriented subtree with vertices V_0, V_1, V_2 and arcs e_{01}, e_{21}. Find all paths P that meet the conditions of Theorem D, starting with arc e_{12}.

15. [*M20*] Prove that a directed graph which is connected and balanced is strongly connected.

▶ **16.** [*M24*] In a popular solitaire game called "clock," the 52 cards of an ordinary deck of playing cards are dealt face down into 13 piles of four each; 12 piles are arranged in a circle like the 12 hours of a clock and the thirteenth pile goes in the center. The solitaire game now proceeds by turning up the top card of the center pile, and then if its face value is k, we place it next to the kth pile. (1, 2, . . . , 13 are equivalent to A, 2, . . . , 10, J, Q, K.) Play continues by turning up the top card of the kth pile and putting it next to *its* pile, etc., until we reach a point where it is impossible to continue since there are no more cards to turn up on the designated pile. (The player has no choice in the game, since the above rules completely specify his actions.) The game is won if all cards are face up when play terminates. [*Reference:* A. Moyse, Jr., *150 ways to play solitaire* (Chicago: Whitman, 1950).]

Show that the game will be won if and only if the following directed graph is an oriented tree: The vertices are $V_1, V_2, \ldots, V_{13}$; the arcs are $e_1, e_2, \ldots, e_{12}$, where e_j goes from V_j to V_k if k is the *bottom* card in pile j after the deal.

(In particular, if the bottom card of pile j is a "j", for $j \neq 13$, it is easy to see that the game is certainly lost, since this card could never be turned up. The result proved in this exercise gives a much faster way to play the game!)

17. [*M32*] What is the probability of winning the solitaire game of clock (described in exercise 16), assuming the deck is randomly shuffled? What is the probability that exactly k cards are still face down when the game is over?

18. [*M30*] (Okada and Onodera, *Bull. Yamagato Univ.* **2** (1952), 89–117.) Let G be a graph with $n+1$ vertices $V_0, V_1, \ldots, V_n$ and m edges $e_1, \ldots, e_m$. Make G into a directed graph by assigning an arbitrary orientation to each edge; then construct the $m \times (n+1)$ matrix A with

$$a_{ij} = \begin{cases} +1, & \text{if init } (e_i) = V_j; \\ -1, & \text{if fin } (e_i) = V_j; \\ 0, & \text{otherwise.} \end{cases}$$

Let A_0 be the $m \times n$ matrix A with column 0 deleted.

a) If $m = n$, show that the determinant of A_0 is equal to 0 if G is not a free tree, and equal to ± 1 if G *is* a free tree.

b) Show that for general m the determinant of $A_0^T A_0$ is the number of free subtrees of G (i.e., the number of ways to choose n of the m edges so that the resulting graph is a free tree). [*Hint:* Use (a) and the result of exercise 1.2.3–46.]

19. [*M31*] (W. T. Tutte, *Proc. Cambridge Phil. Soc.* **44** (1948), 463–482.) Let G be a directed graph with vertices $V_0, V_1, \ldots, V_n$. Let A be the $(n+1)\times(n+1)$ matrix with

$$a_{ij} = \begin{cases} -k, & \text{if } i \neq j \text{ and there are } k \text{ arcs from } V_i \text{ to } V_j; \\ t, & \text{if } i = j \text{ and there are } t \text{ arcs from } V_j \text{ to other vertices.} \end{cases}$$

(It follows that $a_{i0} + a_{i1} + \cdots + a_{in} = 0$ for $0 \le i \le n$.) Let A_0 be the same matrix with row 0 and column 0 deleted. For example, if G is the directed graph of Fig. 36, we have

$$A = \begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & -2 \\ -1 & -1 & 2 \end{pmatrix}, \qquad A_0 = \begin{pmatrix} 3 & -2 \\ -1 & 2 \end{pmatrix}.$$

a) Show that in the special case $a_{00} = 0$ and $a_{jj} = 1$ for $1 \le j \le n$, and if G contains no arcs from a vertex to itself, then G is an oriented tree with root V_0 if and only if $\det A_0 = 1$; and if G is not an oriented tree, then $\det A_0 = 0$.

b) Show that in the general case, $\det A_0$ is the number of oriented subtrees of G with root V_0 (i.e., the number of ways to select n of the arcs of G so that the resulting directed graph is an oriented tree, with V_0 as the root). [*Hint:* Use induction on the number of arcs.]

20. [*M21*] If G is a graph on $n+1$ vertices $V_0, \ldots, V_n$, let B be the $n \times n$ matrix defined as follows for $1 \le i, j \le n$:

$$b_{ij} = \begin{cases} t, & \text{if } i = j \text{ and there are } t \text{ edges touching } V_j; \\ -1, & \text{if } i \neq j \text{ and } V_i \text{ is adjacent to } V_j; \\ 0, & \text{otherwise.} \end{cases}$$

For example, if G is the graph of Fig. 29 in Section 2.3.4.1, with $(V_0, V_1, V_2, V_3, V_4) = (A, B, C, D, E)$, we find that

$$B = \begin{pmatrix} 3 & 0 & -1 & -1 \\ 0 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}.$$

Show that the number of free subtrees of G is $\det B$. [*Hint:* Use exercise 18 or 19.]

21. [*HM38*] Fig. 36 is an example of a directed graph that is not only balanced, it is *regular*, which means every vertex has the same in-degree and out-degree as every other vertex. Let G be a regular directed graph with $n+1$ vertices $V_0, V_1, \ldots, V_n$, in which every vertex has in-degree and out-degree equal to m. (Hence there are $(n+1)m$ arcs in all.) Let G^* be the graph with $(n+1)m$ vertices corresponding to the arcs of G; let a vertex of G^* corresponding to an arc from V_j to V_k in G be denoted by V_{jk}. An arc goes from V_{jk} to $V_{j'k'}$ in G^* if and only if $k = j'$. For example, if G is the directed graph of Fig. 36, G^* is as shown in Fig. 37. An Eulerian circuit in G is a Hamiltonian circuit in G^* and conversely.

Prove that the number of oriented subtrees of G^* is $m^{(n+1)(m-1)}$ times the number of oriented subtrees of G. [*Hint:* Use exercise 19.]

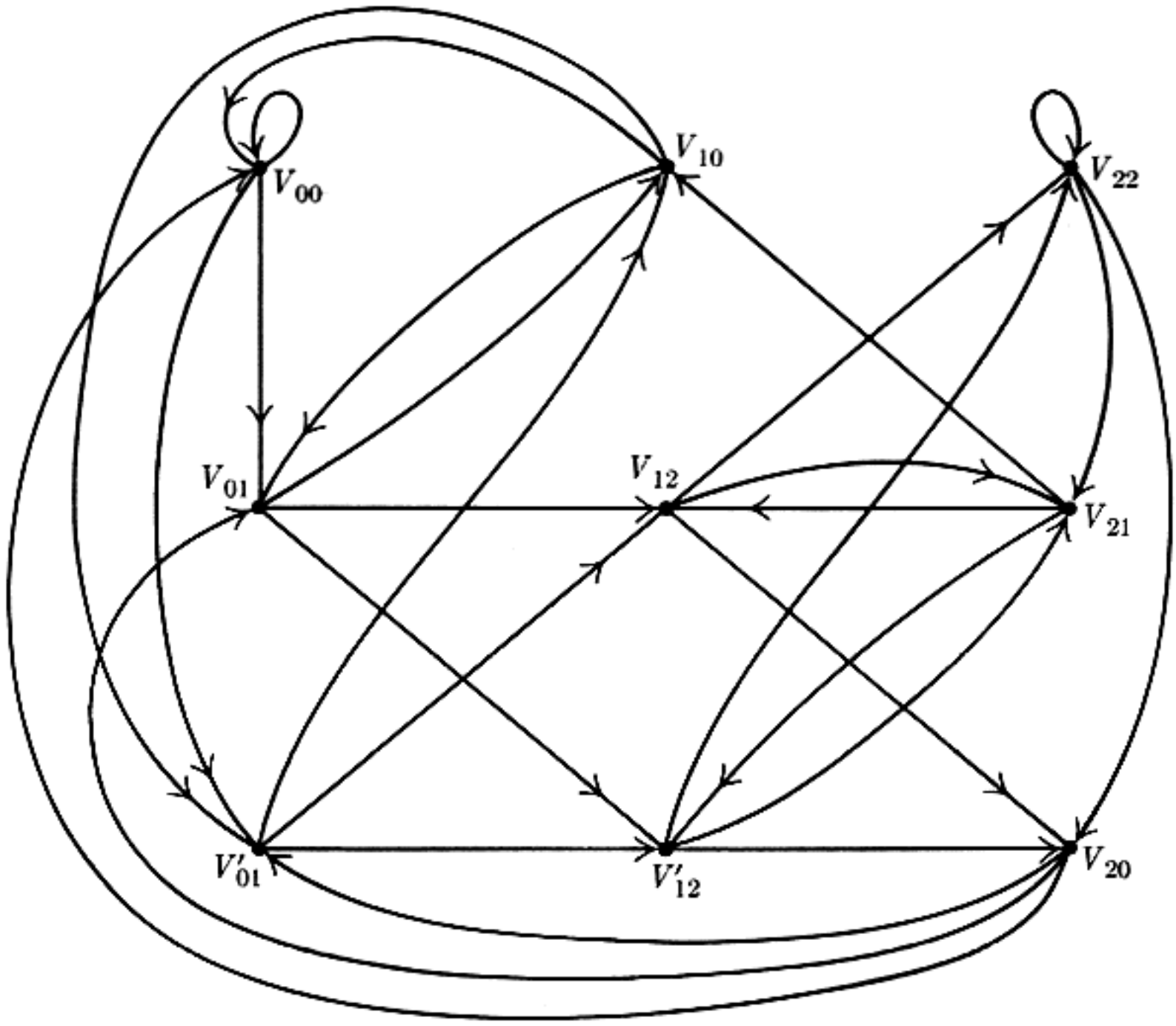


Fig. 37. Edge-graph corresponding to Fig. 36.

▸ **22.** [*M26*] Let G be a balanced, directed graph with vertices $V_1, V_2, \ldots, V_n$ and no isolated vertices. Let σ_j be the out-degree of V_j. Show that the number of Eulerian circuits of G is

$$(\sigma_1 + \sigma_2 + \cdots + \sigma_n)\, T \prod_{1 \le j \le n} (\sigma_j - 1)!,$$

where T is the number of oriented subtrees of G with root V_1. [*Note:* The factor $(\sigma_1 + \cdots + \sigma_n)$, which is the number of arcs of G, may be omitted if the Eulerian circuit $(e_1, \ldots, e_m)$ is regarded as equal to $(e_k, \ldots, e_m, e_1, \ldots, e_{k-1})$.]

▸ **23.** [*M33*] (N. G. de Bruijn.) For each set of nonnegative integers $x_1, \ldots, x_k$ less than m, let $f(x_1, \ldots, x_k)$ be a nonnegative integer less than m. Define an infinite sequence as follows: $X_1 = X_2 = \cdots = X_k = 0$; $X_{n+k+1} = f(X_{n+k}, \ldots, X_{n+1})$ when $n \ge 0$. For how many of the m^{m^k} possible functions f is this sequence periodic with a period of the maximum length m^k? [*Hint:* Construct a directed graph with vertices $(x_1, \ldots, x_{k-1})$ for all $0 \le x_j < m$, and with arcs from $(x_1, x_2, \ldots, x_{k-1})$ to $(x_2, \ldots, x_{k-1}, x_k)$; apply exercises 21 and 22.]

▸ **24.** [*M20*] Let G be a connected, directed graph with arcs $e_0, e_1, \ldots, e_m$. Let $E_0, E_1, \ldots, E_m$ be a set of positive integers which satisfy Kirchhoff's law for G, i.e., for each vertex V,

$$\sum_{\mathrm{init}(e_j)=V} E_j = \sum_{\mathrm{fin}(e_j)=V} E_j.$$

Assume further that $E_0 = 1$. Prove that there is an oriented path in G from fin (e_0) to init (e_0) such that edge e_0 does not appear in the path, and for $1 \le j \le m$ edge e_j appears exactly E_j times. [*Hint:* Apply Theorem G to a suitable directed graph.]

▶ **25.** [*26*] Design a computer representation for directed graphs which generalizes the right-threaded binary tree representation of a tree. Use two link fields ALINK, BLINK and two one-bit fields ATAG, BTAG; and design the representation so that: (a) there is one node for each *arc* of the directed graph (*not* for each vertex); (b) if the directed graph is an oriented tree with root R, and if we add an arc from R to a new vertex H, then the representation of this directed graph is essentially the same as a right-threaded representation of this oriented tree (with some order imposed on the sons in each family), such that ALINK, BLINK, BTAG are respectively the same as LLINK, RLINK, RTAG in Section 2.3.2; and (c) the representation is symmetric in the sense that interchanging ALINK, ATAG with BLINK, BTAG is equivalent to changing the direction on all the arcs of the directed graph.

▶ **26.** [*HM39*] ("Analysis of a random algorithm.") Let G be a directed graph on the vertices $V_1, V_2, \ldots, V_n$. Assume that G represents the flow chart for an algorithm, where V_1 is the "start" vertex and V_n is the "stop" vertex. (Therefore V_n is a root of G.) Suppose each arc e of G has been assigned a probability $p(e)$, where the probabilities satisfy the conditions

$$0 < p(e) \le 1; \qquad \sum_{\text{init}(e)=V_j} p(e) = 1, \qquad 1 \le j < n.$$

Consider a "random path," which starts at V_1 and which subsequently chooses branch e of G with probability $p(e)$, until V_n is reached; the choice of branch taken at each step is to be independent of all previous choices.

For example, consider the graph of exercise 2.3.4.1–6, and assign the respective probabilities $1, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, \frac{3}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}$ to arcs $e_1, e_2, \ldots, e_9$. Then the path "Start–A–B–C–A–D–B–C–Stop" is chosen with probability $1 \cdot \frac{1}{2} \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{3}{4} \cdot 1 \cdot \frac{1}{4} = \frac{3}{128}$.

Such random paths are called *Markov chains*, after the Russian mathematician Andrei A. Markov who first made extensive studies of stochastic processes of this kind. The situation serves as a model for certain algorithms, although our requirement that each choice of path must be independent of the others is a very strong assumption. The problem we wish to solve here is to analyze the computation time for algorithms of this kind.

The analysis is facilitated by considering the $n \times n$ matrix $A = (a_{ij})$, where $a_{ij} = \sum p(e)$ summed over all arcs e which go from V_i to V_j. If there is no such arc, $a_{ij} = 0$. The matrix A for the example considered above is

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & \frac{3}{4} & 0 & 0 & \frac{1}{4} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

It follows easily that $(A^k)_{ij}$ is the probability that a path starting at V_i will be at V_j after k steps.

Prove the following facts, for an arbitrary directed graph G of the above type: (a) The matrix $(I - A)$ is nonsingular. [Hint: Show there is no nonzero vector x with $xA^n = x$.] (b) The average number of times vertex V_j appears in the path is

$$(I - A)^{-1}_{1j} = \text{cofactor}_{j1}(I - A)/\det(I - A), \qquad \text{for} \qquad 1 \leq j \leq n.$$

[Thus in the example considered we find that the vertices A, B, C, D are traversed respectively $\frac{13}{6}$, $\frac{7}{3}$, $\frac{7}{3}$, $\frac{5}{3}$ times, on the average.] (c) The probability that V_j occurs in the path is

$$a_j = \text{cofactor}_{j1}(I - A)/\text{cofactor}_{jj}(I - A);$$

furthermore, $a_n = 1$, so the path terminates in a finite number of steps with probability one. (d) The probability that a random path starting at V_j will never return to V_j is

$$b_j = \det(I - A)/\text{cofactor}_{jj}(I - A).$$

(e) The probability that V_j occurs exactly k times in the path is

$$a_j(1 - b_j)^{k-1}b_j, \qquad \text{for} \qquad k \geq 1, \qquad 1 \leq j \leq n.$$

2.3.4.3. The "infinity lemma." Until now we have concentrated mainly on finite trees, i.e., trees with only finitely many vertices (nodes), but the definitions we have given for free trees and oriented trees apply to infinite graphs as well. Infinite *ordered* trees may be defined in several ways, for example, by extending the concepts of "Dewey decimal notation" to infinite collections of numbers, as in exercise 2.3–14. Even in the study of computer algorithms there is occasionally a need to know the properties of infinite trees (for example, in order to prove by contradiction that a certain tree is *not* infinite). One of the most fundamental properties of infinite trees, first stated in its full generality by D. König, is the following:

Theorem K. (*The "infinity lemma."*) *In any infinite oriented tree for which every vertex has finite degree, there is an "infinite path from the root," i.e., an infinite sequence of vertices* $V_0, V_1, V_2, \ldots$ *in which* V_0 *is the root and* $\text{fin}\,(e[V_{j+1}]) = V_j$ *for all* $j \geq 0$.

Proof. We define the path by starting with V_0, the root of the oriented tree. Assume that $j \geq 0$ and that V_j has been chosen having infinitely many descendants. The degree of V_j is finite by hypothesis, so V_j has finitely many sons $U_1, \ldots, U_n$. At least one of these sons must possess infinitely many descendants, so we take V_{j+1} to be such a son of V_j.

Now $V_0, V_1, V_2, \ldots$ is an infinite path from the root. ∎

Students of calculus may recognize that the argument used here is essentially like that used to prove the classical Bolzano-Weierstrass theorem, "A bounded, infinite set of real numbers has an accumulation point." One way of stating

Theorem K, as König observed, is: "Assuming the human race never dies out, there is a man now living having a line of descendants that will never die out."

Most people think that Theorem K is completely obvious when they first encounter it, but after more thought and a consideration of further examples they realize that there is something "profound" about the infinity lemma. Although the degree of each node of the tree is finite, we have not assumed it is *bounded* (less than some number N for all vertices), so there may be nodes with higher and higher degrees. If we stop to consider things carefully, it is at least conceivable that everyone's descendants will ultimately die out although there will be some families that go on a million generations, others a billion, etc., etc. In fact, H. W. Watson once published a "proof" that under certain laws of biological probability carried out indefinitely, there will be infinitely many people born in the future but each family line will die out with probability one. His paper [*J. Anthropological Inst. Gt. Britain and Ireland* **4** (1874), 138–144] actually contains important and far-reaching theorems in spite of the minor slip which caused him to make this erroneous statement, and it is significant that he did not find his conclusions to be logically inconsistent.

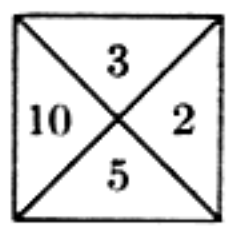
The contrapositive of Theorem K is directly applicable to computer algorithms: "If we have an algorithm that periodically divides itself up into finitely many subalgorithms, and if each chain of subalgorithms ultimately terminates, then the algorithm itself terminates."

Phrased yet another way, suppose we have a set S, finite or infinite, such that each element of S is a sequence $(x_1, x_2, \ldots, x_n)$ of positive integers of finite length $n \geq 0$. If we impose the conditions that

i) If $(x_1, \ldots, x_n)$ is in S, so is $(x_1, \ldots, x_k)$ for $0 \leq k \leq n$.
ii) If $(x_1, \ldots, x_n)$ is in S, only finitely many x_{n+1} exist for which $(x_1, \ldots, x_n, x_{n+1})$ is also in S.
iii) There is no infinite sequence $(x_1, x_2, \ldots)$ all of whose initial subsequences $(x_1, x_2, \ldots, x_n)$ lie in S.

Then S is essentially an oriented tree, specified essentially in a Dewey decimal notation, and Theorem K tells us S is *finite*.

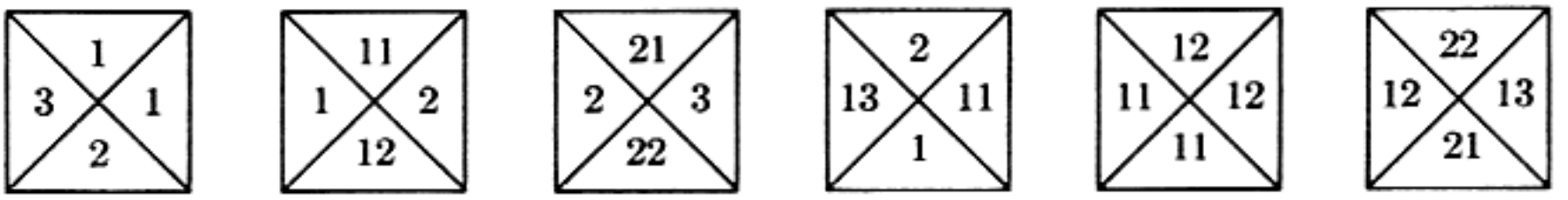
One of the most convincing examples of the potency of Theorem K has recently been given by Hao Wang, in connection with his "domino problem." A *domino type* is a square divided into four parts, each part having a specified number in it, e.g.,



(1)

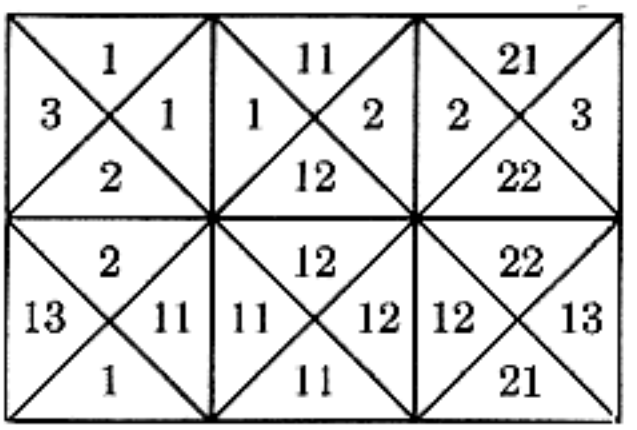
The problem of *tiling the plane* is to take a finite set of domino types, with an infinite supply of dominoes of each type, and to show how to place one in each square of an infinite plane (without rotating or reflecting the domino types) such that two dominoes are adjacent only if they have equal numbers where

they touch. For example, we can tile the plane using the six domino types



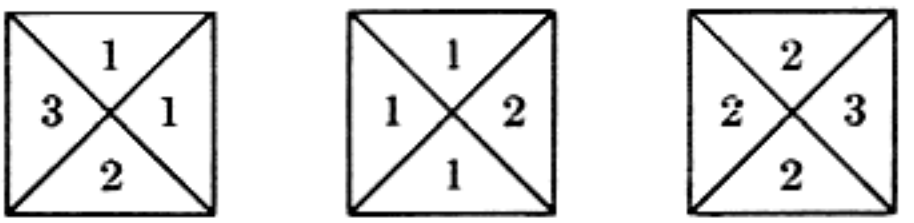
(2)

in essentially only one way, by repeating the rectangle



(3)

over and over. The reader may easily verify that there is no way to tile the plane with the three domino types



(4)

Wang's observation [see *Sci. Am.* **213** (November, 1965), 98–106] is that *if it is possible to tile the upper right quadrant of the plane, it is possible to tile the whole plane.* This is certainly unexpected, since a method for tiling the upper right quadrant involves a "boundary" along the x- and y-axes, and it would seem to give no hint as to how to tile the upper left quadrant of the plane (since domino types may not be rotated or reflected). We cannot get rid of the boundary merely by shifting the upper-quadrant solution down and to the left, since it does not make sense to shift the solution by more than a finite amount. But Wang's proof runs as follows: The existence of an upper-right-quadrant solution implies there is a way to tile a $2n \times 2n$ square, for all n. The set of all solutions to the problem of tiling squares with an even number of cells on each side forms an oriented tree, if the sons of each $2n \times 2n$ solution x are the possible $(2n+2) \times (2n+2)$ solutions that can be obtained by bordering x. The root of this oriented tree is the 0×0 solution; its sons are the 2×2 solutions, etc. Each node has only finitely many sons, since the problem of tiling the plane assumes only finitely many domino types are given; hence by the infinity lemma there is an infinite path from the root. This means there is a way to tile the whole plane!

EXERCISES

1. [*M10*] The text refers to a set S containing finite sequences of positive integers, and states that this set is "essentially an oriented tree." What is the root of this oriented tree, and what are the arcs?

2. [*20*] Show that if rotation of domino types is allowed, it is always possible to tile the plane.

▶ **3.** [*M23*] If it is possible to tile the upper right quadrant of the plane when given an *infinite* set of domino types, is it always possible to tile the whole plane?

4. [*M25*] (H. Wang.) The six domino types (2) lead to a "toroidal" solution to the tiling problem, i.e., a solution in which some rectangular pattern [namely (3)] is replicated throughout the entire plane.

Assume without proof that whenever it is possible to tile the plane with a finite set of domino types, there is a toroidal solution using those domino types. Use this assumption together with the infinity lemma to design an algorithm which, given the specifications of any finite set of domino types, determines in a finite number of steps whether or not there exists a way to tile the plane with these types.

5. [*M40*] Show that using the following 92 domino types it is possible to tile the plane, but that there is no "toroidal" solution in the sense defined in exercise 4.

To simplify the specification of the 92 types, let us first introduce some notation. Define the following "basic codes":

$$\begin{array}{llll}
\alpha = (1, 2, 1, 2) & \beta = (3, 4, 2, 1) & \gamma = (2, 1, 3, 4) & \delta = (4, 3, 4, 3) \\
a = (Q, D, P, R) & b = (\ ,\ , L, P) & c = (U, Q, T, S) & d = (\ ,\ , S, T) \\
N = (Y,\ , X,\ ) & J = (D, U,\ , X) & K = (\ , Y, R, L) & B = (\ ,\ ,\ ,\ ) \\
R = (\ ,\ , R, R) & L = (\ ,\ , L, L) & P = (\ ,\ , P, P) & S = (\ ,\ , S, S) \\
 & T = (\ ,\ , T, T) & X = (\ ,\ , X, X) & \\
Y = (Y, Y,\ ,\ ) & U = (U, U,\ ,\ ) & D = (D, D,\ ,\ ) & Q = (Q, Q,\ ,\ )
\end{array}$$

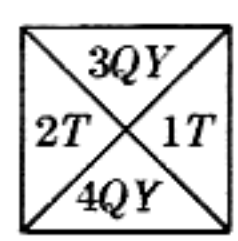
The domino types are now

$$\begin{array}{ll}
\alpha\{a, b, c, d\} & \text{[4 types]} \\
\beta\{Y\{B, U, Q\}\{P, T\}, \{B, U, D, Q\}\{P, S, T\}, K\{B, U, Q\}\} & \text{[21 types]} \\
\gamma\{\{\{X, B\}\{L, P, S, T\}, R\}\{B, Q\}, J\{L, P, S, T\}\} & \text{[22 types]} \\
\delta\{X\{L, P, S, T\}\{B, Q\}, Y\{B, U, Q\}\{P, T\}, N\{a, b, c, d\}, & \\
\quad J\{L, P, S, T\}, K\{B, U, Q\}, \{R, L, P, S, T\}\{B, U, D, Q\}\} & \text{[45 types]}
\end{array}$$

These abbreviations mean that the basic codes are to be put together component by component and sorted into alphabetic order in each component, thus: $\beta Y\{B, U, Q\}\{P, T\}$ stands for six types βYBP, βYUP, βYQP, βYBT, βYUT, βYQT. The type βYQT is

$$(3, 4, 2, 1)(Y, Y,\ ,\ )(Q, Q,\ ,\ )(\ ,\ , T, T) = (3QY, 4QY, 2T, 1T)$$

after multiplying corresponding components and sorting into order. This is intended to correspond to the domino type shown below where we use strings of symbols instead of numbers in the four quarters of the type. Two domino types can be placed next to each other only if they have the same string of symbols at the place they touch.



A domino type of "class β" means one which has a β in its specification as given above. To get started on the solution to this exercise, note that any domino of class β must have one of class α to its left and to its right, and that there must be one of class δ above and below. An "αa" domino must have "βKB" or "βKU" or "βKQ" to its right, and then must come an "αb" domino, etc.

(The above construction is a simplified version of a similar one given by Robert Berger, who went on to prove that the general problem in exercise 4, without the invalid assumption, cannot be solved. See *Memoirs Amer. Math. Soc.* **66** (1966).)

▶ **6.** [*M23*] (D. König.) In a famous paper [*Nieuw Archief voor Wiskunde* (2) **15** (1927), 212–216], B. L. van der Waerden proved that:

> "If k and m are positive integers, and if we have k sets $S_1, \ldots, S_k$ of positive integers with every positive integer included in at least one of these sets, then at least one of the sets S_j contains an arithmetic progression of length m."

(The latter statement means there exist integers a and $\delta > 0$ such that $a + \delta$, $a + 2\delta$, $\ldots$, $a + m\delta$ are all in S_j.) If possible, use this result and the infinity lemma to prove the stronger statement:

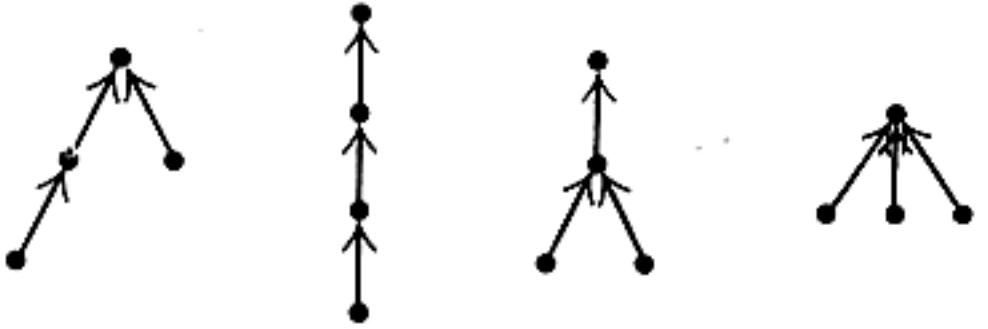
> "If k and m are positive integers, there is a number N such that if we have k sets $S_1, \ldots, S_k$ of integers with every integer between 1 and N included in at least one of these sets, then at least one of the sets S_j contains an arithmetic progression of length m."

▶ **7.** [*M30*] If possible, use van der Waerden's theorem of exercise 6 and the infinity lemma to prove the stronger statement:

> "If k is a positive integer, and if we have k sets $S_1, \ldots, S_k$ of integers with every positive integer included in at least one of these sets, then at least one of the sets S_j contains an infinitely long arithmetic progression."

▶ **8.** [*M34*] (J. B. Kruskal, *Trans. Am. Math. Soc.* **95** (1960), 210–225.) If T and T' are (finite, ordered) trees, let the notation $T \subseteq T'$ signify that T can be embedded in T', as in exercise 2.3.2–22. Prove that if $T_1, T_2, T_3, \ldots$ is any infinite sequence of trees, there exist integers $j < k$ such that $T_j \subseteq T_k$. (In other words, it is impossible to construct an infinite sequence of trees in which no tree "contains" any of the earlier trees of the sequence. This fact may be used to prove that certain algorithms must terminate.)

2.3.4.4. Enumeration of trees. Some of the most direct applications of the mathematical theory of trees to the analysis of algorithms are connected with formulas for counting how many different trees there are of various kinds. For example, if we want to know how many different oriented trees can be constructed having four vertices, we find there are just 4 possibilities:



(1)

For our first enumeration problem, let us determine the number a_n of structurally different oriented trees with n vertices. Obviously $a_1 = 1$. If $n > 1$, the tree has a root and various subtrees; suppose there are j_1 subtrees with 1 vertex, j_2 with 2 vertices, etc. Then we may choose j_k of the a_k possible trees in

$$\binom{a_k + j_k - 1}{j_k}$$

ways, since repetitions are allowed (cf. exercise 1.2.6–60), and so we see that

$$a_n = \sum_{j_1+2j_2+\cdots=n-1} \binom{a_1 + j_1 - 1}{j_1} \cdots \binom{a_{n-1} + j_{n-1} - 1}{j_{n-1}}. \tag{2}$$

If we consider the generating function $A(z) = \sum_n a_n z^n$, with $a_0 = 0$, we find that the identity

$$\frac{1}{(1 - z^r)^a} = \sum_j \binom{a + j - 1}{j} z^{rj}$$

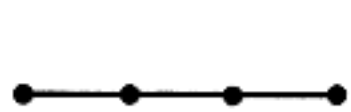
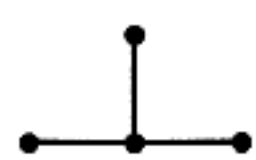
together with (2) implies

$$A(z) = z/(1 - z)^{a_1}(1 - z^2)^{a_2}(1 - z^3)^{a_3} \cdots . \tag{3}$$

This is not an especially nice form for $A(z)$, since it involves an infinite product and the coefficients $a_1, a_2, \ldots$ appear on the right-hand side; a somewhat more aesthetic way to represent $A(z)$ is given in exercise 1, and this leads to a reasonably efficient formula for calculating the values a_n (see exercise 2) and, in fact, it also shows the asymptotic behavior of a_n for large n (see exercise 4). We find that

$$\begin{aligned} A(z) = {} & z + z^2 + 2z^3 + 4z^4 + 9z^5 + 20z^6 + 48z^7 + 115z^8 \\ & + 286z^9 + 719z^{10} + 1842z^{11} + \cdots \end{aligned} \tag{4}$$

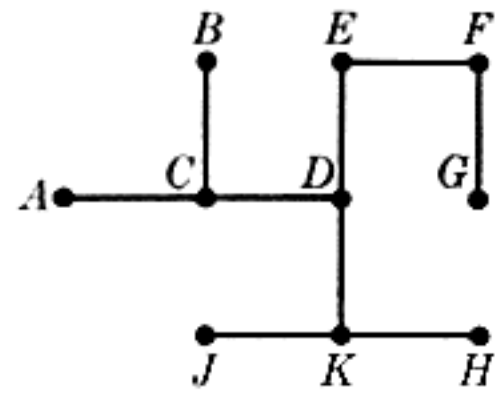
Now that we have essentially found the number of oriented trees, it is quite interesting to determine the number of structurally different *free trees* with n vertices. There are just two distinct free trees with four vertices, namely

 and  (5)

because the first two and last two oriented trees of (1) become identical when the orientation is dropped.

We have seen that it is possible to select any vertex X of a free tree and to assign directions to the edges in a unique way so that it becomes an oriented tree with X as root. Once this has been done, for a given vertex X, suppose there are k subtrees of the root X, with $s_1, s_2, \ldots, s_k$ vertices in these respective

subtrees. Clearly, k is the number of arcs touching X; and $s_1 + s_2 + \cdots + s_k = n - 1$, one less than the total number of vertices in the free tree. In these circumstances, we say that the *height* of X is $\max(s_1, s_2, \ldots, s_k)$. Thus in the tree



(6)

the vertex D has height 3 (each of the subtrees leading from D has three of the nine remaining vertices), and vertex E has height $\max(7, 2) = 7$. A vertex with minimum height is called a *centroid* of the free tree.

Let X and $s_1, s_2, \ldots, s_k$ be as above, and let $Y_1, Y_2, \ldots, Y_k$ be the roots of the subtrees emanating from X. Clearly, the height of Y_1 is at least $n - s_1 = 1 + s_2 + \cdots + s_k$, since when Y_1 is the assumed root there are $n - s_1$ points in its subtree through X. If there is a centroid Y in the Y_1 subtree, we have

$$\text{height}(X) = \max(s_1, s_2, \ldots, s_k) \geq \text{height}(Y) \geq 1 + s_2 + \cdots + s_k,$$

and this implies $s_1 > s_2 + \cdots + s_k$. A similar result may be derived if we replace Y_1 by Y_j in this discussion. So *at most one of the subtrees at a vertex can contain a centroid.*

This is a strong condition, for it implies that *there are at most two centroids in a free tree, and if two centroids exist, they are adjacent.* (See exercise 9.)

Conversely, if $s_1 > s_2 + \cdots + s_k$, there *is* a centroid in the Y_1 subtree, since

$$\text{height}(Y_1) \leq \max(s_1 - 1, 1 + s_2 + \cdots + s_k) \leq s_1 = \text{height}(X),$$

and the height of all nodes in the $Y_2, \ldots, Y_k$ subtrees is at least $s_1 + 1$. We have proved *the vertex X is the only centroid of a free tree if and only if*

$$s_j \leq s_1 + \cdots + s_k - s_j, \qquad \text{for} \qquad 1 \leq j \leq k. \tag{7}$$

Therefore the number of free trees with n vertices, having only one centroid, is the number of oriented trees with n vertices minus the number of oriented trees violating condition (7); the latter consist of an oriented tree with s_j vertices and another oriented tree with $n - s_j \leq s_j$ vertices. The number with one centroid therefore comes to

$$a_n - a_1 a_{n-1} - a_2 a_{n-2} - \cdots - a_{\lfloor n/2 \rfloor} a_{\lceil n/2 \rceil}. \tag{8}$$

A free tree with two centroids has an even number of vertices, and the height of each centroid is $n/2$ (see exercise 10). So if $n = 2m$, the number of bicen-

troidal free trees is the number of choices of 2 things out of a_m with repetition, namely

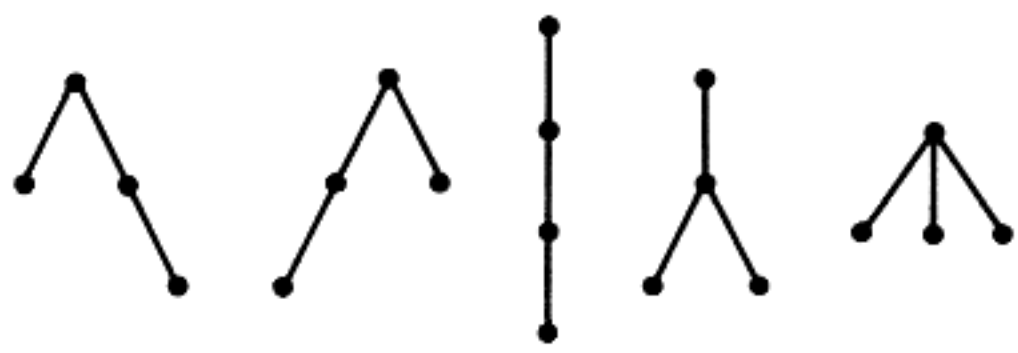
$$\binom{a_m + 1}{2}.$$

Thus, to get the total number of free trees, we add $\frac{1}{2}a_{n/2}(a_{n/2} + 1)$ to (8) when n is even. The form of Eq. (8) suggests a simple generating function, and, indeed, we find without difficulty that *the generating function for the number of structurally different free trees is*

$$\begin{aligned} F(z) &= A(z) - \tfrac{1}{2}A(z)^2 + \tfrac{1}{2}A(z^2) \\ &= z + z^2 + z^3 + 2z^4 + 3z^5 + 6z^6 + 11z^7 + 23z^8 \\ &\quad + 47z^9 + 106z^{10} + 235z^{11} + \cdots. \end{aligned} \tag{9}$$

This simple relation between $F(z)$ and $A(z)$ was first obtained by R. Otter in 1948.

Now let us turn to the question of enumerating *ordered trees*, which are our principal concern with respect to computer programming algorithms. There are five structurally different ordered trees with four vertices:



(10)

[The first two of these are identical as oriented trees, so only one of them appears in (1) above.]

Before we examine the number of different ordered tree structures, let us first consider the case of *binary trees*, since this is closer to actual computer representation and it is easier to study. Let b_n be the number of different binary trees with n nodes. From the definition of binary tree it is apparent that $b_0 = 1$, and for $n > 0$ the number of possibilities is the number of ways to put a binary tree with k nodes to the left of the root and another with $n - 1 - k$ nodes to the right. So

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-1} b_0, \qquad n \geq 1. \tag{11}$$

From this relation it is clear that the generating function

$$B(z) = b_0 + b_1 z + b_2 z^2 + \cdots$$

satisfies the equation

$$zB(z)^2 = B(z) - 1.$$

Solving this quadratic equation and using the fact that $B(0) = 1$, we obtain

$$\begin{aligned} B(z) &= \frac{1}{2z}\left(1 - \sqrt{1-4z}\right) \\ &= \frac{1}{2z}\left(1 - \sum_{n\geq 0}\binom{\frac{1}{2}}{n}(-4z)^n\right) \\ &= \sum_{m\geq 0}\binom{\frac{1}{2}}{m+1}(-1)^m 2^{2m+1} z^m \\ &= 1 + z + 2z^2 + 5z^3 + 14z^4 + 42z^5 + 132z^6 + 429z^7 \\ &\qquad + 1430z^8 + 4862z^9 + 16796z^{10} + \cdots . \end{aligned} \tag{12}$$

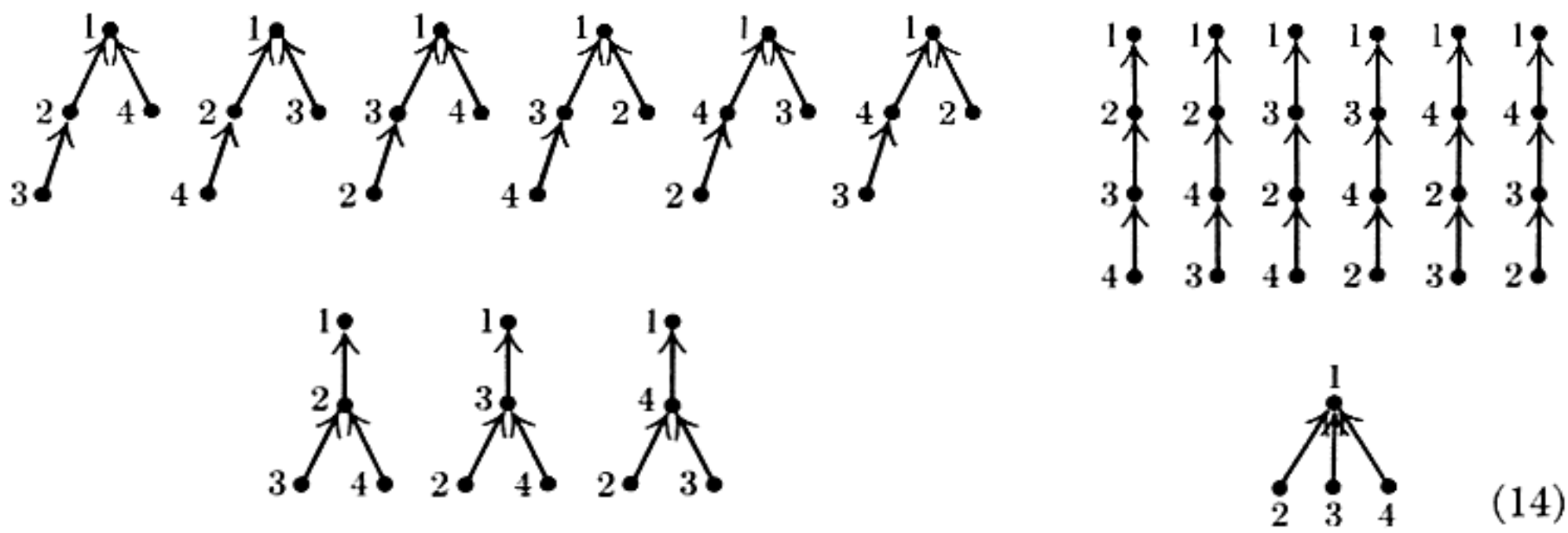
The desired answer is therefore

$$b_n = \binom{\frac{1}{2}}{n+1}(-1)^n 2^{2n+1} = \frac{1}{n+1}\binom{2n}{n}. \tag{13}$$

By Stirling's approximation, this is asymptotically $4^n/n\sqrt{\pi n} + O(4^n n^{-5/2})$.

Returning to our question about ordered trees with n nodes, we can see that this is essentially the same question as the number of binary trees, since we have a standard correspondence between binary trees and forests, and a tree minus its root is a forest. Hence *the number of (ordered) trees with n vertices is b_{n-1}, the number of binary trees with $n - 1$ vertices.*

The enumerations performed above assume that the vertices are indistinguishable points. If we label the vertices 1, 2, 3, 4 in (1) and insist that 1 is to be the root, we now get 16 different oriented trees:



(14)

The question of enumeration for labeled trees is clearly quite different from the one solved above. In this case it can be rephrased as follows: "Consider drawing lines pointing from each of the vertices 2, 3, and 4 to another vertex; there are three choices of lines emanating from each vertex, so there are $3^3 = 27$ possibilities in all. How many of these 27 ways will yield oriented trees with 1 as the root?" The answer, as we have seen, is 16. A similar reformulation of the

same problem, this time for the case of n vertices, is the following: "Let $f(x)$ be an integer-valued function such that $f(1) = 1$ and $1 \leq f(x) \leq n$ for all integers $1 \leq x \leq n$. We call f a 'tree function' if $f^n(x)$, that is, $f(f(\cdots(f(x))\cdots))$ iterated n times, equals 1, for all x. How many tree functions are there?" This problem comes up, for example, in connection with random number generation. We will find, rather surprisingly, that on the average exactly one out of every n such functions f is a tree function.

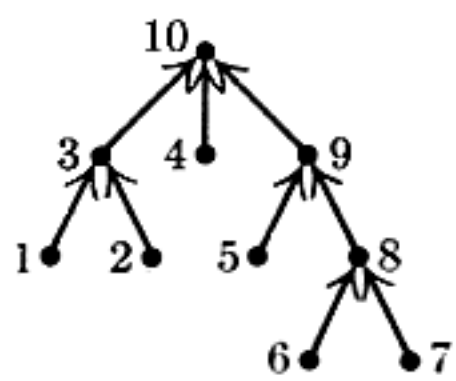
The solution to this enumeration problem can readily be derived using the general formulas for counting subtrees of graphs that have been developed in previous sections (see exercise 12). But there is a much more informative way to solve the problem, because it gives us a new and compact manner to represent oriented tree structure.

Let us suppose we are given an oriented tree with vertices $\{1, 2, \ldots, n\}$ and with $n - 1$ arcs, where the arcs go from j to $f(j)$ for all j except the root. There is at least one terminal vertex (leaf); let V_1 be the smallest number of a terminal vertex. If $n > 1$, write down $f(V_1)$ and delete both V_1 and the arc from V_1 to $f(V_1)$ from the tree; and then let V_2 be the smallest number whose vertex is terminal in the resulting tree. If $n > 2$, write down $f(V_2)$ and delete both V_2 and the arc from V_2 to $f(V_2)$ from the tree; and proceed in this way until all vertices have been deleted except the root. The resulting sequence of $n - 1$ numbers,

$$f(V_1), f(V_2), \ldots, f(V_{n-1}), \tag{15}$$

with $1 \leq f(V_j) \leq n$, is called the *canonical representation* of the original oriented tree.

For example, the oriented tree



(16)

with 10 vertices has the canonical representation 3, 3, 10, 10, 9, 8, 8, 9, 10.

The important point here is that we can reverse this process and go from any sequence of $n - 1$ numbers (15) back to an oriented tree which produced it. For if we have any sequence $x_1, x_2, \ldots, x_{n-1}$ of numbers between 1 and n, let V_1 be the smallest number which does not appear in the sequence $x_1, \ldots, x_{n-1}$; then let V_2 be the smallest number $\neq V_1$ which does not appear in the sequence $x_2, \ldots, x_{n-1}$; and so on. After obtaining a permutation $V_1, V_2, \ldots, V_n$ of the integers $1, 2, \ldots, n$ in this way, draw arcs from vertex V_j to vertex x_j, for $1 \leq j < n$. This gives a construction of a directed graph with no oriented cycles, and by exercise 2.3.4.2–7 it is an oriented tree. Clearly, the sequence $x_1, x_2, \ldots, x_{n-1}$ is the same as the sequence (15) for this oriented tree.

Since the process is reversible, we have obtained a one-to-one correspondence between $(n-1)$-tuples of numbers $\{1, 2, \ldots, n\}$ and oriented trees on these vertices. Hence *there are n^{n-1} distinct oriented trees with n labeled vertices.* If we specify that one vertex is to be the root, there is clearly no difference between one vertex and another, so there are n^{n-2} distinct oriented trees on $\{1, 2, \ldots, n\}$ having a given root. This accounts for the $16 = 4^{4-2}$ trees in (14). From this information it is easy to determine the number of *free trees* with labeled vertices (see exercise 22). The number of *ordered trees* with labeled vertices is also easy to determine once we know the answer to this problem when no labels are involved (see exercise 23). So the problems of enumerating the three fundamental classes of trees, with both labeled and unlabeled vertices, have now been essentially resolved in this section.

It is interesting to see what would happen if we were to apply our usual method of generating functions to the problem of enumerating labeled oriented trees. For this purpose we would probably find it easiest to consider the quantity $r(n, q)$, the number of labeled directed graphs with n vertices, with no oriented cycles, and with one arc emanating from each of q designated vertices. The number of labeled oriented trees with a specified root is therefore $r(n, n-1)$. In this notation we find by simple counting arguments that, for fixed m,

$$r(n, q) = \sum_{\substack{k,t \\ t-k=m}} \binom{q}{k} r(t, k) r(n-t, q-k), \qquad \text{if} \qquad 0 < m < n - q, \tag{17}$$

$$r(n, q) = \sum_{k} \binom{q}{k} r(n-1, q-k), \qquad \text{if} \qquad q = n - 1. \tag{18}$$

The first of these relations is obtained if we partition the undesignated vertices into two groups A and B, with m vertices in A and $n - q - m$ vertices in B; then the q designated vertices are partitioned into k vertices, which begin paths leading into A, and $q - k$ vertices, which begin paths leading into B. Relation (18) is obtained by considering oriented trees in which the root has degree k.

The form of these relations indicates that we can work profitably with the generating function

$$G_m(z) = r(m, 0) + r(m+1, 1)z + \frac{r(m+2, 2)z^2}{2!} = \cdots = \sum_k \frac{r(k+m, k)z^k}{k!}.$$

In these terms Eq. (17) says that $G_{n-q}(z) = G_m(z)G_{n-q-m}(z)$, and therefore by induction on m, we find that $G_m(z) = G_1(z)^m$. Now from Eq. (18), we obtain

$$G_1(z) = \sum_{n \geq 1} \frac{r(n, n-1)z^{n-1}}{(n-1)!} = \sum_{k \geq 0} \sum_{n \geq 1} \frac{r(n-1, n-1-k)z^{n-1}}{k!}$$

$$= \sum_{k \geq 0} \frac{z^k}{k!} G_k(z) = \sum_{k \geq 0} \frac{(zG_1(z))^k}{k!} = e^{zG_1(z)}.$$

In other words, putting $G_1(z) = w$, the solution to our problem comes from the coefficients of the solution to the transcendental equation

$$w = e^{zw}. \tag{19}$$

This equation can be solved with the use of Lagrange's inversion formula, i.e., $z = \zeta/f(\zeta)$ implies that

$$\zeta = \sum_{n \geq 1} \frac{z^n}{n!} g_n^{(n-1)}(0),$$

where $g_n(\zeta) = f(\zeta)^n$, when f is analytic in the neighborhood of the origin, and $f(0) \neq 0$ (see exercise 33). In this case, we may set $\zeta = zw$, $f(\zeta) = e^{\zeta}$, and we deduce the solution

$$w = \sum_{n \geq 0} \frac{(n+1)^{n-1}}{n!} z^n, \tag{20}$$

in agreement with the answer obtained above.

G. N. Raney has shown that we can extend this method in an important way to obtain an explicit power series for the solution to the considerably more general equation

$$w = y_1 e^{z_1 w} + y_2 e^{z_2 w} + \cdots + y_s e^{z_s w},$$

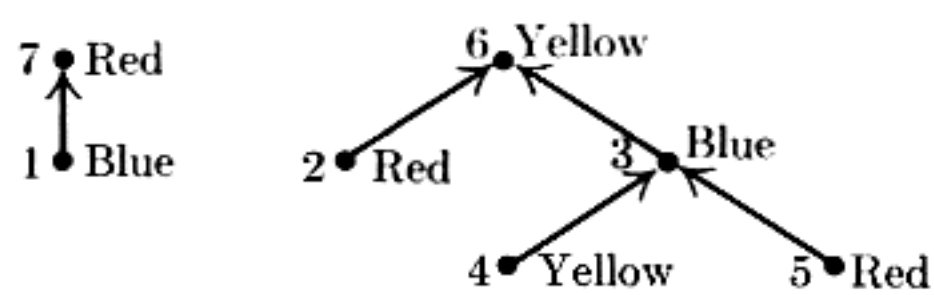
solving for w in terms of a power series in $y_1, \ldots, y_s$ and $z_1, \ldots, z_s$. For this generalization, let us consider s-dimensional vectors of integers

$$\mathbf{n} = (n_1, n_2, \ldots, n_s),$$

and let us write for convenience

$$\sum \mathbf{n} = n_1 + n_2 + \cdots + n_s.$$

Suppose that we have s "colors" $C_1, C_2, \ldots, C_s$, and consider directed graphs in which each vertex is assigned a color, e.g.,



(21)

Let $r(\mathbf{n}, \mathbf{q})$ be the number of ways to draw arcs and to assign colors to the vertices $\{1, 2, \ldots, n\}$, such that

i) for $1 \leq i \leq s$ there are exactly n_i vertices of color C_i (hence $n = \sum \mathbf{n}$);
ii) there are q arcs, one leading from each of the vertices $\{1, 2, \ldots, q\}$;
iii) for $1 \leq i \leq s$ there are exactly q_i arcs leading to vertices of color C_i (hence $q = \sum \mathbf{q}$);
iv) there are no oriented cycles (hence $q < n$).

Let us call this an $(\mathbf{n}, \mathbf{q})$-construction.

For example, if C_1 = red, C_2 = yellow, and C_3 = blue, then (21) shows a $((3, 2, 2), (1, 2, 2))$-construction. When there is only one color, we have the oriented tree problem which we have already solved.

Let $\mathbf{n}$ and $\mathbf{q}$ be fixed s-place vectors of nonnegative integers, and let $n = \sum \mathbf{n}$, $q = \sum \mathbf{q}$. For each $(\mathbf{n}, \mathbf{q})$-construction and each number k, $1 \leq k \leq n$, we will define a *canonical representation* consisting of four things:

a) a number t, with $q < t \leq n$;
b) a sequence of n colors, with n_i of color C_i;
c) a sequence of q colors, with q_i of color C_i;
d) for $1 \leq i \leq s$, a sequence of q_i elements of the set $\{1, 2, \ldots, n_i\}$.

The canonical representation is defined as follows: First list the vertices $\{1, 2, \ldots, q\}$ in the order $V_1, V_2, \ldots, V_q$ of the canonical representation of oriented trees (as given above), and then write below vertex V_j the number $f(V_j)$ of the vertex on the arc leading from V_j. Let $t = f(V_q)$; and let the sequence (c) of colors be the respective colors of the vertices $f(V_1), \ldots, f(V_q)$. Let the sequence (b) of colors be the respective colors of the vertices $k, k+1, \ldots, n, 1, \ldots, k-1$. Finally, let the ith sequence in (d) be $x_{i1}, x_{i2}, \ldots, x_{iq_i}$, where $x_{ij} = m$ if the jth C_i-colored element of the sequence $f(V_1), \ldots, f(V_q)$ is the mth C_i-colored element of the sequence $k, k+1, \ldots, n, 1, \ldots, k-1$.

For example, consider construction (21) and let $k = 3$. We start by listing $V_1, \ldots, V_5$ and $f(V_1), \ldots, f(V_5)$ below them as follows:

1	2	4	5	3
7	6	3	3	6

Hence $t = 6$, and sequence (c) represents the respective colors of 7, 6, 3, 3, 6, namely red, yellow, blue, blue, yellow. Sequence (c) represents the respective colors of 3, 4, 5, 6, 7, 1, 2, namely blue, yellow, red, yellow, red, blue, red. Finally, to get the sequences in (d), proceed as follows:

color	*elements this color in* 3, 4, 5, 6, 7, 1, 2	*elements this color in* 7, 6, 3, 3, 6	*encode column 3 by column 2*
red	5, 7, 2	7	2
yellow	4, 6	6, 6	2, 2
blue	3, 1	3, 3	1, 1

Hence the (d) sequences are 2; 2, 2; and 1, 1.

From the canonical representation, we can recover both the original $(\mathbf{n}, \mathbf{q})$-construction and the number k as follows: From (a) and (c) we know the color of vertex t. The last element of the (d) sequence for this color tells us, in conjunction with (b), the position of t in the sequence $k, \ldots, n, 1, \ldots, k-1$; hence we know k and the colors of all vertices. Then the subsequences in (d) together with (b) and (c) determine $f(V_1), f(V_2), \ldots, f(V_q)$, and finally the directed graph is reconstructed by locating $V_1, \ldots, V_q$ as we did for oriented trees.

The reversibility of this canonical representation allows us to count the number of possible $(\mathbf{n}, \mathbf{q})$-constructions, since there are $n - q$ choices for (a), the multinomial coefficient

$$\binom{n}{n_1, \ldots, n_s}$$

choices for (b),

$$\binom{q}{q_1, \ldots, q_s}$$

choices for (c), and $n_1^{q_1} n_2^{q_2} \cdots n_s^{q_s}$ choices for (d). Dividing by the n choices for k, we have the general result

$$r(\mathbf{n}, \mathbf{q}) = \frac{n-q}{n} \frac{n!}{n_1! \cdots n_s!} \frac{q!}{q_1! \cdots q_s!} n_1^{q_1} n_2^{q_2} \cdots n_s^{q_s}. \tag{22}$$

Furthermore, we can derive analogs of Eqs. (17) and (18):

$$r(\mathbf{n}, \mathbf{q}) = \sum_{\substack{\mathbf{k},\mathbf{t} \\ \Sigma(\mathbf{t}-\mathbf{k})=m}} \binom{\Sigma\mathbf{q}}{\Sigma\mathbf{k}} r(\mathbf{t}, \mathbf{k}) r(\mathbf{n} - \mathbf{t}, \mathbf{q} - \mathbf{k}), \quad \text{if} \quad 0 < m < \Sigma(\mathbf{n} - \mathbf{q}), \tag{23}$$

with the convention that $r(\mathbf{0}, \mathbf{0}) = 1$ and $r(\mathbf{n}, \mathbf{q}) = 0$ if any n_i or q_i is negative or if $q > n$;

$$r(\mathbf{n}, \mathbf{q}) = \sum_{1 \le i \le s} \sum_{k} \binom{\Sigma\mathbf{q}}{k} r(\mathbf{n} - \mathbf{e}_i, \mathbf{q} - k\mathbf{e}_i), \quad \text{if} \quad \Sigma\mathbf{n} = 1 + \Sigma\mathbf{q}, \tag{24}$$

where $\mathbf{e}_i$ is the vector with 1 in position i and zeros elsewhere. Relation (23) is based on breaking the vertices $\{q + 1, \ldots, n\}$ into two parts having m and $n - q - m$ elements, respectively; the second relation is derived by removing the unique root and considering the remaining structure. We now obtain the following result:

Theorem R (George N. Raney, *Canadian J. Math.* **16** (1964), 755–762). *Let*

$$w = \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=1}} \frac{r(\mathbf{n}, \mathbf{q})}{(\Sigma\mathbf{q})!} y_1^{n_1} \cdots y_s^{n_s} z_1^{q_1} \cdots z_s^{q_s}, \tag{25}$$

where $r(\mathbf{n}, \mathbf{q})$ is defined by (22), *and where $\mathbf{n}$, $\mathbf{q}$ are s-dimensional integer vectors. Then w satisfies the identity*

$$w = y_1 e^{z_1 w} + y_2 e^{z_2 w} + \cdots + y_s e^{z_s w}. \tag{26}$$

Proof. By (23) and induction on m, we find that

$$w^m = \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=m}} \frac{r(\mathbf{n}, \mathbf{q})}{(\Sigma\mathbf{q})!} y_1^{n_1} \cdots y_s^{n_s} z_1^{q_1} \cdots z_s^{q_s}. \tag{27}$$

Now by (24),

$$w = \sum_{1\le i\le s} \sum_{k} \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=1}} \frac{r(\mathbf{n}-\mathbf{e}_i, \mathbf{q}-k\mathbf{e}_i)}{k!(\Sigma\mathbf{q}-k)!} y_1^{n_1}\cdots y_s^{n_s} z_1^{q_1}\cdots z_s^{q_s}$$

$$= \sum_{1\le i\le s} \sum_{k} \frac{1}{k!} y_i z_i^k \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=k}} \frac{r(\mathbf{n}, \mathbf{q})}{(\Sigma\mathbf{q})!} y_1^{n_1}\cdots y_s^{n_s} z_1^{q_1}\cdots z_s^{q_s}$$

$$= \sum_{1\le i\le s} \sum_{k} \frac{1}{k!} y_i z_i^k w^k. \quad \blacksquare$$

A survey of enumeration formulas for trees, based on skillful manipulations of generating functions, has been given by I. J. Good [*Proc. Cambridge Philos. Soc.* **61** (1965), 499–517; **64** (1968), 489]. This important paper contains extensive generalizations of many of the formulas derived in this section.

EXERCISES

1. [*M20*] (G. Polya.) Show that

$$A(z) = z \cdot \exp\left(A(z) + \tfrac{1}{2}A(z^2) + \tfrac{1}{3}A(z^3) + \cdots\right).$$

[*Hint:* Take logarithms of (3).]

2. [*HM24*] (R. Otter.) Show that the numbers a_n satisfy the following condition:

$$na_{n+1} = a_1 s_{n1} + 2a_2 s_{n2} + \cdots + na_n s_{nn},$$

where

$$s_{nk} = \sum_{1 \le j \le n/k} a_{n+1-jk}.$$

(These formulas are useful for the calculation of the a_n, since $s_{nk} = s_{(n-k)k} + a_{n+1-k}$.)

3. [*M40*] Write a computer program which determines the number of (unlabeled) free trees and of oriented trees with n vertices, for $n \le 100$. (Use the result of exercise 2.) Explore arithmetical properties of these numbers; can anything be said about their prime factors, or their residues modulo p?

▸ **4.** [*HM39*] (R. Otter, *Ann. Math.* **49** (1948), 583–599.) Using complex variable theory, determine the asymptotic value of the number of oriented trees as follows: (See exercise 1). (a) Show that there is a real number α between 0 and 1 for which $A(z)$ has radius of convergence α and $A(z)$ converges absolutely for all complex z such that $|z| \le \alpha$, having maximum value $A(\alpha) = a < \infty$. [*Hint:* When a power series has nonnegative coefficients, it either is entire or it has a positive real singularity; and show that $A(z)/z$ is bounded as $z \to \alpha-$.] (b) Let

$$F(z, w) = \exp\left(zw + \tfrac{1}{2}A(z^2) + \tfrac{1}{3}A(z^3) + \cdots\right) - w.$$

Show that in a neighborhood of $(z, w) = (\alpha, a/\alpha)$, $F(z, w)$ is analytic in each variable separately. (c) Show that at the point $(z, w) = (\alpha, a/\alpha)$, $\partial F/\partial w = 0$; hence $a = 1$. (d) At the point $(z, w) = (\alpha, 1/\alpha)$ show that

$$\frac{\partial F}{\partial z} = \beta = \alpha^{-2} + \sum_{k \geq 2} \alpha^{k-2} A'(\alpha^k), \qquad \text{and} \qquad \frac{\partial^2 F}{\partial w^2} = \alpha.$$

(e) When $|z| = \alpha$ and $z \neq \alpha$, show that $\partial F/\partial w \neq 0$; hence $A(z)$ has only one singularity on $|z| = \alpha$. (f) Prove there is a region larger than $|z| < \alpha$ in which

$$\frac{1}{z} A(z) = \frac{1}{\alpha} - \sqrt{2\beta(1 - z/\alpha)} + R(z),$$

where $R(z)$ is an analytic function of $\sqrt{z - \alpha}$. (g) Prove that consequently

$$a_n = \frac{1}{\alpha^{n-1} n} \sqrt{\beta/2\pi n} + O(n^{-5/2} \alpha^{-n}).$$

(*Note:* $1/\alpha = 2.95576$, and $\alpha\sqrt{\beta/2\pi} = 0.43992$.)

▸ **5.** [*M25*] (A. Cayley.) Let c_n be the number of (unlabeled) oriented trees having n leaves (i.e., vertices with in-degree zero) and having at least two subtrees at every other vertex. Thus $c_3 = 2$, by virtue of the two trees

Find a formula analogous to (3) for the generating function

$$C(z) = \sum_n c_n z^n.$$

6. [*M25*] Let an "oriented binary tree" be an oriented tree in which each vertex has in-degree two or less. Find a reasonably simple relation which defines the generating function $G(z)$ for the number of distinct oriented binary trees with n vertexes, and find the first few values.

7. [*HM40*] Obtain asymptotic values for the numbers of exercise 6. (See exercise 4.)

8. [*20*] According to Eq. (9), there are six free trees with six vertices. Draw them, and indicate their centroids.

9. [*M20*] From the fact that at most one subtree of a vertex in a free tree can contain a centroid, prove there are at most two centroids in a free tree, and if there are two, then they are adjacent.

▸ **10.** [*M22*] Prove that a free tree with n vertices and two centroids consists of two free trees with $n/2$ vertices, joined by an edge. Conversely, if two free trees with m vertices are joined by an edge, we obtain a free tree with $2m$ vertices and two centroids.

▸ **11.** [*M28*] The text derives the number of different binary trees with n nodes (Eq. 13). Generalize this to find the number of different t-ary trees with n nodes. (Cf. exercise 2.3.1; a t-ary tree is either empty or consists of a root and t disjoint t-ary trees.) [*Hint:* Use Eq. (21) of Section 1.2.9.]

12. [*M20*] Find the number of labeled oriented trees with n vertices by using determinants and the result of exercise 2.3.4.2–19. (See also exercise 1.2.3–36.)

13. [*15*] What oriented tree on the vertices 1, 2, . . . , 10 has the canonical representation 3, 1, 4, 1, 5, 9, 2, 6, 5?

14. [*10*] True or false: The last entry, $f(V_{n-1})$, in the canonical representation of an oriented tree, is always the root of that tree.

15. [*21*] Discuss the relationships that exist (if any) between the topological sort algorithm of Section 2.2.3 and the canonical representation of an oriented tree.

16. [*25*] Design an algorithm (as efficient as possible) which converts from the canonical representation of an oriented tree to a conventional computer representation using "FATHER" links.

▶ **17.** [*M26*] Let $f(x)$ be an integer-valued function, where $1 \le f(x) \le m$ for all integers $1 \le x \le m$. Define $x \equiv y$ if $f^r(x) = f^s(y)$ for some $r, s \ge 0$, where $f^0(x) = x$ and $f^{r+1}(x) = f(f^r(x))$. By using methods of enumeration like those in this section, show that the number of functions such that $x \equiv y$ for all x and y is $m^{m-1}Q(m)$, where $Q(m)$ is the function defined in Section 1.2.11.3. (For related results, see exercise 3.1–14.)

18. [*24*] Show that the following method is another way to define a one-to-one correspondence between $(n-1)$-tuples of numbers from 1 to n and oriented trees with n labeled vertices: Let the leaves of the tree be $V_1, \ldots, V_k$ in ascending order. Let $(V_1, V_{k+1}, V_{k+2}, \ldots, V_\ell)$ be the path from V_1 to the root, and write down the vertices $V_\ell, \ldots, V_{k+2}, V_{k+1}$. Then let $(V_2, V_{\ell+1}, V_{\ell+2}, \ldots, V_r)$ be the shortest oriented path from V_2 such that V_r has already been written down, and write down $V_r, \ldots, V_{\ell+2}, V_{\ell+1}$. Then let $(V_3, V_{r+1}, \ldots, V_s)$ be the shortest oriented path from V_3 such that V_s has already been written, and write $V_s, \ldots, V_{r+1}$; and so on. For example, the tree (16) would be encoded as 10, 3, 3, 10, 10, 9, 9, 8, 8. Show that this process is reversible, and, in particular, draw the oriented tree with vertices 1, 2, . . . , 10 and representation 3, 1, 4, 1, 5, 9, 2, 6, 5.

19. [*M24*] How many different labeled, oriented trees are there having n vertices, k of which are leaves (i.e., have in-degree zero)?

20. [*M24*] (J. Riordan.) How many different labeled, oriented trees are there having n vertices, k_0 of which have in-degree 0, k_1 have in-degree 1, k_2 have in-degree 2, . . . ? (Note that necessarily $k_0 + k_1 + k_2 + \cdots = n$, and $k_1 + 2k_2 + 3k_3 + \cdots = n - 1$.)

▶ **21.** [*M21*] Enumerate the number of labeled oriented trees in which each vertex has in-degree zero or two. (Cf. exercise 20 and exercise 2.3–20.)

22. [*M20*] How many *labeled* free trees are possible with n vertices? (In other words, if we are given n vertices, there are $2^{\binom{n}{2}}$ possible graphs having these vertices, depending on which of the $\binom{n}{2}$ possible edges are incorporated into the graph; how many of these graphs are free trees?)

23. [*M21*] How many ordered trees are possible with n labeled vertices? (Give a simple formula involving factorials.)

24. [*M16*] All labeled oriented trees with vertices 1, 2, 3, 4 and with root 1 are shown in (14). How many would there be if we listed all labeled *ordered* trees with these vertices and this root?

25. [*M20*] What is the value of the quantity $r(n, q)$ which appears in Eqs. (17) and (18)? (Give an explicit formula; the text only mentions that $r(n, n-1) = n^{n-2}$.)

26. [*20*] In terms of the notation at the end of this section, draw the ((3, 2, 4), (2, 3, 2))-construction [analogous to (21)], and find the number k, which corresponds to the canonical representation having $t = 8$, sequences of colors "red, yellow, blue, red, yellow, blue, red, blue, blue" and "red, yellow, blue, red, yellow, blue, yellow", and sequences 3, 2; 1, 2, 1; 2, 4.

▶ **27.** [*M28*] Let $U_1, U_2, \ldots, U_p, \ldots, U_q; V_1, V_2, \ldots, V_r$ be vertices of a directed graph, where $1 \le p \le q$. Let f be any function from the set $\{p+1, \ldots, q\}$ into the set $\{1, 2, \ldots, r\}$, and let the directed graph contain exactly $q - p$ arcs, from U_k to $V_{f(k)}$ for $p < k \le q$. Show that the number of ways to add r additional arcs, one from each of the V's to one of the U's, such that the resulting directed graph contains no oriented cycles, is $q^{r-1}p$. Prove this by generalizing the canonical representation method, i.e., setting up a one-to-one correspondence between all such ways of adding r further arcs and the set of all sequences of integers $a_1, a_2, \ldots, a_r$, where $1 \le a_k \le q$ for $1 \le k < r$, and $1 \le a_r \le p$.

28. [*M22*] Use the result of exercise 27 to enumerate the number of labeled free trees on vertices $U_1, \ldots, U_m, V_1, \ldots, V_n$, such that all edges go from U_j to V_k for some j and k.

29. [*HM22*] Prove that if $E_k(r, t) = r(r + kt)^{k-1}/k!$, and if $zx^t = \ln x$, then

$$x^r = \sum_k E_k(r, t) z^k$$

for sufficiently small $|z|$ and $|x - 1|$. [Use the fact that $G_m(z) = G_1(z)^m$ in the discussion following Eq. (18).] In this formula, r stands for an arbitrary real number. [*Note:* As a consequence of this formula we have the identity

$$\sum_k E_k(r, t) E_{n-k}(s, t) = E_n(r + s, t);$$

this implies Abel's binomial theorem (Eq. 16 of Section 1.2.6). Compare also Eq. (31) of that section.]

30. [*M23*] Let $n, x, y, z_1, \ldots, z_n$ be positive integers. Consider a set of $x + y + z_1 + \cdots + z_n$ vertices r_i, s_{jk}, t_j ($1 \le i \le x + y$, $1 \le j \le n$, $1 \le k < z_j$), in which arcs have been drawn from s_{jk} to t_j for all j, k. According to exercise 27, there are $(x + y)(x + y + z_1 + \cdots + z_n)^{n-1}$ ways to draw arcs from $t_1, \ldots, t_n$ to other vertices such that the resulting directed graph contains no oriented cycles. Use this to prove Hurwitz's generalization of the binomial theorem:

$$\sum x(x + \epsilon_1 z_1 + \cdots + \epsilon_n z_n)^{\epsilon_1 + \cdots + \epsilon_n - 1} y\big(y + (1 - \epsilon_1)z_1 + \cdots + (1 - \epsilon_n)z_n\big)^{n-1-\epsilon_1-\cdots-\epsilon_n}$$
$$= (x + y)(x + y + z_1 + \cdots + z_n)^{n-1},$$

where the sum is over all 2^n choices of $\epsilon_1, \ldots, \epsilon_n$ equal to 0 or 1.

31. [*M24*] Solve exercise 5 for ordered trees; i.e., derive the generating function for the number of unlabeled ordered trees with n terminal nodes and no nodes of degree 1.

32. [*M37*] (A. Erdélyi and I. M. H. Etherington, *Edinburgh Math. Notes* **32** (1940), 7–12.) How many (ordered, unlabeled) trees are there with n_0 nodes of degree 0, n_1 of degree 1, ..., n_m of degree m, and none of higher degree with m? (An explicit

solution to this problem can be given in terms of factorials, thereby considerably generalizing the result of exercise 11.)

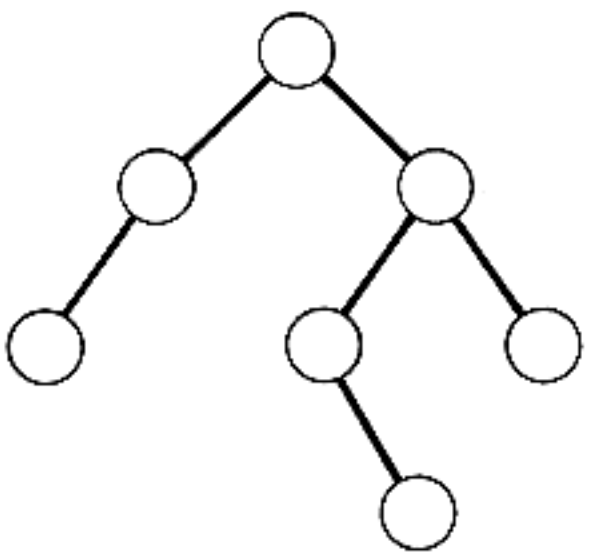
▶ **33.** [*M28*] The text gives an explicit power series solution of the equation $w = y_1 e^{z_1 w} + \cdots + y_r e^{z_r w}$, based on enumeration formulas for certain oriented forests. Similarly, show that the enumeration formula of exercise 32 leads to an explicit power series solution to the equation

$$w = z_1 w^{e_1} + z_2 w^{e_2} + \cdots + z_r w^{e_r},$$

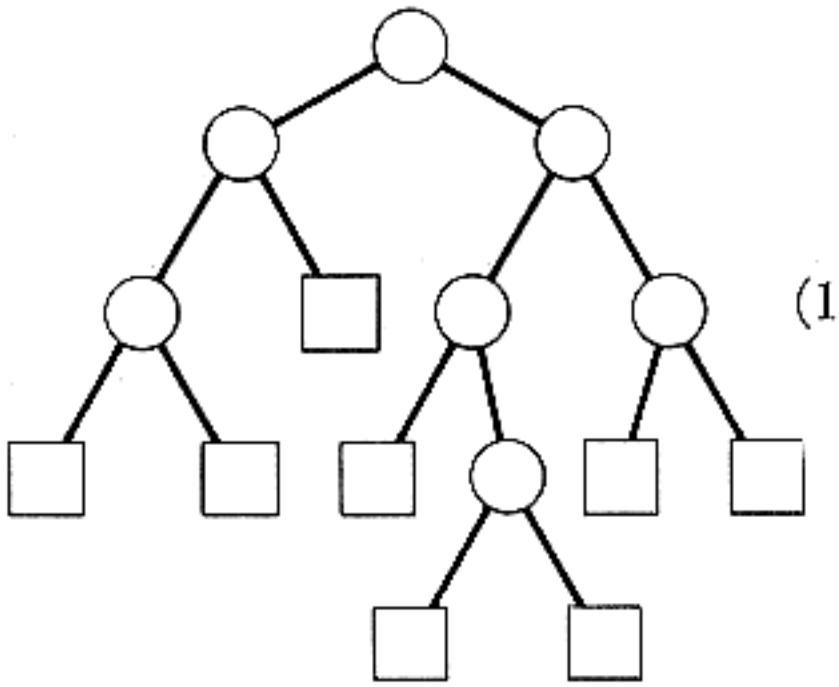
expressing w as a power series in $z_1, \ldots, z_r$. (Here $e_1, \ldots, e_r$ are fixed nonnegative integers, at least one of which is zero.)

2.3.4.5. Path length. The concept of the "path length" of a tree is of great importance in the analysis of algorithms, since this quantity is often directly related to the execution time. Our primary concern is with binary trees, since this is so close to the computer representations.

In the following discussion let us extend each binary tree diagram by adding special nodes wherever a null subtree was present in the original tree, so that



becomes



(1)

After the square-shaped nodes have been added in this way, the structure is sometimes more convenient to deal with theoretically. It is clear that in this augmented tree, every circular node has two sons and every square node has none. (Compare with exercise 2.3–20.) If there are n circular nodes and s square nodes, we have $n + s - 1$ edges (since the diagram is a free tree), and, counting another way, by the number of sons, we see there are $2n$ edges. Hence it is clear that

$$s = n + 1; \tag{2}$$

i.e., the number of "external" nodes just added is one more than the number of "internal" nodes we had originally. (For another proof, see exercise 2.3.1–14.) Formula (2) is correct even when $n = 0$.

Assume that a binary tree has been extended in this way. The *external path length of the tree*, E, is defined to be the sum—taken over all terminal (square) nodes—of the lengths of the paths from the root to each node. The *internal path length*, I, is the same quantity summed over the nonterminal (circular)

nodes. In (1) the external path length is $E = 3+3+2+3+4+4+3+3 = 25$, and the internal path length is $I = 2+1+0+2+3+1+2 = 11$. These two quantities are always related by the formula

$$E = I + 2n, \tag{3}$$

where n is the number of nonterminal nodes.

To prove formula (3), consider deleting a nonterminal node V at a distance k from the root, where both sons of V are terminal. The quantity E goes down $2(k+1)$, since the sons of V are removed, and it then goes up k, since V becomes terminal, so the net change in E is $-k-2$. The net change in I is $-k$, so (3) may be proved by induction.

It is not hard to see that the internal path length (and hence the external path length also) is highest when we have a degenerate tree with linear structure; in that case the internal path length is

$$(n-1) + (n-2) + \cdots + 1 + 0 = \tfrac{1}{2}(n^2 - n).$$

It can be shown that the "average" path length over all binary trees is essentially proportional to $n\sqrt{n}$ (see exercise 5).

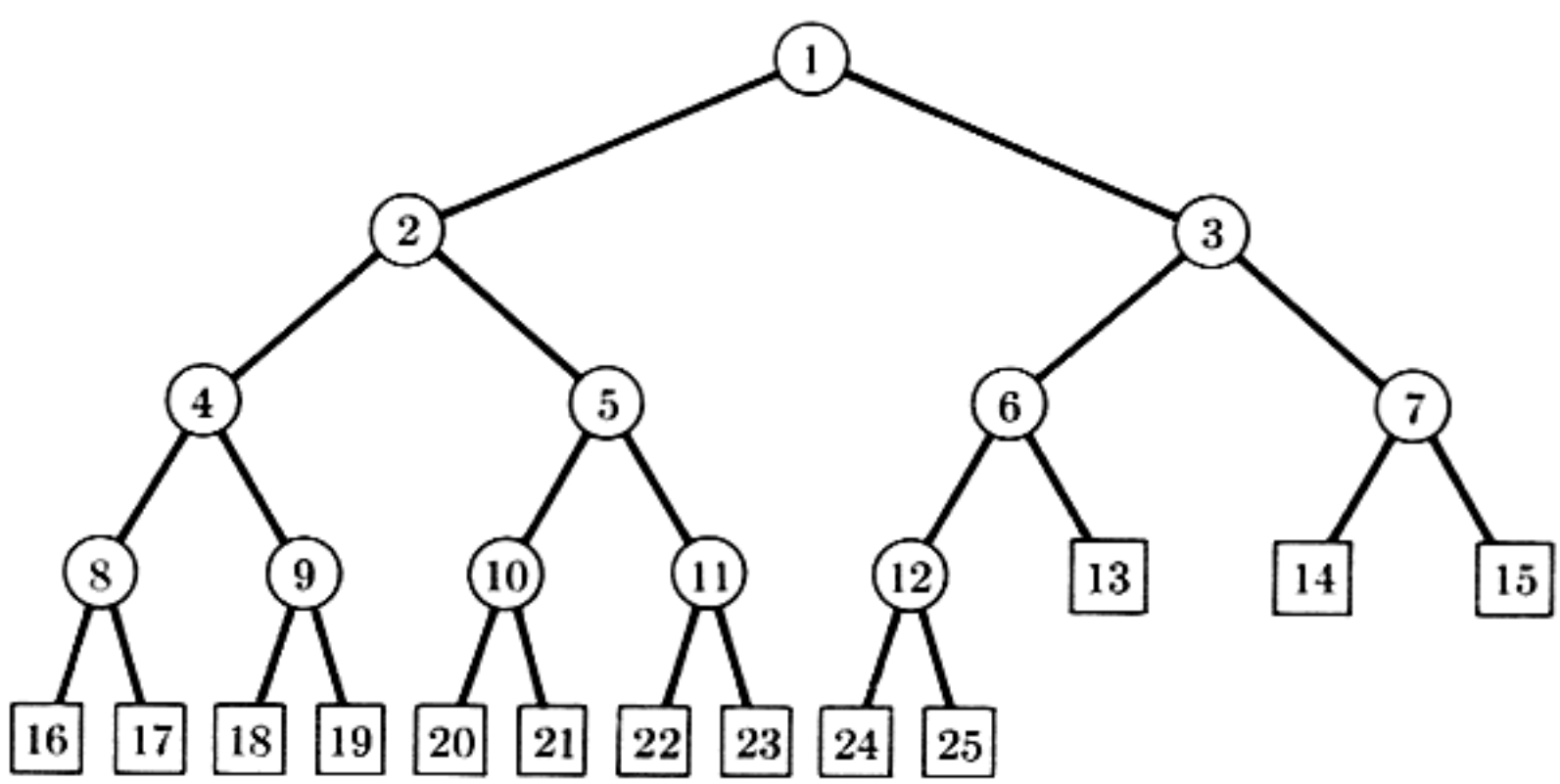
Consider now the problem of discovering a binary tree with n nodes having *minimum* path length: such a tree will be important, since it will minimize the computation time for various algorithms. Clearly, only one node (the root) can be at zero distance from the root; at most two nodes can be at distance 1 from the root, at most four can be 2 away, etc. So we see *the internal path length is always at least as big as the sum of the first n terms of the series*

$$0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, \ldots$$

This is the sum $\sum_{1\le k\le n} \lfloor \log_2 k \rfloor$, which we know from exercise 1.2.4–42 is

$$(n+1)q - 2^{q+1} + 2, \qquad q = \lfloor \log_2 (n+1) \rfloor. \tag{4}$$

Thus the optimum value (4) is essentially of the form $n \log n$; this optimum is clearly achieved in a tree which looks essentially like this (illustrated for $n = 12$):

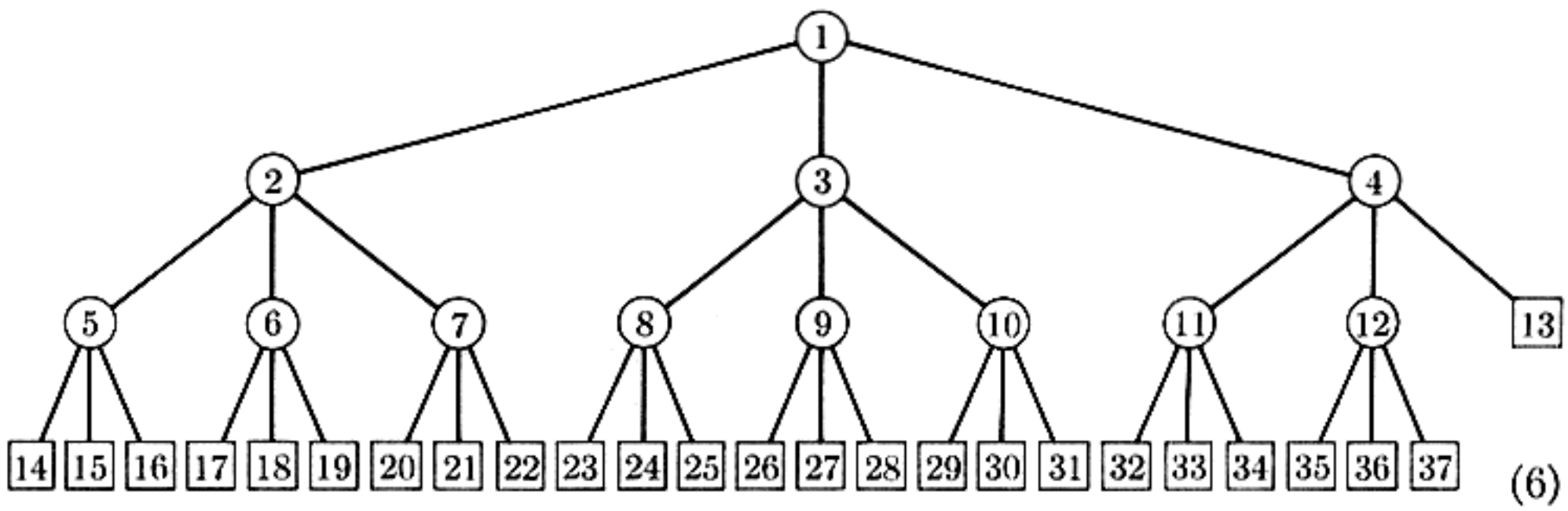


(5)

A tree such as (5) is called a *complete binary tree* with n internal nodes. In the general case we may number the nodes $1, 2, \ldots, n$; this numbering has the useful property that the father of node k is node $\lfloor k/2 \rfloor$; the sons of node k are nodes $2k$ and $2k+1$. The terminal nodes are numbers $n+1$ through $2n+1$, inclusive.

It follows that a complete binary tree may be simply represented in sequential memory locations, with the structure implicit in the locations of the nodes. The complete binary tree appears explicitly or implicitly in many important computer algorithms, so the reader should give it special attention.

These concepts have important generalizations to ternary, quaternary, etc. trees. We define a *t-ary tree* as a set of nodes which is either empty or consists of a root and t ordered, disjoint t-ary trees. (Cf. the definition of binary tree in Section 2.3.) The *complete ternary tree* with 12 internal nodes is



(6)

It is easy to see how this generalizes to the complete t-ary tree with the internal nodes $1, 2, \ldots, n$: the father of node k is node

$$\lfloor (k+t-2)/t \rfloor = \lceil (k-1)/t \rceil,$$

and the sons of node k are

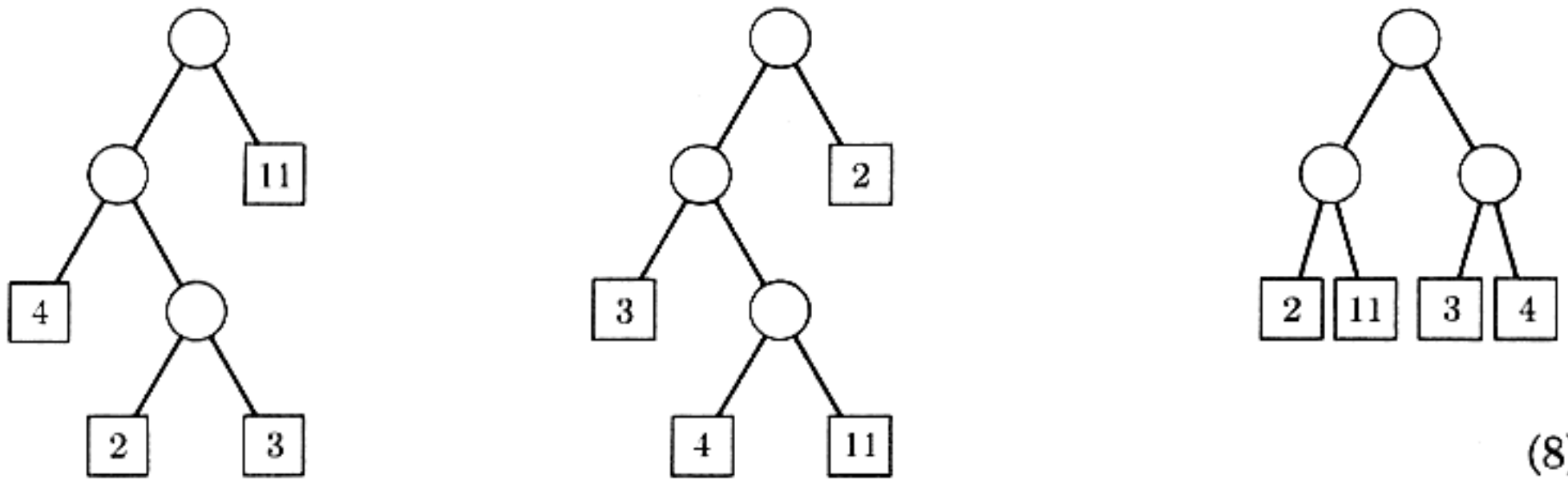
$$t(k-1)+2, \qquad t(k-1)+3, \qquad \ldots, \qquad tk+1.$$

This tree has the minimum internal path length among all t-ary trees with n internal nodes: its internal path length is (see exercise 8)

$$\left(n + \frac{1}{t-1}\right) q - \frac{(t^{q+1} - t)}{(t-1)^2}, \qquad q = \lfloor \log_t ((t-1)n+1) \rfloor. \tag{7}$$

These results have another important generalization if we shift our point of view slightly. Suppose that we are given m real numbers $w_1, w_2, \ldots, w_m$; the problem is to find an extended binary tree with m terminal nodes, and to associate the numbers $w_1, \ldots, w_m$ with these nodes, in such a way that the sum $\sum w_j l_j$ is minimized, where l_j is the length of path from the root and the sum is taken over all terminal nodes. For example, if the given numbers are 2, 3, 4, 11,

we can form extended binary trees such as these three:

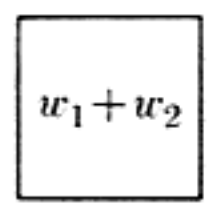


(8)

Here the "weighted" path lengths $\sum w_j l_j$ are 34, 53, and 40, respectively. (Note that a perfectly balanced tree does *not* give the minimum weighted path length when the weights are 2, 3, 4, and 11, although we have seen that it does give the minimum in the special case $w_1 = w_2 = \cdots = w_m = 1$.)

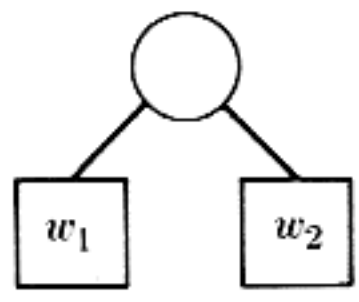
There are several interpretations of weighted path length in connection with different computer algorithms; for example, we can apply it to the merging of sorted strings of respective lengths $w_1, w_2, \ldots, w_m$ (see Chapter 5). One of the most straightforward applications of this idea is to consider a binary tree as a general search procedure, where we start at the root and then make some test; the outcome of the test sends us to one of the two branches, where we may make further tests, etc. For example, if we want to decide which of four different alternatives is true, and if these possibilities will be true with the respective probabilities $\frac{2}{20}$, $\frac{3}{20}$, $\frac{4}{20}$, and $\frac{11}{20}$, the tree which minimizes the weighted path length constitutes an *optimal search procedure* in this case. [These are the weights shown in (8).]

An elegant algorithm for finding a tree with minimum weighted path length has been given by D. Huffman: First find the two w's of lowest value, say w_1 and w_2. Then solve the problem for $m - 1$ weights $w_1 + w_2, w_3, \ldots, w_m$, and replace the node



(9)

in this solution by



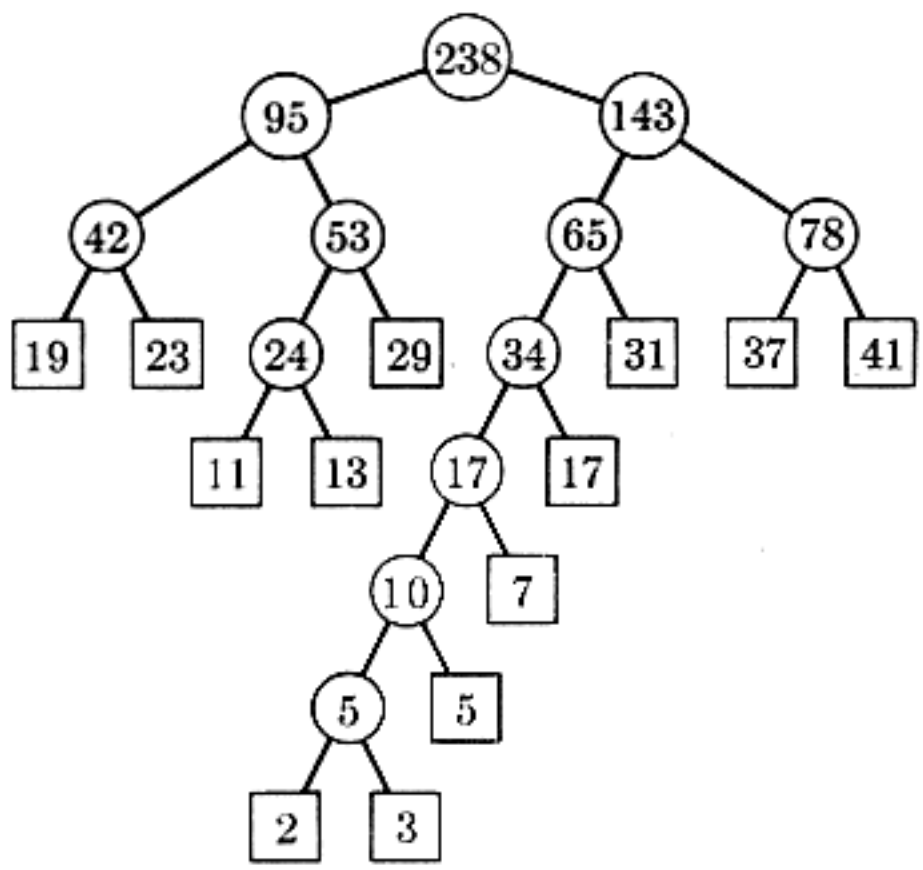
(10)

As an example of Huffman's method, let us find the optimal tree for the weights 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41. First we combine $2 + 3$, and look for the solution to $5, 5, 7, \ldots, 41$; then we combine $5 + 5$, etc. The

computation is summarized as follows:

2	3	5	7	11	13	17	19	23	29	31	37	41
	5	5	7	11	13	17	19	23	29	31	37	41
		10	7	11	13	17	19	23	29	31	37	41
			17	11	13	17	19	23	29	31	37	41
			17		24	17	19	23	29	31	37	41
					24	34	19	23	29	31	37	41
					24	34		42	29	31	37	41
						34		42	53	31	37	41
								42	53	65	37	41
								42	53	65		78
									95	65		78
									95			143
												238

Therefore the following tree corresponds to Huffman's construction:



(11)

(The numbers inside the circular nodes show the correspondence between this tree and our computation; see also exercise 9.)

It is not hard to show that this method does in fact minimize the weighted path length, by induction on m. Suppose that $m \geq 2$ and $w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_m$, and suppose that we are given a tree which minimizes the weighted path length. (Such a tree certainly exists, since only finitely many binary trees with m terminal nodes are possible.) Let V be an internal node of maximum distance from the root. If w_1 and w_2 are not the weights already attached to the sons of V, we can interchange them with the values which are already there and not increase the weighted path length. Thus there is a tree which minimizes the weighted path length and which contains the subtree (10). Now it is easy to prove that the weighted path length of such a tree is minimized if and only if the tree with (10) replaced by (9) has minimum path length for the weights $w_1 + w_2, w_3, \ldots, w_m$. (See exercise 9.)

In general, there are many trees which minimize $\sum w_j l_j$. If the w's are kept in order throughout the construction, and if when w_1, w_2 are removed the quantity $w_1 + w_2$ is placed higher in the ordering than any of the other weights of the same value (i.e., between w_k and w_{k+1}, where $w_k \le w_1 + w_2 < w_{k+1}$), then the tree constructed by Huffman's method has the smallest value of max l_j and of $\sum l_j$ among all trees which minimize $\sum w_j l_j$. [See the article by Eugene S. Schwartz, *Information and Control* 7 (1964), 37–44.]

Huffman's method can be generalized to t-ary trees as well as binary trees. (See exercise 10.)

EXERCISES

1. [*12*] Are there any other binary trees with 12 internal nodes and minimum path length, besides the complete binary tree (5)?

2. [*17*] Draw an extended binary tree with terminal nodes containing the weights 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, having minimum weighted path length.

▶ 3. [*M24*] An extended binary tree with m terminal nodes determines a set of path lengths $l_1, l_2, \ldots, l_m$ which describe the length of path from the root to the respective terminal nodes. Conversely, if we are given a set of numbers $l_1, l_2, \ldots, l_m$, is it always possible to construct an extended binary tree in which these numbers are the path lengths in some order? Show that this is possible if and only if $\sum_{1 \le j \le m} 2^{-l_j} = 1$.

▶ 4. [*M25*] (E. S. Schwartz.) Assume that $w_1 \le w_2 \le \cdots \le w_m$. Show that there is an extended binary tree which minimizes $\sum w_j l_j$ and for which the terminal nodes in left to right order contain the respective values $w_1, w_2, \ldots, w_m$. [For example, tree (11) does *not* meet this condition since the weights appear in the order 19, 23, 11, 13, 29, 2, 3, 5, 7, 17, 31, 37, 41. We seek a tree for which the weights appear in ascending order, and this does not always happen with Huffman's construction.]

5. [*HM26*] Let

$$B(w, z) = \sum_{n,p \ge 0} b_{np} w^p z^n,$$

where b_{np} is the number of binary trees with n nodes and internal path length p. [Thus,

$$B(w, z) = 1 + z + 2wz^2 + (w^2 + 4w^3)z^3 + (4w^4 + 2w^5 + 8w^6)z^4 + \cdots;$$

$B(1, z)$ is the function $B(z)$ of Eq. (12) in Section 2.3.4.4.] (a) Find a functional relation which characterizes $B(w, z)$. (b) Use the result of (a) to determine the *average internal path length* of a binary tree with n nodes, assuming that each of the

$$\frac{1}{n+1}\binom{2n}{n}$$

trees is equally probable. (c) Find the asymptotic value of this quantity.

6. [*16*] If a t-ary tree is extended with "square" nodes as in (1), what is the relation between the number of square and circular nodes corresponding to Eq. (2)?

7. [*M21*] What is the relation between external and internal path length in a t-ary tree? (Cf. exercise 6; a generalization of Eq. (3) is desired.)

8. [*M23*] Prove Eq. (7).

9. [*M21*] The numbers which appear in the circular nodes of (11) are equal to the sums of the weights in the terminal nodes of the corresponding subtree. Show that the sum of all values in the circular nodes is equal to the weighted path length.

▸ **10.** [*M24*] (D. Huffman.) Show how to construct a t-ary tree with minimum weighted path length, given weights $w_1, w_2, \ldots, w_m$. Construct an optimal ternary tree for weights 1, 4, 9, 16, 25, 36, 49, 64, 81, 100.

11. [*16*] Is there any connection between the complete binary tree (5) and the "Dewey decimal notation" for binary trees described in exercise 2.3.1–5?

2.3.4.6. History and bibliography. Trees have of course been in existence since the third day of creation and perhaps earlier, and through the ages tree structures (especially *family* trees) have been in common use. The concept of tree as a formally defined *mathematical* entity seems to have appeared first in the work of G. Kirchhoff [*Annalen der Physik und Chemie* **72** (1847), 497–508], who used trees to find a set of fundamental cycles in an electrical network in connection with the law that bears his name, essentially as we did in Section 2.3.4.1. The concept also appeared at about the same time in the book *Geometrie der Lage* (pp. 20–21) by K. G. Chr. von Staudt. The name "tree" and many results dealing mostly with enumeration of trees began to appear ten years later in a series of papers by Arthur Cayley [see *Collected Mathematical Papers of A. Cayley* **3** (1857), 242–246; **4** (1859) 114–115; **9** (1874), 202–204; **9** (1875), 427–460; **10** (1877), 598–600; **11** (1881), 365–367; **13** (1889), 26–28]. Cayley was unaware of the previous work of Kirchhoff and von Staudt; his investigations began with studies of the structure of algebraic formulas, and they were later inspired chiefly by applications to the problem of isomers in chemistry. Tree structures were also independently studied by J. B. Listing [*Göttingen Abhandlungen* **10** (1862)] and C. Jordan [*Journal f. d. reine und angewandte Math.* **70** (1869).

The infinity lemma was formulated first by Denes König [*Fundamenta Mathematicae* **8** (1926), 114–134], and he gave it a prominent place in his classic book *Theorie der endlichen und unendlichen Graphen* (Leipzig, 1936), Chapter 6. A similar result called the "fan theorem" occurred slightly earlier in the work of L. E. J. Brouwer [*Verhandelingen Akad. Amsterdam* **12** (1919), 7], but this involved much stronger hypotheses; for a discussion of Brouwer's work see A. Heyting, *Intuitionism* (1956), Section 3.4.

Formula (3) of Section 2.3.4.4 for enumerating unlabeled oriented trees was given by Cayley in his first paper on trees. In his second paper he enumerated unlabeled ordered trees; an equivalent problem had already been proposed and solved by J. de Segner and L. Euler 100 years earlier (*Novi Commentarii Academiae Scientiarum Petropolitanae* **7** (1760), 13–15, 203–209), and it was the subject of seven papers by G. Lamé, E. Catalan, O. Rodrigues, and J. Binet in *Journal de Mathématiques* **3, 4** (1838, 1839).

Cayley published the formula n^{n-2} as the number of *labeled* free trees 32 years after his first paper on the subject, and at that time he hinted at the connection between trees and $(n-1)$-tuples of numbers. (Cayley's discussion in that paper is extremely vague and can hardly be considered a proof of the formula n^{n-2}, since he merely worked out a few special cases and asserted that the same procedure would work in general, without really saying what the procedure was. There is, of course, little doubt that Cayley could have proved the formula if he had been pressed to do so.) The explicit correspondence between trees and $(n-1)$-tuples of numbers was first published by Heinz Prüfer [*Arch. Math. u. Phys.* **27** (1918), 142–144], quite independently of Cayley's prior work. An interesting bibliography, which lists over 25 papers having to do with the formula n^{n-2} and its generalizations, has been compiled by J. W. Moon, in *A Seminar on Graph Theory* (F. Harary, ed.), Chapter 11, (New York: Holt, Rinehart, and Winston, 1967), 70–78.

A very important paper on the enumeration of trees and many other kinds of enumeration problems was published by G. Polya in *Acta Math.* **68** (1937), 145–253. For discussion of enumeration problems for graphs and an excellent bibliography see the survey paper, "Unsolved Problems in the Enumeration of Graphs" by F. Harary, *Matematikai Kutató Intézetének Közleményei* (Publ. of the Math. Institute), Hungarian Academy of Sciences, **A5** (1960), 63–95.

The principle of minimizing weighted path length by repeatedly combining the smallest weights was discovered by D. Huffman [*Proc. IRE* **40** (1952), 1098–1101], in connection with the design of codes for minimizing message lengths. The same idea was independently published by Seth Zimmerman [*AMM* **66** (1959), 690–693].

Several noteworthy recent papers dealing with the theory of tree structures have been cited in Sections 2.3.4.1 through 2.3.4.5 in connection with particular topics.

For further discussion of the mathematical properties of trees, see the following references and their bibliographies:

CLAUDE BERGE, *The Theory of Graphs*, tr. by Alison Doig (London: Methuen, 1962), Chapter 16–17.

ØYSTEIN ORE, *Theory of Graphs* (Amer. Math Society, 1962), Chapter 4.

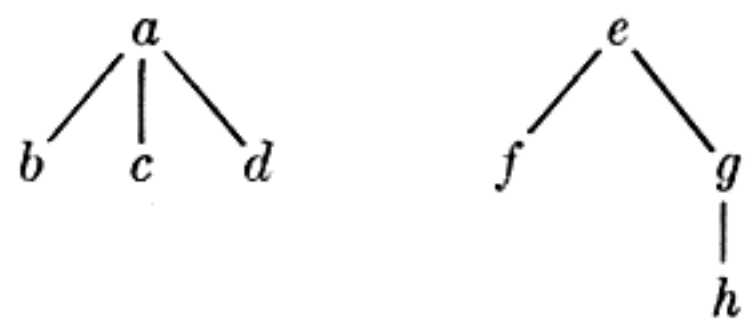
JOHN RIORDAN, *Introduction to Combinatorial Analysis* (New York: Wiley, 1958), Chapter 6.

The theoretical study of generalizations of graphs, to structures involving nodes containing several information and link fields, is still in its infancy, and it promises to be important in the future.

2.3.5. Lists and Garbage Collection

Near the beginning of Section 2.3 we defined a List as "a finite sequence of zero or more atoms or Lists."

Any forest is a List; for example,

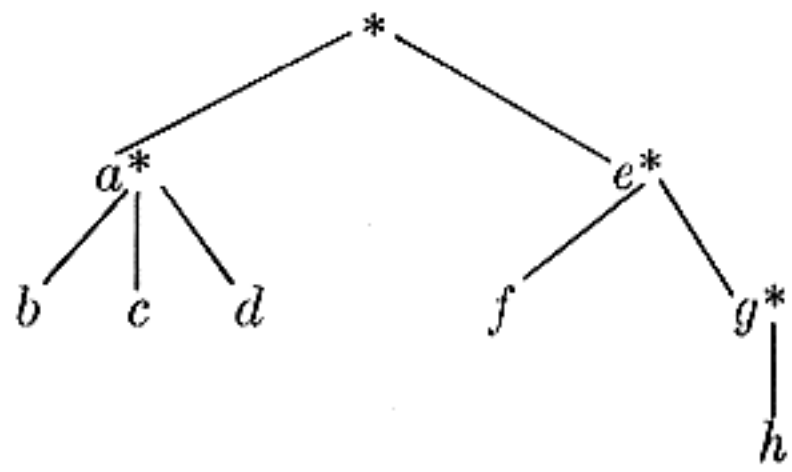


(1)

may be regarded as the List

$$(a:(b, c, d), e:(f, g:(h)), \tag{2}$$

and the corresponding List diagram would be



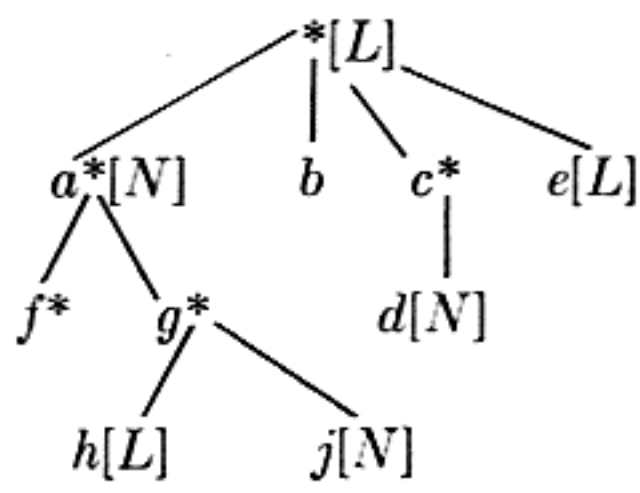
(3)

The reader should review at this point the introduction to Lists given earlier, in particular (3), (4), (5), (6), (7) in the beginning of Section 2.3. Recall that the notation "a:" which appears in (2) above means that the List (b, c, d) is "labeled" with the attribute "a" besides its structural information that it is a List of three atoms b, c, and d. This is compatible with our general convention that each node of a tree may contain information besides its structural connections. However, as was discussed for trees in Section 2.3.3, it is quite possible and sometimes desirable to insist that all Lists be unlabeled, so that all the information appears in the atoms.

Although any forest may be regarded as a List, the converse is not true. The following List is perhaps more typical than (2) and (3) since it shows how the restrictions of tree structure may be violated:

$$L = (a:N, b, c:(d:N), e:L), \qquad N = (f:(), g:(h:L, j:N)) \tag{4}$$

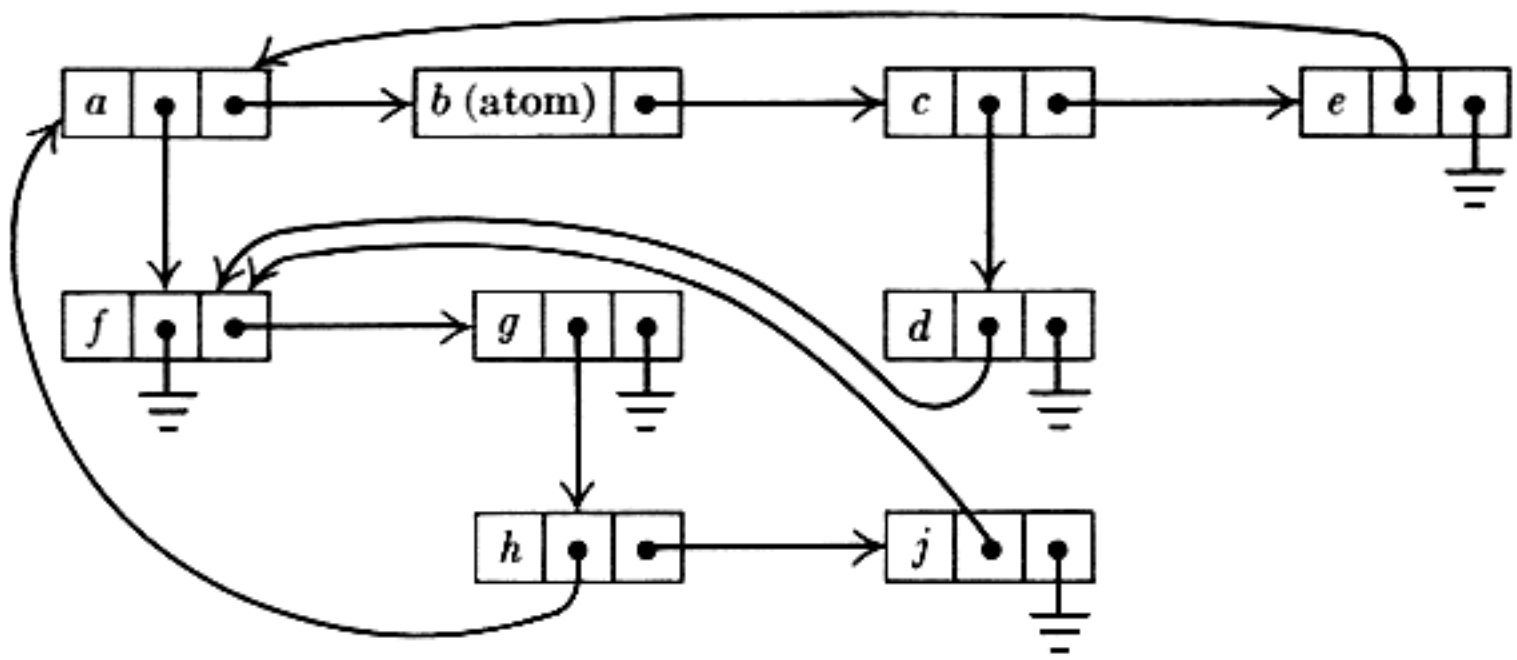
which may be diagramed as



(5)

[Cf. diagram (7) in Section 2.3.]

As we might expect, there are many ways to represent List structures within a computer memory. These are usually variations on the same basic theme according to which binary trees are used to represent general forests of trees: one field RLINK, say, is used to point to the next element of a List, and another field DLINK may be used to point to the first element of a sub-List. By a natural extension of the memory representation described in Section 2.3.2, we would represent the List (5) as follows:

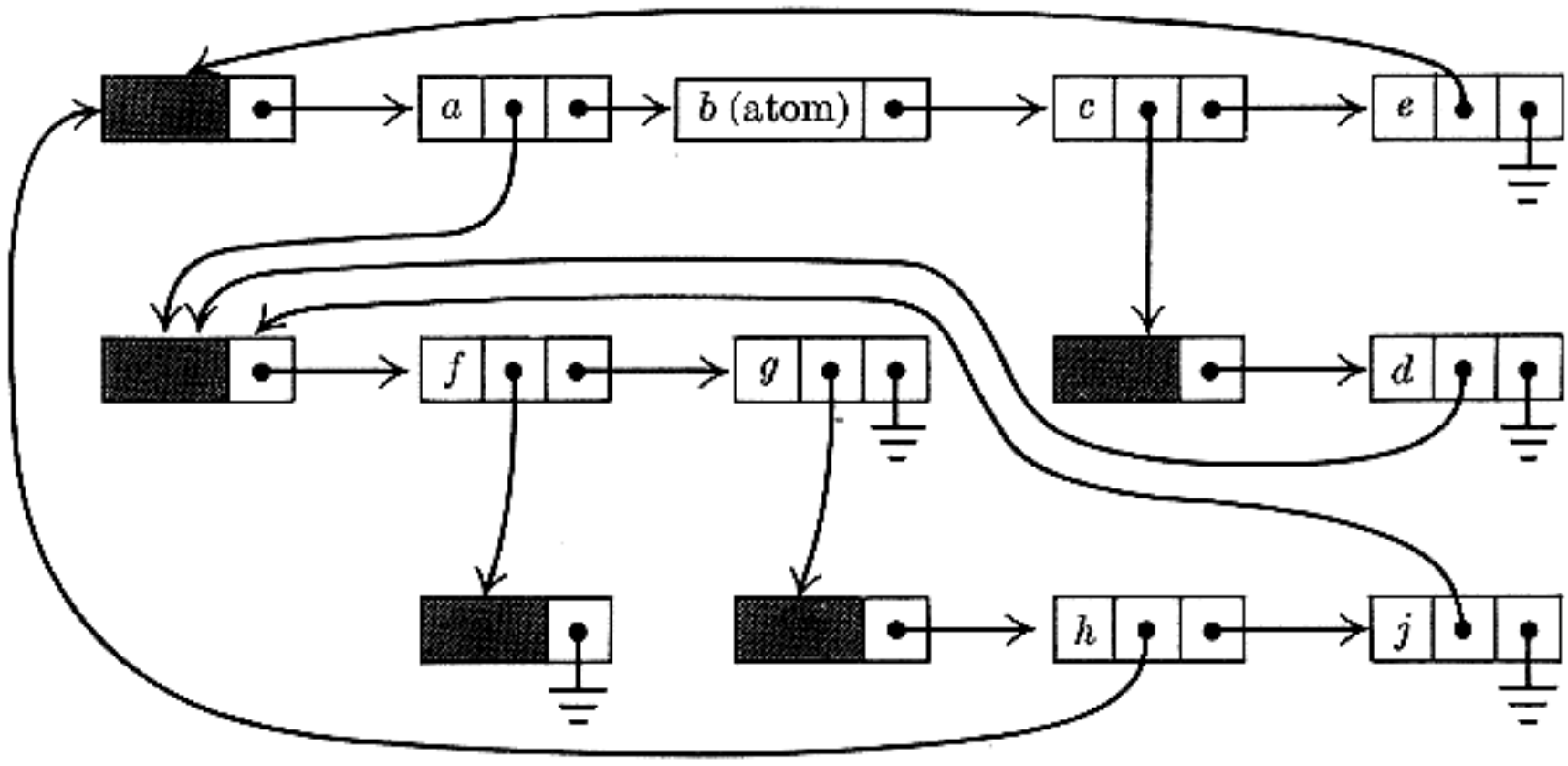


(6)

Unfortunately, this simple idea is *not* quite adequate for the most common List processing applications. For example, suppose that we have the List $L = (A, a, (A, A))$, which contains three references to another List $A = (b, c, d)$. One of the typical List processing operations is to remove the leftmost element of A, so that A becomes (c, d); but this requires *three* changes to the representation of L, if we are to use the technique shown in (6), since each pointer to A points to the element b that is being deleted. A moment's reflection will convince the reader that it is extremely undesirable to change the pointers in every reference to A just because the first element of A is being deleted. (*Note:* In this example we could try to be tricky, assuming that there are no pointers to the element c, by moving the entire element c into the location formerly occupied by b and then deleting the element c. But this trick fails to work when A loses its last element and becomes empty.)

For this reason the representation scheme (6) is generally replaced by another scheme which is similar, but uses a *List head* to begin each List, as was introduced in Section 2.2.4. Each List contains an additional node called its List head, so that the configuration (6) would, for example, be represented as shown in diagram (7) at the top of the next page.

The introduction of these header nodes is not really a waste of memory space in practice, since many uses for the apparently unused fields (which are shaded areas in diagram (7)) generally present themselves. For example, there is room for a reference count, or a pointer to the right end of the List, or an alphabetic name, or a "scratch" field which aids traversal algorithms, etc.



(7)

Note that in our original diagram (6), the node containing b is an atom while the node containing f specifies an empty List. These two things are structurally identical, and so the reader would be quite justified in asking why we bother to talk about "atoms" at all; with no loss of generality we could have defined Lists as merely "a finite sequence of zero or more Lists," with our usual convention that each node of a List may contain data besides its structural information. This point of view is certainly defensible and it makes the concept of an "atom" seem very artificial. There is, however, a good reason for singling out atoms as we have done, when efficient use of computer memory is taken into consideration, since atoms are not subject to the same sort of general-purpose manipulation that is desired for Lists. The memory representation (6) shows there is probably more room for information in an atomic node, b, than in a List node, f; and when List head nodes are also present as in (7), there is a dramatic difference between the storage requirements for the nodes b and f. Thus the concept of atoms is introduced primarily to aid in the effective use of computer memory. [Typical Lists contain many more atoms than our example would indicate; the example (4)–(7) is intended to show the complexities that are possible, not the simplicities that are usual.]

A List is in essence nothing more than a linear list whose elements may contain pointers to other Lists. The common operations we wish to perform on Lists are the usual ones desired for linear lists (creation, destruction, insertion, deletion, splitting, concatenation), plus further operations which are primarily of interest for tree structures (copying, traversal, input and output of nested information). For these purposes any of the three basic techniques for representing linked linear lists in memory—namely straight, circular, or double linkage—can be used, with varying degrees of efficiency depending on the algorithms being employed. For these three types of representation, diagram (7) might appear in memory as listed in (8) at the top of the next page.

Memory location	*Straight linkage* INFO	DLINK	RLINK	*Circular linkage* INFO	DLINK	RLINK	*Double linkage* INFO	DLINK	LLINK	RLINK
010:	—	head	020	—	head	020	—	head	050	020
020:	*a*	060	030	*a*	060	030	*a*	060	010	030
030:	*b*	atom	040	*b*	atom	040	*b*	atom	020	040
040:	*c*	090	050	*c*	090	050	*c*	090	030	050
050:	*e*	010	Λ	*e*	010	010	*e*	010	040	010
060:	—	head	070	—	head	070	—	head	080	070
070:	*f*	110	080	*f*	110	080	*f*	110	060	080
080:	*g*	120	Λ	*g*	120	060	*g*	120	070	060
090:	—	head	100	—	head	100	—	head	100	100
100:	*d*	060	Λ	*d*	060	090	*d*	060	090	090
110:	—	head	Λ	—	head	110	—	head	110	110
120:	—	head	130	—	head	130	—	head	140	130
130:	*h*	010	140	*h*	010	140	*h*	010	120	140
140:	*j*	060	Λ	*j*	060	120	*j*	060	130	120

(8)

Here "LLINK" is used for a pointer to the left in a doubly linked representation. Note that the INFO and DLINK fields are identical in all three forms.

There is no need to repeat here the algorithms for List manipulation in any of these three forms, since the ideas are identical to those we have already seen many times in this chapter. The following important points about Lists, which distinguish them from the simpler special cases treated earlier, should be noted, however:

1) It is implicit in the above memory representation that atomic nodes are distinguishable from nonatomic nodes; furthermore, when circular or doubly linked lists are being used, it is desirable to distinguish header nodes from the other types, as an aid in traversing the Lists. Therefore each node generally contains a TYPE field which tells what kind of information the node represents. This TYPE field is often used also to distinguish between various types of atoms (e.g., between alphabetic, integer, or floating-point data, for use when printing or displaying answers).

2) The following are two examples of possible ways to design the format of nodes for general List manipulation with the MIX computer:

a) Possible one-word format, assuming all INFO appears in atoms:

S	T	REF	RLINK

(9)

S (sign): "mark bit" used in "garbage collection" (see below).

T (type): T = 0 for List head; T = 1 for sub-List element; T > 1 for atoms.

REF: When T = 0, REF is a "reference count" (see below); when T = 1, REF points to the List head of the sub-List in question; when T > 1, REF points to a node containing a mark bit and five bytes of atomic information.

RLINK: Pointer for straight or circular linkage as in (8).

b) Possible two-word format:

S	T	LLINK	RLINK
INFO			

(10)

S,T: As in (9).

LLINK,RLINK: Usual pointers for double linkage as in (8).

INFO: Full word of information associated with this node; for a header node this may include a reference count, a running pointer to the interior of the List to facilitate linear traversal, an alphabetic name, etc.

3) It is clear that Lists are very general structures; indeed, it seems fair to state that any structure whatsoever can be represented as a List when appropriate conventions are made. Because of this universality of Lists, a large number of programming systems have been designed to facilitate List manipulation, and there are usually several such systems available at any computer installation. These systems are based on some general-purpose format for nodes such as (9) or (10) above, designed for flexibility in List operations. Actually, it is clear that this general-purpose format is usually not the best format suited to a *particular* application, and the processing time using the general-purpose routines is noticeably slower than a person would achieve by hand-tailoring the system to his particular problem. For example, it is easy to see that nearly all of the applications we have worked out so far in this chapter would be encumbered by a general-List representation as in (9) or (10) instead of the node format that was given in each case. A List manipulation routine must often examine the T-field when it processes nodes, and this was not needed in any of our programs so far. This loss of efficiency is repaid in many instances by the comparative ease of programming and the reduction of debugging time that are obtained with the general-purpose system.

4) There is one extremely significant difference between algorithms for List processing and the algorithms given previously in this chapter. Since a single List may be contained in many other Lists, it is by no means clear exactly when a List should be returned to the pool of available storage. Our algorithms so far have always said "AVAIL $\Leftarrow$ X", whenever NODE(X) was no longer needed. But since general Lists can grow and die in such an unpredictable manner while a program runs, it is often quite difficult to tell just when a particular node is superfluous. Therefore the problem of maintaining the list of available space is considerably more difficult with Lists than in the simple cases considered previously. We will devote the rest of this section to a discussion of this problem.

Let us imagine we are designing a general-purpose List processing system that will be used by hundreds of other programmers. Two principal methods

have been suggested for maintaining the available space list: the use of *reference counters*, and *garbage collection*. The reference-counter technique makes use of a new field in each node, which contains a count of how many arrows point to this node. Such a count is rather easy to maintain as a program runs, and whenever it drops to zero, the node in question becomes available. The garbage-collection technique requires a new one-bit field in each node called the "mark bit." The idea in this case is to write nearly all the algorithms so that they do not return any nodes to free storage, and to let the program run merrily along until all of the available storage is gone; then a "garbage-collector" algorithm makes use of the mark bits to return to available storage all nodes that are not currently accessible, and the program continues.

Neither of these two methods is completely satisfactory. The principal drawback of the reference-counter method is the fact that it does not always free all the nodes that are available. It works fine for overlapped Lists, but recursive Lists, like our examples L and N in (4), will *never* be returned to storage by the reference counter technique; their counts will be nonzero (since they refer to themselves) even when no other List accessible to the running program points to them. Furthermore, the reference-counter method uses a good chunk of space in each node (although sometimes this space is available anyway due to the computer word size).

The difficulty with the garbage-collection technique, besides the annoying loss of a bit in each node, is that it runs very slowly when nearly all the memory space is in use; and in such cases the number of free storage cells found by the reclamation process is not worth the effort. Those programs which exceed the capacity of storage (and many undebugged programs do!) often waste a good deal of time calling the garbage collector several, almost fruitless, times just before storage is finally exhausted. A partial solution to this problem is to let the programmer specify a number k, such that he does not wish to continue processing after a garbage collection run has found k or fewer free nodes. A further problem is the occasional difficulty of determining exactly what Lists are not garbage at a given stage; if the programmer has been using any nonstandard techniques or keeping any pointer values in unusual places, chances are good that the garbage collector will go awry. Some of the greatest mysteries in the history of computer program debugging have been caused by the fact that garbage collection suddenly took place at an unexpected time during the running of programs that had worked many times before. Garbage collection also requires that programmers keep valid information in all pointer fields at all times, although it is often convenient to leave meaningless information in fields which are never referred to by the program (for example, the link in the rear node of a queue, see exercise 2.2.3–6). We might also note that garbage collection is unsuitable for "real-time" applications, because even if the garbage collector goes into action infrequently, it requires large amounts of computer time on these occasions. (However, see exercise 12.)

Although garbage collection requires one mark bit for each node, it is possible to keep a separate table of all the mark bits packed together in another memory area, with a suitable correspondence between the location of a node and its mark bit. On some computers this idea can lead to a method of handling garbage collection which is more attractive than giving up a bit in each node, but on many other computers it slows down the process of garbage collection quite a bit.

J. Weizenbaum has suggested an interesting modification of the reference-counter technique. Using doubly linked List structures, he puts a reference counter only in the header of each List. Thus, when pointer variables traverse a List, they are not included in the reference counts for the individual nodes; but since the programmer knows the rules by which reference counts are maintained for entire Lists, he knows (in theory) how to avoid referring to any List that has a reference count of zero. The programmer also has the ability to explicitly override reference counts and to return certain Lists to available storage. These ideas require the programmer to exercise caution; they prove to be somewhat dangerous in the hands of inexperienced programmers and have tended to make program debugging more difficult due to the consequences of referring to nodes that have been erased. The nicest part of Weizenbaum's approach is his treatment of Lists whose reference count has just gone to zero: such a List is appended at the *end* of the current available space list—this is easy to do with doubly linked lists—and it is considered for available space only after all previously available cells are used up; then as the individual nodes of this List do become available, the reference counters of Lists *they* refer to are decreased by one. This delayed action of erasing the Lists is quite efficient with respect to running time; but it tends to make incorrect programs run correctly for awhile! For further details see *CACM* **6** (1963), 524–544.

Algorithms for garbage collection are quite interesting for several reasons. In the first place, such an algorithm is useful in other situations when we want to "mark all nodes directly or indirectly referred to by a given node." (For example, we might want to find all subroutines called directly or indirectly by a certain subroutine; cf. exercise 2.2.3–26. See also the ancestor algorithm in Chapter 7.)

Garbage collection generally proceeds in two phases. We assume that the mark bits of all nodes are initially zero (or we set them all to zero). Now the first phase marks all the nongarbage nodes, starting from those which are immediately accessible to the main program. The second phase makes a sequential pass over the entire memory pool area, putting all unmarked nodes onto the list of free space. The marking phase is the most interesting, and so we will concentrate our attention on it. There are variations on the second phase which make it nontrivial; see exercise 9.

The most interesting feature of garbage collection is the fact that while this algorithm is running, *there is only a very limited amount of storage available which*

we can use to control our marking algorithm. This intriguing problem will become clear in the following discussion; it is a difficulty which is not appreciated by most people when they first hear about the idea of garbage collection, and for many years there was no good solution to it.

The following marking algorithm is perhaps the most obvious:

Algorithm A (*Marking*). Let the entire memory used for List storage be `NODE(1)`, `NODE(2)`, . . . , `NODE(M)`, and suppose that these words either are "atoms" or contain two link fields `REF` and `RLINK`. Assume that all nodes are initially *unmarked*. The purpose of this algorithm is to *mark* all of the nodes which can be reached by a chain of `REF` and/or `RLINK` pointers in nonatomic nodes, starting from a set of "immediately accessible" nodes.

A1. [Initialize.] Mark all nodes that are "immediately accessible," i.e., the nodes pointed to by certain fixed locations in the main program which are used as a source for all memory accesses. Set `K` $\leftarrow 1$.

A2. [Does `NODE(K)` imply another?] Set `K1` $\leftarrow$ `K` $+ 1$. If `NODE(K)` is an atom or unmarked, go to step A3. Otherwise, if `NODE(REF(K))` is unmarked, mark it, and if it is not an atom, set `K1` $\leftarrow$ min (`K1`, `REF(K)`). Similarly, if `NODE(RLINK(K))` is unmarked, mark it, and if it is not an atom, set `K1` $\leftarrow$ min (`K1`, `RLINK(K)`).

A3. [Done?] Set `K` $\leftarrow$ `K1`. If `K` $\leq$ `M`, return to step A2; otherwise the algorithm terminates. ▮

Throughout this algorithm and the ones which follow in this section, we will assume for convenience that the nonexistent node "`NODE(Λ)`" *is "marked."* (For example, `REF(K)` or `RLINK(K)` may equal Λ in step A2.)

A variant of Algorithm A sets `K1` $\leftarrow$ `M` $+ 1$ in step A1, removes the operation "`K1` $\leftarrow$ `K` $+ 1$" from step A2, and instead changes step A3 to

"**A3′.** [Done?] Set `K` $\leftarrow$ `K` $+ 1$. If `K` $\leq$ `M`, return to step A2. Otherwise if `K1` $\leq$ `M`, set `K` $\leftarrow$ `K1` and `K1` $\leftarrow$ `M` $+ 1$ and return to step A2. Otherwise the algorithm terminates."

It is very difficult to give a precise analysis of Algorithm A, or to determine whether it is better or worse than the variant just described, since no meaningful way to describe the probability distribution of the input presents itself. We can say it takes up time proportional to n`M` in the worst case, where n is the number of cells it marks, and, in general, we can be sure it is very slow when n is large. *Algorithm A would be too slow to make garbage collection a usable technique.*

Another fairly evident marking algorithm is to follow all paths and to record branch points on a stack as we go:

Algorithm B (*Marking*). This algorithm achieves the same effect as Algorithm A, using `STACK[1]`, `STACK[2]`, . . . as auxiliary storage to keep track of all paths that have not yet been pursued to completion.

B1. [Initialize.] Let T be the number of immediately accessible nodes; mark them and place pointers to them in STACK[1], . . . , STACK[T].

B2. [Stack empty?] If T = 0, the algorithm terminates.

B3. [Remove top entry.] Set K ← STACK[T], T ← T − 1.

B4. [Examine links.] If NODE(K) is an atom, return to B2. Otherwise, if NODE(RLINK(K)) is unmarked, mark it and set T ← T + 1, STACK[T] ← RLINK(K); if NODE(REF(K)) is unmarked, mark it and set T ← T + 1, STACK[T] ← REF(K). Return to B2. ▮

Algorithm B clearly has an execution time essentially proportional to the number of cells it marks, and this is as good as we could possibly expect; but it is not really usable for garbage collection because there is no place to keep the stack! It does not seem unreasonable to assume that the stack in Algorithm B might grow up to, say, five percent of the size of memory; but when garbage collection is called, and all available space has been used up, there is only a fixed (rather small) number of cells to use for such a stack. Most of the early garbage collection procedures were essentially based on this algorithm, and if the special stack space was used up, then the entire program was terminated.

A somewhat better alternative is possible, using a fixed stack size, by combining Algorithms A and B:

Algorithm C (*Marking*). This algorithm achieves the same effect as Algorithms A and B, using an auxiliary table of H cells, STACK[0], STACK[1], . . . , STACK [H − 1].

In this algorithm, the action "insert X on the stack" means the following: "Set T ← (T + 1) mod H, and STACK[T] ← X. If T = B, set B ← (B + 1) mod H and K1 ← min (K1, STACK[B])." (Note that T points to the current top of the stack, and B points one place below the current bottom; STACK essentially operates as an input-restricted deque.)

C1. [Initialize.] Set T ← −1, B ← −1, K1 ← M + 1. Mark all the immediately accessible nodes, and successively insert their locations onto the stack (as just described above).

C2. [Stack empty?] If T = B, go to C5.

C3. [Remove top entry.] Set K ← STACK[T], T ← (T − 1) mod H.

C4. [Examine links.] If NODE(K) is an atom, return to C2. Otherwise, if NODE(RLINK(K)) is unmarked, mark it and insert RLINK(K) on the stack. Similarly, if NODE(REF(K)) is unmarked, mark it and insert REF(K) on the stack. Return to C2.

C5. [Sweep.] If K1 > M, the algorithm terminates. (The variable K1 represents the smallest location where there is a possibility of a new lead to a node that should be marked.) Otherwise, if NODE(K1) is unmarked, increase K1 by 1 and repeat this step. If NODE(K1) is marked, set K ← K1, increase K1 by 1, and go to C4. ▮

This algorithm and Algorithm B can be improved if X is never put on the stack when NODE(X) is an atom; such modifications are straightforward and they have been left out to avoid making the algorithms unnecessarily complicated.

Algorithm C is essentially Algorithm A when H = 1 and Algorithm B when H = M; clearly, it is gradually more efficient as H becomes larger. Unfortunately, Algorithm C defies a precise analysis for the same reason as Algorithm A, and we have no good idea how large H should be to make this method fast enough. It is plausible but uncomfortable to say a value of H = 50 is sufficient to make Algorithm C usable for garbage collection in most applications.

Algorithms B and C use a stack kept in sequential memory locations; we have seen earlier in this chapter that linked memory techniques are well suited to maintaining stacks which are not consecutive in memory. This suggests the idea that we might keep the stack of Algorithm B somehow scattered *through the same memory area in which we are collecting garbage.* This could be done easily if we were to give the garbage collection routine a little more room in which to breathe. Suppose, for example, we assume that all Lists begin with a List head node; assume further that although the RLINK field of each header node is used to point to the first node of the corresponding list, the REF field of each header word is completely meaningless and inaccessible to the main program. We can then redesign Algorithm B so that the stack is maintained in the REF fields of the header nodes:

Algorithm D (*Marking*). This algorithm achieves the same effect as Algorithms A, B, and C, but it assumes that the REF field in each List head node may be changed by the marking routine, and that the REF field in other nonatomic nodes points to a List head.

D1. [Initialize.] Set T ← Λ. Then for each pointer P to the head of an immediately accessible List (cf. step A1 of Algorithm A), if NODE(P) is unmarked, mark it and set REF(P) ← T, T ← P.

D2. [Stack empty?] If T = Λ, the algorithm terminates.

D3. [Remove top entry.] Set P ← T, T ← REF(T).

D4. [Move right.] Set P ← RLINK(P); if NODE(P) is marked, go to D2. Otherwise mark NODE(P). If NODE(P) is not an atom, set Q ← REF(P); and if furthermore NODE(Q) is unmarked, mark it and set REF(Q) ← T, T ← Q. Repeat step D4. ▮

(This algorithm works with circularly linked Lists as well as with straight linkage, since the condition "NODE(P) is marked" in step D4, which terminates motion to the right, includes as special cases "P = Λ" for straight linkage, or "P is the head of the current List" for circular linkage.)

Algorithm D may be compared to Algorithm B, which is quite similar, and its running time is essentially proportional to the number of nodes marked. However, Algorithm D is *not* recommended, because its seemingly rather mild restrictions are often too stringent for a general List-processing system. This

algorithm essentially requires that all List structures be well-formed [as in (7)] whenever garbage collection is called into action. But algorithms which manipulate List structures *momentarily* leave the List structures malformed, and it is important that a garbage collector such as Algorithm D will not be used during these momentary periods. Moreover, there are several List-manipulation algorithms which intentionally play havoc with the link fields in Lists during their operation, although they are designed so that well-formed Lists are restored again after the algorithm has been completed; for example, see the differentiation algorithm, 2.3.2D, or the algorithm of exercise 2.3.1–21. Care must also be taken in step D1 when the program contains pointers to the middle of a List.

These considerations bring us to Algorithm E, which is an elegant marking method discovered independently by Peter Deutsch and by Herbert Schorr and W. M. Waite in 1965. The assumptions used in this algorithm are just a little different from those of Algorithms A through D.

Algorithm E (*Marking*). Assume that a collection of nodes is given having the following fields:

MARK (a one-bit field, initially zero in each node),
ATOM (another one-bit field),
ALINK (a pointer field),
BLINK (a pointer field).

When ATOM $= 0$, the ALINK and BLINK fields may contain Λ or a pointer to another node of the same format; when ATOM $= 1$, the contents of the ALINK and BLINK fields are irrelevant to this algorithm.

Given a pointer P0, this algorithm sets the MARK field to 1 in NODE(P0) and in every other node which can be reached from NODE(P0) by a chain of ALINK and BLINK pointers in nodes with ATOM $= 0$. The algorithm uses three pointer variables, T, Q, and P, and modifies the links during its execution in such a way that all ATOM, ALINK, and BLINK fields are restored to their original settings after completion, although they may be changed temporarily.

E1. [Initialize.] Set T $\leftarrow \Lambda$, P $\leftarrow$ P0. (Throughout the remainder of this algorithm, the variable T has a dual significance: When T $\neq \Lambda$, it points to the top of what is essentially a stack as in Algorithm D; and the node that T points to once contained a link equal to P in place of the "artificial" stack link which currently occupies NODE(T).)

E2. [Mark.] Set MARK(P) $\leftarrow 1$.

E3. [Atom?] If ATOM(P) $= 1$, go to E6.

E4. [Down ALINK.] Set Q $\leftarrow$ ALINK(P). If Q $\neq \Lambda$ and MARK(Q) $= 0$, set ATOM(P) $\leftarrow 1$, ALINK(P) $\leftarrow$ T, T $\leftarrow$ P, P $\leftarrow$ Q, and go to E2. (Here the ATOM field and ALINK fields are temporarily being altered, so that the list structure in certain marked nodes has been rather drastically changed. But these changes will be restored in step E6.)

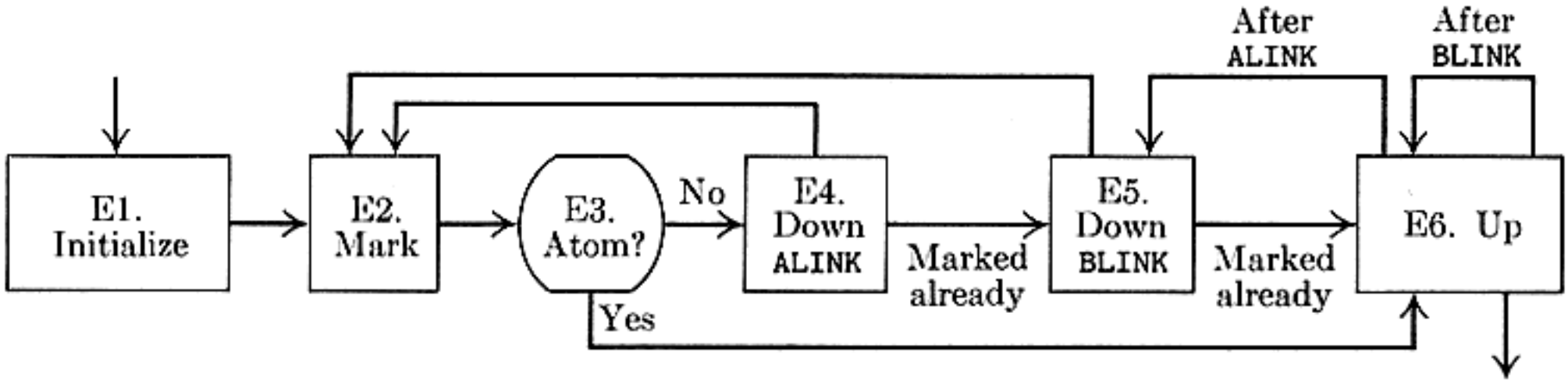


Fig. 38. Flowchart for Algorithm E.

E5. [Down BLINK.] Set Q ← BLINK(P). If Q ≠ Λ and MARK(Q) = 0, set BLINK(P) ← T, T ← P, P ← Q, and go to E2.

E6. [Up.] (This step undoes the link switching made in step E4 or E5; the setting of ATOM(T) tells whether ALINK(T) or BLINK(T) is to be restored.) If T = Λ, the algorithm terminates. Otherwise set Q ← T. If ATOM(Q) = 1, set ATOM(Q) ← 0, T ← ALINK(Q), ALINK(Q) ← P, P ← Q, and return to E5. If ATOM(Q) = 0, set T ← BLINK(Q), BLINK(Q) ← P, P ← Q, and return to E6. ∎

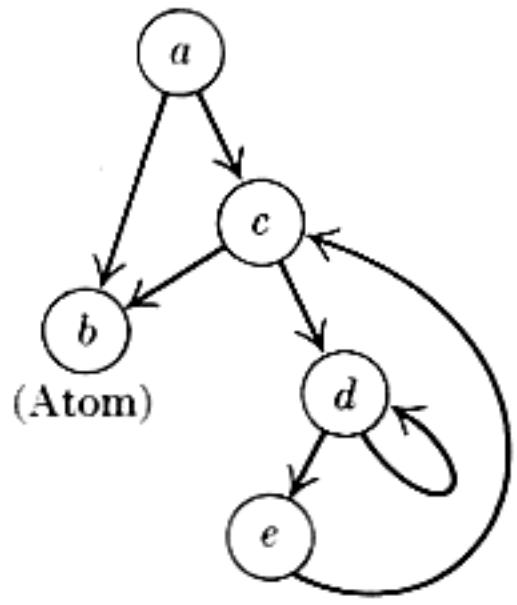
A flow chart of Algorithm E is shown in Fig. 38, and an example of the algorithm in action is given in Fig. 39, which shows the successive steps encountered for a simple List structure. The reader will find it worth while to study this algorithm very carefully; note how the list structure is artificially changed in steps E4 and E5, in order to keep track of the stack analogous to the stack in Algorithm D. When we return to a previous state, the ATOM field is used to tell which of ALINK, BLINK contains the artificial address. The "nesting" shown at the bottom of Fig. 39 illustrates how each nonatomic node is visited three times during Algorithm E (the same configuration (T, P) occurs at the beginning of steps E2, E5, and E6).

A proof that Algorithm E is valid can be formulated by induction on the number of nodes that are to be marked. One proves at the same time that P = P0 at the conclusion of the algorithm; for details, see exercise 3. Algorithm E will run faster if step E3 is deleted and instead special tests for "ATOM(Q) = 1" and appropriate actions are made in steps E4 and E5, as well as a test "ATOM(P0) = 1" in step E1. We have stated the algorithm in its present form for simplicity; the modifications just stated appear in the answer to exercise 4.

The idea used in Algorithm E can be applied to problems other than garbage collection; in fact, its use for tree traversal has already been mentioned in exercise 2.3.1–21. The reader may also find it useful to compare Algorithm E with the simpler problem solved in exercise 2.2.3–7.

Algorithms B and D are readily adapted to a situation where atomic nodes are indicated by a bit in the node that points to the atom, not in the atom itself. If Algorithm E is to be adapted to such a situation, it will be necessary to introduce another 1-bit field in nonatomic nodes. This is generally not hard to do. (For example, when there are two words per node, the least significant bit of each link field may be used to store temporary information.)

Although Algorithm E requires a time proportional to the number of nodes it marks, this constant of proportionality is not as small as in Algorithm B; the fastest garbage collection method known combines Algorithms B and E, as discussed in exercise 5.



a ALINK[MARK]	$b[0]$	.	$\Lambda[1]$	.	b	.	.	.	.	.	.	.	.	.	.
a BLINK[ATOM]	$c[0]$	.	[1]	.	[0]	Λ	.	.	.	.	.	.	.	.	c
b ALINK[MARK]	$-[0]$	.	.	[1]	.	.	.	.	.	.	.	.	.	.	.
b BLINK[ATOM]	$-[1]$	.	.	.	.	.	.	.	.	.	.	.	.	.	.
c ALINK[MARK]	$b[0]$	.	.	.	.	.	[1]	.	.	.	.	.	.	.	.
c BLINK[ATOM]	$d[0]$	.	.	.	.	.	.	a	.	.	.	.	.	d	.
d ALINK[MARK]	$e[0]$	.	.	.	.	.	.	.	$c[1]$	.	.	e	.	.	.
d BLINK[ATOM]	$d[0]$	.	.	.	.	.	.	.	[1]	.	.	[0]	.	.	.
e ALINK[MARK]	$\Lambda[0]$	.	.	.	.	.	.	.	.	[1]	.	.	.	.	.
e BLINK[ATOM]	$c[0]$	.	.	.	.	.	.	.	.	.	.	.	.	.	.
T	—	Λ	a	a	Λ	a	a	c	d	d	d	c	c	a	Λ
P	—	a	b	b	a	c	c	d	e	e	e	d	d	c	a
Next step	E1	E2	E2	E6	E5	E2	E5	E2	E2	E5	E6	E5	E6	E6	E6

Nesting

Fig. 39. A structure marked by Algorithm E. (The table shows only changes which have occurred since the previous step.)

Let us conclude this section with some quantitative remarks about the efficiency of garbage collection as opposed to the philosophy of "AVAIL ⇐ X", as was used in most of the previous examples in this chapter. In each of the previous cases we could have omitted all specific mention of returning nodes to free space and we could have substituted a garbage collector instead. (In a special-purpose application, as opposed to a set of general-purpose List manipulation subroutines, the programming and debugging of a garbage collector is more difficult than the methods we have used, and, of course, garbage collection requires an extra bit reserved in each node; but we are interested here in the relative speed of the programs once they have been written and debugged.)

The best garbage collection routines known have an execution time essentially of the form $c_1\mathtt{N} + c_2\mathtt{M}$, where c_1 and c_2 are constants, N is the number of nodes marked, and M is the total number of nodes in the memory. Thus M − N is the number of free nodes found, and the amount of time required to return

these nodes to free storage is $(c_1\mathtt{N} + c_2\mathtt{M})/(\mathtt{M} - \mathtt{N})$ per node. Let $\mathtt{N} = \rho\mathtt{M}$; this figure becomes $(c_1\rho + c_2)/(1 - \rho)$. So if $\rho = \frac{3}{4}$, i.e., if the memory is three-fourths full, it takes $3c_1 + 4c_2$ units of time per free node returned to storage; when $\rho = \frac{1}{4}$, the corresponding figure is only $\frac{1}{3}c_1 + \frac{4}{3}c_2$. If we do not use the garbage collection technique, the amount of time per node returned is essentially a constant, c_3, and it is doubtful that c_3/c_1 will be very large. Hence we can see to what extent garbage collection is inefficient when the memory becomes full, and how it is correspondingly efficient when the demand on memory is light.

It is possible to combine garbage collection with some of the other methods of returning cells to free storage; these ideas are not mutually exclusive, and some systems employ both the reference counter and the garbage collection schemes, besides allowing the programmer to explicitly erase nodes. The idea is to employ garbage collection only as a "last resort" whenever all other methods of returning cells have failed. This idea is of debatable merit, in view of the inefficiency of garbage collection when memory is nearly packed, and since so few programs come close to filling memory without exceeding the limits shortly thereafter.

EXERCISES

▶ **1.** [*M21*] In Section 2.3.4 we saw that trees are special cases of the "classical" mathematical concept of a directed graph. Can Lists be described in graph-theoretic terminology?

2. [*20*] In Section 2.3.1 we saw that tree traversal can be facilitated using a "threaded" representation inside the computer. Can List structures be threaded in an analogous way?

3. [*M26*] Prove the validity of Algorithm E. [*Hint:* See the proof of Algorithm 2.3.1T.]

4. [*28*] Write a MIX program for Algorithm E, assuming that nodes are represented as one MIX word, with MARK the (0:0) field [+ = 0, − = 1], ATOM the (1:1) field, ALINK the (2:3) field, BLINK the (4:5) field, and $\Lambda = 0$. Also, determine the execution time of your program in terms of relevant parameters. (Note that in the MIX computer the problem of determining whether a memory location contains −0 or +0 is not quite trivial, and this can be a factor in your program.)

5. [*25*] (Schorr and Waite.) Give a marking algorithm which combines Algorithms B and E as follows: The assumptions of Algorithm E with regard to the fields within nodes, etc., are assumed; however, an auxiliary stack STACK[1], STACK[2], . . . , STACK[N] is used as in Algorithm B, and the mechanism of Algorithm E is employed only when the stack is full.

6. [*00*] The quantitative discussion at the end of this section says garbage collection takes up approximately $c_1N + c_2M$ units of time; where does this "c_2M" term come from?

7. [*24*] (R. W. Floyd.) Design a marking algorithm that is similar to Algorithm E in that it uses no auxiliary stack, except (a) it has a more difficult task to do, in that each node contains only MARK, ALINK, and BLINK fields (so there is no ATOM field to

provide additional control), but (b) it has a simpler task to do, in that it marks only a binary tree instead of a general List. Here ALINK and BLINK are the usual LLINK and RLINK in a binary tree.

▶ **8.** [*27*] (L. P. Deutsch.) Design a marking algorithm similar to algorithms D and E in that it uses no auxiliary memory for a stack, but modify the method so that it works with nodes of variable size and with a variable number of pointers having the following format: The first word of a node has two fields MARK and SIZE; the MARK field is to be treated as in Algorithm E, and the SIZE field contains a number $n \geq 0$. This means there are n consecutive words after the first word, each containing two fields MARK (which is zero and should remain so) and LINK (which is Λ or points to the first word of another node). For example, a node with three pointers would comprise four consecutive words:

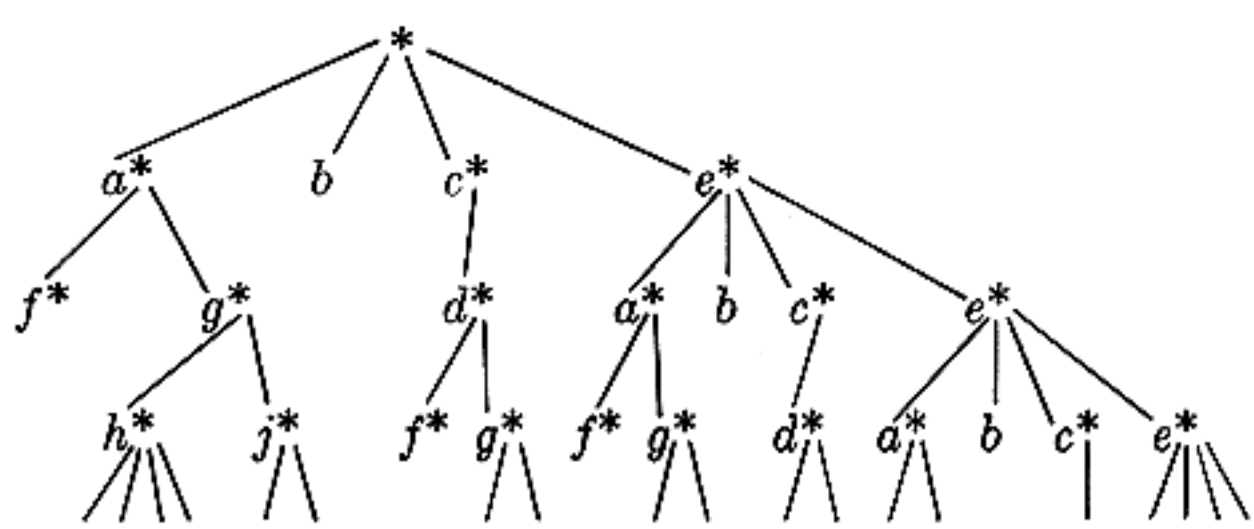
First word	MARK = 0 (Will be set to 1)	SIZE = 3
Second word	MARK = 0	LINK = first pointer
Third word	MARK = 0	LINK = second pointer
Fourth word	MARK = 0	LINK = third pointer.

9. [*28*] (D. Edwards.) Design an algorithm for the second phase of garbage collection which "compacts storage" in the following sense: Let NODE(1), . . . , NODE(M) be ▶ one-word nodes with fields MARK, ATOM, ALINK, and BLINK, as described in Algorithm E. Assume MARK = 1 in all nodes that are not garbage. The desired algorithm should relocate the marked nodes, if necessary, so that they all appear in consecutive locations NODE(1), . . . , NODE(k), and at the same time the ALINK and BLINK fields of nonatomic nodes should be altered if necessary so that the List structure is preserved.

10. [*28*] Design an algorithm which copies a List structure, assuming that an internal representation like that in (7) is being used. (Thus, if your procedure is asked to copy ▶ the List whose head is the node at the upper left corner of (7), a new set of Lists having 14 nodes, and with structure and information identical to that shown in (7), will be created.)

Assume that the List structure is stored in memory using S, T, REF, and RLINK fields as in (9), and that T(P0) = 0 (i.e., NODE(P0) is a List head). Assume further that the REF field in each List head node is Λ; if these REF fields are used by your copying procedure, they should be reset to Λ again afterwards.

11. [*M30*] Any List structure can be "fully expanded" into a tree structure by repeating all overlapping elements until none are left; when the List is recursive, this gives an infinite tree. For example, the List (5) would expand into an infinite tree whose first four levels are



Design an algorithm to test the equivalence of two List structures, in the sense that they have the same diagram when fully expanded. [For example, Lists A and B are equivalent in this sense, if

$$
\begin{aligned}
A &= (a{:}C, b, a{:}(b{:}D)) \\
B &= (a{:}(b{:}D), b, a{:}E) \\
C &= (b{:}(a{:}C)) \\
D &= (a{:}(b{:}D)) \\
E &= (b{:}(a{:}C)).]
\end{aligned}
$$

12. [*30*] (M. Minsky.) Show that it is possible to use a garbage collection method reliably in a "real time" application, e.g., when a computer is controlling some physical device, even when stringent upper bounds are placed on the maximum execution time required for each List operation performed. [*Hint:* Garbage collection can be arranged to work in parallel with the List operations, if appropriate care is taken.]

2.4. MULTILINKED STRUCTURES

Now that we have examined linear lists and tree structures in detail, the principles of representing structural information within a computer should be evident. In this section we will examine another application of these techniques, this time for the typical case in which the structural information is slightly more complicated: in higher-level applications, several types of structure are usually present simultaneously.

A "multilinked structure" involves nodes with several link fields in each node, not just one or two as in most of our previous examples. We have already seen some examples of multiple linkage, e.g., the simulated elevator system in Section 2.2.5 and the multivariate polynomials in Section 2.3.3.

We shall see that the presence of many different kinds of links per node does *not* necessarily make the accompanying algorithms any more difficult to write or to understand than the algorithms already studied. We will also discuss the important question, "*How much structural information ought to be explicitly recorded in memory?*"

The problem we will consider arises in connection with writing a compiler program for translating COBOL and related languages. A programmer who uses COBOL may give alphabetic names to the quantities in his program on several levels; for example, he may have two files of data for sales and purchases which have the following structure:

```
1 SALES                    1 PURCHASES
   2 DATE                     2 DATE
      3 MONTH                    3 DAY
      3 DAY                      3 MONTH
      3 YEAR                     3 YEAR
   2 TRANSACTION              2 TRANSACTION
      3 ITEM                     3 ITEM                    (1)
      3 QUANTITY                 3 QUANTITY
      3 PRICE                    3 PRICE
      3 TAX                      3 TAX
      3 BUYER                    3 SHIPPER
         4 NAME                     4 NAME
         4 ADDRESS                  4 ADDRESS
```

This configuration indicates that each item in SALES consists of two parts, the DATE and the TRANSACTION; the DATE is further divided into three parts, and likewise TRANSACTION has five subdivisions. Similar remarks apply to PURCHASES. The relative order of these names indicates the order in which the quantities appear in external representations of the file (e.g., punched cards or magnetic tape); note that in this example "DAY" and "MONTH" appear in opposite order in the two files. A COBOL programmer gives further information, not shown in this illustration, that tells how much space each item of information occupies and in what format it appears; these considerations are not relevant to us in this section, so they will not be mentioned further.

A COBOL programmer first describes the layout of his files and the other variables in his program, then he gives the algorithms that manipulate these quantities. To refer to an individual variable in the example above, it would not be sufficient merely to give the name DAY, since there is no way of telling if this variable called DAY is in the SALES file or in the PURCHASES file. Therefore a COBOL programmer is given the ability to write "DAY OF SALES" to refer to the DAY part of a SALES item. He could also write, more completely,

"DAY OF DATE OF SALES",

but in general there is no need to give more qualification than necessary to avoid ambiguity. Thus,

"NAME OF SHIPPER OF TRANSACTION OF PURCHASES"

may be abbreviated to

"NAME OF SHIPPER",

since only one part of the data has been called SHIPPER.

These rules of COBOL may be stated more precisely as follows:

a) Each name is immediately preceded by an associated positive integer called its "level number." A name either refers to an *elementary item* or else it is the name of a *group* of one or more items whose names follow. In the latter case, each item of the group must have the same level number, which must be greater than the level number of the group name. (For example, DATE and TRANSACTION above have level number 2, which is greater than the level number 1 of SALES.)

b) To refer to an elementary item or group of items named A_0, the general form is

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_n,$$

where $n \geq 0$ and where, for $0 \leq j < n$, A_j is the name of some item contained directly or indirectly within a group named A_{j+1}. There must be exactly one item A_0 satisfying this condition.

c) If the same name A_0 appears in several places, there must be a way to refer to each use of the name by using qualification.

As an example of rule (c), the data configuration

```
1 AA
  2 BB
    3 CC
    3 DD
  2 CC
```
(2)

would not be allowed, since there is no unambiguous way to refer to the second appearance of CC. (See exercise 4.)

There is another feature of COBOL which affects compiler writing and the application we are considering, namely an option in the language which makes it possible to refer to many items at once. A COBOL programmer may write

```
MOVE CORRESPONDING α TO β
```

which moves all items with corresponding names from data area α to data area β. For example, the COBOL statement

```
MOVE CORRESPONDING DATE OF SALES TO DATE OF PURCHASES
```

would mean that the values of MONTH, DAY, and YEAR from the SALES file are to be moved to the variables DAY, MONTH, YEAR in the PURCHASES file. (The relative order of DAY and MONTH is thereby interchanged.)

The problem we will investigate in this section is to design three algorithms suitable for use in a COBOL compiler, which are to do the following things:

Operation 1. To process a description of names and level numbers such as (1), putting the relevant information into tables within the compiler for use in operations 2 and 3.

Operation 2. To determine if a given qualified reference, as in rule (b), is valid, and when it is valid to locate the corresponding data item.

Operation 3. To find all corresponding pairs of items indicated by a "CORRESPONDING" statement.

We will assume that a "symbol table subroutine" exists within our compiler, which will convert an alphabetic name into a pointer to a memory location that contains a table entry for that name. (Methods for constructing symbol table algorithms are discussed in detail in Chapter 6.) In addition to the Symbol Table, there is a larger table which contains one entry for each item of data in the COBOL source program that is being compiled; we will call this the *Data Table.*

Clearly, we cannot design an algorithm for operation 1 until we know what kind of information is to be stored in the Data Table, and the form of the Data Table depends on what information we need to perform operations 2 and 3; thus we look first at operations 2 and 3.

In order to determine the meaning of the COBOL reference

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_n, \qquad n \geq 0, \tag{3}$$

we should first look up the name A_0 in the Symbol Table. There ought to be a series of links from the Symbol Table entry to all Data Table entries for this name. Then for each Data Table entry we will want a link to the entry for the group item which contains it. Now if there is a further link field from the Data Table items back to the Symbol Table, it is not hard to see how a reference like (3) can be processed. Furthermore, we will want some sort of links from the Data Table entries for group items to the items in the group, in order to locate the pairs indicated by "MOVE CORRESPONDING".

We have thereby found a possible need for five link fields in each Data Table entry:

PREV (a link to the previous entry with the same name, if any);
FATHER (a link to the smallest group, if any, containing this item);
NAME (a link to the Symbol Table entry for this item);
SON (a link to the first subitem of a group);
BROTHER (a link to the next subitem in the group containing this item).

It is clear that COBOL data structures like those for SALES and PURCHASES above are essentially trees; and the FATHER, SON, and BROTHER links which appear here are familiar from our previous study. (The conventional binary tree representation of a tree consists of the SON and BROTHER links; adding the FATHER link gives what we have called a "triply linked tree." The five links above consist of these tree links together with PREV and NAME which superimpose further information on the tree structure.)

Perhaps not all five of these links will turn out to be necessary, or sufficient, but we will first try to design our algorithms under the tentative assumption that Data Table entries will involve these five link fields (plus further information irrelevant to our problems). As an example of the multiple linking used, consider the two COBOL data structures

```
1 A          1 H
3 B          5 F
  7 C          8 G
  7 D        5 B
3 E          5 C
3 F            9 E
  4 G          9 D
               9 G
```
(4)

They would be represented as shown in (5) (with links indicated symbolically). Note that the LINK field of the Symbol Table entries points to the most recently encountered Data Table entry for the symbolic name in question.

The first algorithm we require is one which builds the Data Table in such a form. Note the flexibility in choice of level numbers which is allowed by the COBOL rules; the left structure in (4) is completely equivalent to

```
1 A
  2 B
    3 C
    3 D
  2 E
  2 F
    3 G
```

because level numbers do not have to be sequential. [See rule (a), stated near

Symbol Table

		LINK
A:		A1
B:		B5
C:		C5
D:		D9
E:		E9
F:		F5
G:		G9
H:		H1

(Shading indicates additional information which is not relevant here)

Data Table

	PREV	FATHER	NAME	SON	BROTHER	
A1:	Λ	Λ	A	B3	H1	
B3:	Λ	A1	B	C7	E3	
C7:	Λ	B3	C	Λ	D7	
D7:	Λ	B3	D	Λ	Λ	
E3:	Λ	A1	E	Λ	F3	
F3:	Λ	A1	F	G4	Λ	
G4:	Λ	F3	G	Λ	Λ	
H1:	Λ	Λ	H	F5	Λ	
F5:	F3	H1	F	G8	B5	
G8:	G4	F5	G	Λ	Λ	
B5:	B3	H1	B	Λ	C5	
C5:	C7	H1	C	E9	Λ	
E9:	E3	C5	E	Λ	D9	
D9:	D7	C5	D	Λ	G9	
G9:	G8	C5	G	Λ	Λ	

(5)

the beginning of this section.] There are sequences of level numbers which may not be used, however; for example, if the level number of D in (4) were changed to "6" (in either place) we would have a meaningless data configuration which violates the rule that all items of a group should have the same number, which is higher than the group name. The following algorithm therefore makes sure that this restriction is met.

Algorithm A (*Build Data Table*). This algorithm is given a sequence of pairs (L, P), where L is a positive integer "level number" and P points to a Symbol Table entry, corresponding to COBOL data structures, such as (4) above. The algorithm builds a Data Table as in the example (5) above. When P points to a Symbol Table entry that has not appeared before, LINK(P) will equal Λ. This algorithm uses an auxiliary stack which is treated as usual (using either sequential memory locations, as in Section 2.2.2, or linked allocation, as in Section 2.2.3.)

A1. [Initialize.] Set the stack contents to the single entry (0, Λ). (The stack entries throughout this algorithm are pairs (L, P), where L is an integer and P a pointer; as this algorithm proceeds, the stack contains the level number

and pointers to the last data entries on all levels higher in the tree than the current level. For example, just before encountering the pair "3 F" in the above example, the stack would contain

$$(0, \Lambda) \qquad (1, \mathrm{A1}) \qquad (3, \mathrm{E3})$$

from bottom to top.)

A2. [Next item.] Let (L, P) be the next data item from the input. If the input is exhausted, however, the algorithm terminates. Set Q ⇐ AVAIL (i.e., let Q be the location of a new node in which we can put the next Data Table entry).

A3. [Set name links.] Set

$$\mathtt{PREV(Q)} \leftarrow \mathtt{LINK(P)}, \qquad \mathtt{LINK(P)} \leftarrow \mathtt{Q}, \qquad \mathtt{NAME(Q)} \leftarrow \mathtt{P}.$$

(This properly sets two of the five links in NODE(Q). We now want to set FATHER, SON, and BROTHER appropriately.)

A4. [Compare levels.] Let the top entry of the stack be (L1, P1). If L1 < L, set SON(P1) ← Q (or, if P1 = Λ, set FIRST ← Q, where FIRST is a variable which is to point to the first Data Table entry) and go to A6.

A5. [Remove top level.] If L1 > L, remove the top stack entry, let (L1, P1) be the new entry which has just come to the top of the stack, and repeat step A5. If L1 < L, signal an error (mixed numbers have occurred on the same level). Otherwise, i.e. when L1 = L, set BROTHER(P1) ← Q, remove the top stack entry, and let (L1, P1) be the pair which has just come to the top of the stack.

A6. [Set family links.] Set

$$\mathtt{FATHER(Q)} \leftarrow \mathtt{P1}, \ \mathtt{SON(Q)} \leftarrow \Lambda, \ \mathtt{BROTHER(Q)} \leftarrow \Lambda.$$

A7. [Add to stack.] Place (L, Q) on the top of the stack, and return to step A2. ▮

The introduction of an auxiliary stack, as explained in step A1, makes this algorithm so straight forward that it needs no further explanation.

The next problem is to locate the data table entry corresponding to a reference

$$A_0 \ \mathtt{OF} \ A_1 \ \mathtt{OF} \ \ldots \ \mathtt{OF} \ A_n, \qquad n \geq 0. \tag{6}$$

A "good" compiler will also check to ensure that such a reference is unambiguous. In this case, a straightforward algorithm suggests itself immediately: all we need to do is to run through the list of Data Table entries for the name A_0 and make sure that exactly one of these entries matches the stated qualification $A_1, \ldots, A_n$.

Algorithm B (*Check a qualified reference*). Corresponding to reference (6), a Symbol Table subroutine will find pointers $P_0, P_1, \ldots, P_n$ to the Symbol Table entries for $A_0, A_1, \ldots, A_n$, respectively.

The purpose of this algorithm is to examine $P_0, P_1, \ldots, P_n$ and either to determine that reference (6) is in error, or to set variable Q to the address of the Data Table entry for the item refered to by (6).

B1. [Initialize.] Set Q $\leftarrow \Lambda$, P $\leftarrow$ LINK(P_0).

B2. [Done?] If P $= \Lambda$, the algorithm terminates; at this point Q will equal Λ if (6) does not correspond to any Data Table entry. Otherwise set S $\leftarrow$ P and $k \leftarrow 0$. (S is a pointer variable which will run from P up the tree through FATHER links; k is an integer variable which goes from 0 to n. In practice, the pointers $P_0, \ldots, P_n$ would often be kept in a linked list, and instead of k, we would substitute a pointer variable which traverses this list; see exercise 5.)

B3. [Match complete?] If $k < n$ go on to B4. Otherwise we have found a matching data table entry; if Q $\neq \Lambda$, this is the second entry found, so an error condition is signaled. Set Q $\leftarrow$ P, P $\leftarrow$ PREV(P), and go to B2.

B4. [Increase k.] Set $k \leftarrow k + 1$.

B5. [Move up tree.] Set S $\leftarrow$ FATHER(S). If S $= \Lambda$, we have failed to find a match; set P $\leftarrow$ PREV(P) and go to B2.

B6. [A_k match?] If NAME(S) $= P_k$, go to B3, otherwise go to B5. ∎

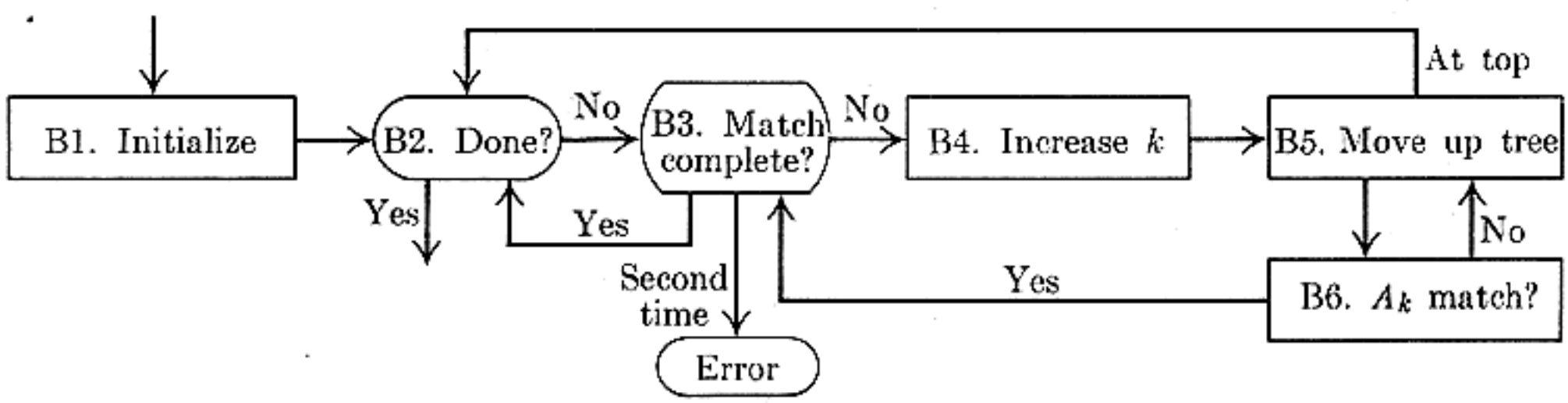


Fig. 40. Algorithm for checking a COBOL reference.

Note that the SON and BROTHER links are not needed by this algorithm.

The third and final algorithm that we need concerns "MOVE CORRESPONDING", and before we design such an algorithm, we must have a precise definition of what is required. The COBOL statement

$$\texttt{MOVE CORRESPONDING } \alpha \texttt{ TO } \beta \tag{7}$$

where α and β are references such as (6) to data items, is an abbreviation for the set of all statements

$$\texttt{MOVE } \alpha' \texttt{ TO } \beta'$$

where there exists an integer $n \geq 0$ and n names $A_0, A_1, \ldots, A_{n-1}$ such that

$$\begin{aligned} \alpha' &= A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_{n-1} \text{ OF } \alpha \\ \beta' &= A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_{n-1} \text{ OF } \beta \end{aligned} \tag{8}$$

and either α' or β' is an elementary item (not a group item). Furthermore we require that (8) show *complete* qualifications, i.e., that A_{j+1} is the father of A_j for $0 \leq j \leq n-1$; α' and β' must be exactly n levels farther down in the tree than α and β are.

In our example (4),

"MOVE CORRESPONDING A TO H"

is an abbreviation for the statements

```
MOVE B OF A TO B OF H
MOVE G OF F OF A TO G OF F OF H
```

The algorithm to recognize all corresponding pairs α', β' is quite interesting although not difficult; we move through the tree, whose root is α, in preorder, simultaneously looking in the β tree for matching names, and skipping over subtrees in which no corresponding elements can possibly occur. The names $A_0, \ldots, A_{n-1}$ of (8) are discovered in the opposite order $A_{n-1}, \ldots, A_0$.

Algorithm C (*Find* CORRESPONDING *pairs*). Given PO and QO, which point to Data Table entries for α and β, respectively, this algorithm successively finds all pairs (P, Q) of pointers to items (α', β') satisfying the constraints mentioned above.

C1. [Initialize.] Set P ← PO, Q ← QO. (In the remainder of this algorithm, the pointer variables P and Q will walk through trees having the respective roots α and β.)

C2. [Elementary?] If SON(P) = Λ or SON(Q) = Λ, output (P,Q) as one of the desired pairs and go to C5. Otherwise set P ← SON(P), Q ← SON(Q). (In this step, P and Q point to items α' and β', satisfying (8), and we wish to MOVE α' TO β' if and only if either α' or β' (or both) is an elementary item.)

C3. [Match name.] (Now P and Q point to data items which have respective qualifications of the forms

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_{n-1} \text{ OF } \alpha$$

and

$$B_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_{n-1} \text{ OF } \beta.$$

The object is to see if we can make $B_0 = A_0$ by examining all the names of the group A_1 OF ... OF A_{n-1} OF β.) If NAME(P) = NAME(Q), go to C2 (a

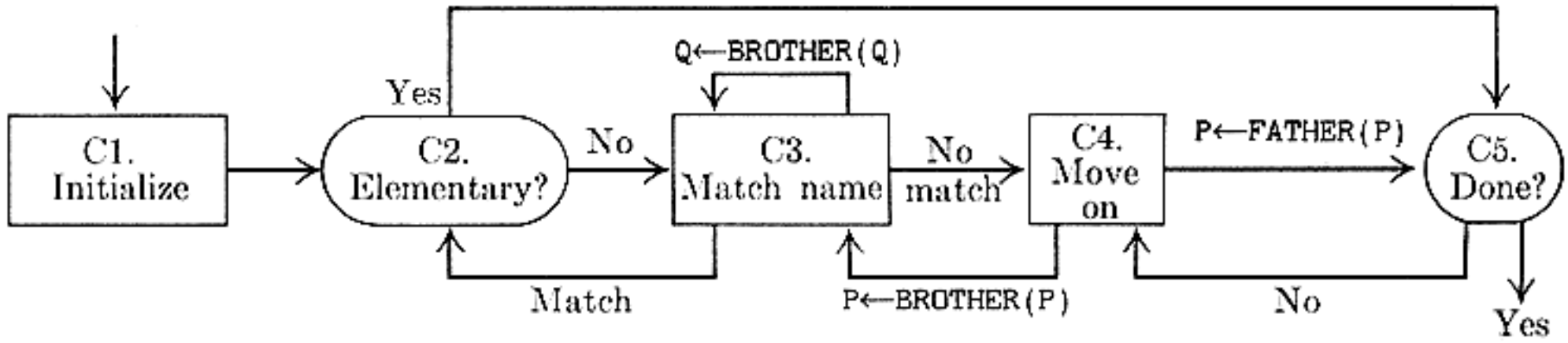


Fig. 41. Algorithm for "MOVE CORRESPONDING."

match has been found). Otherwise, if BROTHER(Q) $\neq \Lambda$, set Q $\leftarrow$ BROTHER(Q) and repeat step C3. (If BROTHER(Q) $= \Lambda$, no matching name is present in the group, and we continue on to step C4.)

C4. [Move on.] If BROTHER(P) $\neq \Lambda$, set

$$\texttt{P} \leftarrow \texttt{BROTHER(P)} \qquad \text{and} \qquad \texttt{Q} \leftarrow \texttt{SON(FATHER(Q))},$$

and go back to C3. If BROTHER(P) $= \Lambda$, set

$$\texttt{P} \leftarrow \texttt{FATHER(P)} \qquad \text{and} \qquad \texttt{Q} \leftarrow \texttt{FATHER(Q)}.$$

C5. [Done?] If P = P0, the algorithm terminates; otherwise go to C4. ▮

A flow chart for this algorithm is shown in Fig. 41. A proof that this algorithm is valid can readily be constructed by induction on the size of the trees involved (see exercise 9).

At this point it is worth while to study the ways in which the five link fields PREV, FATHER, NAME, SON, and BROTHER are used by Algorithms B and C. The striking feature is that these five links constitute a "complete set" in the sense that Algorithms B and C do virtually the minimum amount of work as they move through the Data Table; whenever it is necessary to refer to another Data Table entry, its address is immediately available at our fingertips; we do not need to search fo. it. It would be difficult to imagine how Algorithms B and C could possibly be made any faster if any additional link information were present in the table. (See exercise 11, however.)

Each link field may be viewed as a *clue* to the program, planted there in order to make the algorithms run faster (although, of course, algorithms which build the tables, like Algorithm A, run correspondingly slower, since they have more links to fill in). On the other hand it is clear that the Data Table constructed above contains much redundant information. Let us consider what would happen if we were to *delete* certain of the link fields.

The PREV link, while not used in Algorithm C, is extremely important for Algorithm B, and it seems to be an essential part of any COBOL compiler unless lengthy searches are to be carried out. A field which links together all items of the same name therefore seems essential for efficiency. We could perhaps modify the strategy slightly and adopt circular linking instead of terminating each list

with Λ, but there is no reason to do this unless other link fields are changed or eliminated.

The FATHER link is used in both Algorithms B and C, although its use in Algorithm C could be avoided if we used an auxiliary stack in that algorithm, or if we augmented BROTHER so that "thread" links are included (cf. Section 2.3.2). So we see that the FATHER link has been used in an essential way only in Algorithm B. If the BROTHER link were threaded, so that the items which now have BROTHER $= \Lambda$ would have BROTHER = FATHER instead, it would be possible to locate the father of any data item by following the BROTHER links; the added "thread" links could be distinguished either by having a new TAG field in each node that says whether the BROTHER link is a thread, or by the condition "BROTHER(P) $<$ P" if the Data Table entries are kept consecutively in memory in order of appearance. This would mean a short search would be necessary in step B5, and the algorithm would be correspondingly slower.

The NAME link is used by the algorithms only in steps B6 and C3. In both cases we could make the tests "NAME(S) $= P_k$", "NAME(P) = NAME(Q)" in other ways if the NAME link were not present (cf. exercise 10), but this would significantly slow down the inner loops of both Algorithms B and C. Here again we see a trade-off between the space for a link and the speed of the algorithms. (The speed of Algorithm C is not especially significant in COBOL compilers, when typical uses of MOVE CORRESPONDING are considered; but Algorithm B should be fast.) Experience indicates that other important uses are found for the NAME link within a COBOL compiler, especially in printing diagnostic information.

Since Algorithm A builds the Data Table step by step, and never has occasion to return it to the pool of available storage, we usually find that Data Table entries take consecutive memory locations in the order of appearance of the data items in the COBOL source program. Thus in our example (5), locations A1, B3, . . . would follow each other. This sequential nature of the Data Table leads to certain simplifications; for example, the SON link of each node is either Λ or it points to the node immediately following, so SON can be reduced to a 1-bit field. Alternatively, SON could be removed in favor of a test if FATHER(P $+ c$) = P, where c is the node size in the Data Table.

Thus the five link fields are not all essential, although they are helpful from the standpoint of speed in Algorithms B and C. This situation is fairly typical of most multilinked structures.

It is interesting to note that at least half a dozen people writing COBOL compilers have independently arrived at this same way to maintain a Data Table using five links (or four of the five, usually with the SON link missing). The first publication of such a technique was by H. W. Lawson, Jr. (*ACM National Conference Digest*, Syracuse, N.Y., 1962). But in 1965 an ingenious technique for achieving the effects of Algorithms B and C, using *only two link fields* and sequential storage of the Data Table, without a very great decrease in speed, was introduced by David Dahm; see exercises 12 through 14.

1. [*00*] Considering COBOL data configurations as tree structures, are the data items listed by a COBOL programmer in preorder, postorder, endorder, or none of these orders?

2. [*10*] Comment about the running time of Algorithm A.

3. [*22*] The PL/I language accepts data structures much like those in COBOL, except any sequence of level numbers is possible. For example, the sequence

```
1 A                    1 A
3 B                    2 B
5 C    is equivalent to 3 C
4 D                    3 D
2 E                    2 E
```

In general, rule (a) is modified to read, "The items of a group must have a sequence of nonincreasing level numbers, all of which are greater than the level number of the group name." What modifications to Algorithm A would change it from the COBOL convention to this PL/I convention?

▶ **4.** [*26*] Algorithm A does not detect the error if a COBOL programmer violates rule (c) stated in the text. How should Algorithm A be modified so that only data structures satisfying rule (c) will be accepted?

5. [*20*] In practice, Algorithm B may be given a linked list of Symbol Table references as input, instead of what we called "$P_0, P_1, \ldots, P_n$." Let T be a pointer variable such that

$$\texttt{INFO(T)} \equiv P_0,\ \texttt{INFO(RLINK(T))} \equiv P_1,\ \ldots,$$
$$\texttt{INFO(RLINK}^n\texttt{(T))} \equiv P_n,\ \texttt{RLINK}^{n+1}\texttt{(T)} = \Lambda.$$

Show how to modify Algorithm B so that it uses such a linked list as input.

6. [*23*] The PL/I language accepts data structures much like those in COBOL, but does not make the restriction of rule (c); instead, we have the rule that a qualified reference (3) is unambiguous if it shows "complete" qualification, i.e., if A_{j+1} is the father of A_j for $0 \le j < n$, and if A_n has no father. Rule (c) is now weakened to the simple condition that no two items of a group may have the same name. The second "CC" in (2) would be referred to as "CC OF AA" without ambiguity; the three data items

```
1 A
  2 A
    3 A
```

would be referred to as "A", "A OF A", "A OF A OF A" with respect to the PL/I convention just stated. (*Note:* Actually the word "OF" is replaced by a period in PL/I, and the order is reversed; "CC OF AA" would really be written "AA.CC" in PL/I, but this is not important for the purposes of the present exercise.) Show how to modify Algorithm B so that it follows the PL/I convention, i.e., it does not regard a complete qualification as ambiguous.

7. [*15*] What does the COBOL statement "MOVE CORRESPONDING SALES TO PURCHASES" mean, given the data structures in (1)?

8. [*10*] Under what circumstances is

"MOVE CORRESPONDING α TO β"

exactly the same as

"MOVE α TO β",

according to the definition in the text?

9. [*M23*] Prove that Algorithm C is correct.

10. [*23*] (a) How could the test "NAME(S) = P_k" in step B6 be performed if there were no NAME link in the Data Table nodes? (b) How could the test "NAME(P) = NAME(Q)" in step C3 be performed if there were no NAME link in the Data Table entries? (Assume that all other links are present as in the text.)

▶ **11.** [*23*] What additional links or changes in the strategy of the algorithms of the text could make Algorithm B or Algorithm C faster?

12. [*25*] (D. M. Dahm.) Consider representing the Data Table in sequential locations with just two links for each item:

PREV (as in the text);
SCOPE (links to the last elementary item in this group).

We have SCOPE(P) = P if and only if NODE(P) represents an elementary item. For example, the Data Table of (5) would be replaced by

	PREV	SCOPE		PREV	SCOPE
A1:	A	G4	H1:	A	G9
B3:	A	D7	F5:	F3	G8
C7:	A	C7	G8:	G4	G8
D7:	A	D7	B5:	B3	B5
E3:	A	E3	C5:	C7	G9
F3:	A	G4	E9:	E3	E9
G4:	A	G4	D9:	D7	D9
			G9:	G8	G9

(Compare with (4) of Section 2.3.3.) Note that NODE(P) is part of the tree below NODE(Q) if and only if Q < P ≤ SCOPE(Q). Design an algorithm which performs the function of Algorithm B when the Data Table has this format.

▶ **13.** [*24*] Give an algorithm to substitute for Algorithm A when the Data Table is to have the format shown in exercise 12.

▶ **14.** [*28*] Give an algorithm to substitute for Algorithm C when the Data Table has the format shown in exercise 12.

We have seen how the use of links implies that tables need not be sequentially located in memory; a number of tables may independently grow and shrink in a common "pooled" memory area. However, our discussions have always tacitly assumed that all nodes are the same size, i.e., that they take the same number of memory cells.

For a great many applications, a suitable compromise can be found so that a uniform node size is used for all tables (for example, see exercise 2). Instead of simply taking the maximum size that is needed and wasting space in smaller nodes, it is customary to pick a rather small node size and to employ what may be called the classical *linked-memory philosophy:* "If there isn't room for the information here, let's put it somewhere else and plant a link to it."

For a great many other applications, however, a single node size is not reasonable; we often wish to have nodes of varying sizes sharing a common memory area. Putting this another way, we want algorithms for reserving and freeing variable-size blocks of memory from a larger storage area, where these blocks are to consist of consecutive memory locations. Such techniques are generally called "dynamic storage allocation" algorithms.

Sometimes, often in simulation programs, we want dynamic storage allocation for nodes of rather small sizes (say one to ten words); and at other times, often in "executive" control programs, we are dealing primarily with rather large blocks of information. These two points of view lead to slightly different approaches to dynamic storage allocation, although the methods have much in common. For uniformity in terminology between these two approaches, we will generally use the terms *block* and *area* rather than "node" in this section, to denote a set of contiguous memory locations.

A. Reservation. Figure 42 shows a typical "memory map" or "checkerboard," a chart showing the current state of some memory pool. In this case the memory is shown partitioned into 53 blocks of storage that are "reserved," i.e. in use, mixed together with 21 "free" or "available" blocks that are not in use. After

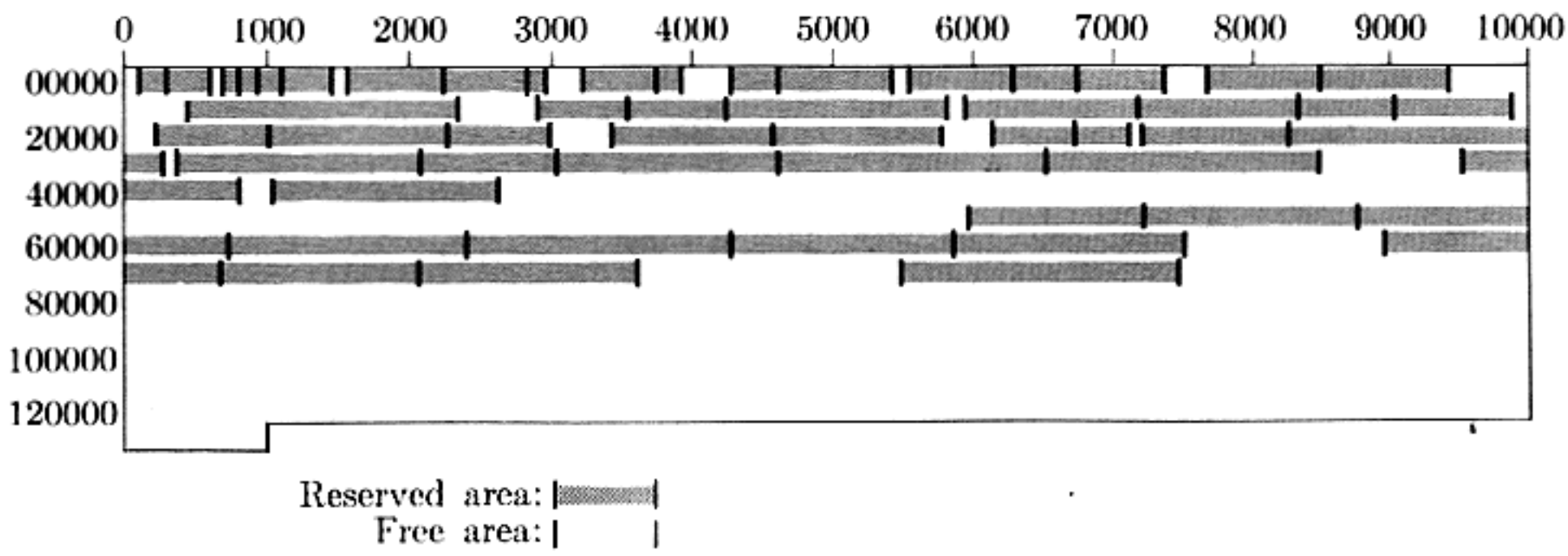


Fig. 42. A memory map.

dynamic storage allocation has been in operation for a while, the computer memory will perhaps look something like this; our first problem is

a) How is this partitioning of available space to be represented inside the computer?
b) Given such a representation of the available spaces, what is a good algorithm for finding a block of n consecutive free spaces and reserving them?

The answer to question (a) is, of course, to keep a *list* of the available space somewhere; this is almost always done best by using the available space *itself* to contain such a list. (An exception is the case when we are allocating storage for a disk file or other memory in which nonuniform access time makes it better to maintain a separate directory of available space.)

Thus, we can *link together* the available segments: the first word of each free storage area may contain the size of that block and the address of the next free area. The free blocks can be linked together in increasing or decreasing order of size, or in order of memory address, or in essentially random order.

For example, consider Fig. 42, which illustrates a large memory of 131,072 words, addressed from 0 to 131071. If we link together the available blocks in order of memory location, we would have one variable `AVAIL` that points to the first free block (in this case `AVAIL` would equal 0), and the other blocks would be represented as follows:

location	SIZE	LINK	
0	101	632	
632	42	1488	
⋮	⋮	⋮	[17 similar entries]
73654	1909	77519	
77519	53553	Λ	[special marker for last link]

Thus locations 0 through 100 form the first available block; after the reserved areas 101–290 and 291–631 shown in Fig. 42, we have more free space in location 632–673; etc.

As for question (b), if we want n consecutive words, clearly we must locate some block of $m \geq n$ available words and reduce its size to $m - n$. (Furthermore, when $m = n$, we must also delete this block from the list.) There may be several blocks with n or more cells, and so the question becomes *which* area should be chosen?

Two principal answers to this question suggest themselves: We can use the "best-fit" method or the "first-fit" method. In the former case, we decide to choose an area with m cells, where m is the smallest value present which is n or more. This usually requires searching the entire list of available space before a decision can be made. The "first-fit" method, on the other hand, simply chooses the first area encountered that has $\geq n$ words.

Historically, the best-fit method was widely used for several years; this naturally appears to be a good policy since it saves the larger available areas for a later time when they might be needed. But several objections to the best-fit technique can be raised: It is rather slow, since it requires a fairly long search; if "best fit" is not substantially better than "first fit" for other reasons, this extra searching time is not worth while. More importantly, the best-fit method tends to increase the number of very small blocks, and proliferation of small blocks is usually undesirable. There are certain situations in which the first-fit technique is demonstrably better than the best-fit method; for example, suppose we are given just two available areas of memory, of sizes 1300 and 1200, and suppose there are subsequent requests for blocks of sizes 1000, 1100, and 250:

memory request	*available areas, "first fit"*	*available areas, "best fit"*	
—	1300, 1200	1300, 1200	
1000	300, 1200	1300, 200	(1)
1100	300, 100	200, 200	
250	50, 100	stuck	

For these reasons the first-fit method can be recommended.

Algorithm A (*First-fit method*). Let AVAIL point to the first available block of storage, and suppose that each available block with address P has two fields: SIZE(P), the number of words in the block; and LINK(P), a pointer to the next available block. The last pointer is Λ. This algorithm searches for and reserves a block of N words, or reports failure.

A1. [Initialize.] Set Q $\leftarrow$ LOC(AVAIL). (Throughout the algorithm we use two pointers, Q and P, which are generally related by the condition P = LINK(Q). *We assume that*

$$\texttt{LINK(LOC(AVAIL))} = \texttt{AVAIL}.)$$

A2. [End of list?] Set P $\leftarrow$ LINK(Q). If P = Λ, the algorithm terminates unsuccessfully; there is no room for a block of N consecutive words.

A3. [Is SIZE enough?) If SIZE(P) $\geq$ N, go to A4; otherwise set Q $\leftarrow$ P and return to step A2.

A4. [Reserve N.] Set K $\leftarrow$ SIZE(P) $-$ N. If K = 0, set LINK(Q) $\leftarrow$ LINK(P). (This, of course, removes an empty area from the list.) Set SIZE(P) $\leftarrow$ K. The algorithm terminates successfully, having reserved an area of length N beginning with location P + K. ▌

This algorithm is certainly straightforward enough. However, a significant improvement in its running speed can be made with only a rather slight change in strategy. This improvement is quite important, and the reader will find it a pleasure to discover it for himself (see exercise 6).

Algorithm A may be used whether storage allocation is desired for small N or large N. Let us temporarily assume, however, that we are primarily interested in *large* values of N. Then note what happens when SIZE(P) is equal to N + 1 in that algorithm: we get to step A4 and reduce SIZE(P) to 1. In other words, an available block of size 1 has just been created; this block is so small it is virtually useless, and it just clogs up the system. We would have been better off if we had reserved the whole block of N + 1 words, instead of saving the extra word; it is often better to expend a few words of memory to avoid handling some unimportant details. Similar remarks apply to blocks of N + K words when K is very small.

If we allow the possibility of reserving slightly more than N words, it will be necessary to remember how many words have been reserved, so that later when this block becomes available again the entire set of N + K words is freed. This added amount of bookkeeping means that we are tying up space in *every* block in order to make the system more efficient only in certain circumstances when a "tight fit" is found; so the strategy doesn't seem especially attractive. However, a special *control word* as the first word of each variable-size block often turns out to be desirable for other reasons, and so it is usually not unreasonable to expect the SIZE field to be present in the first word of every block whether it is available or not.

In accordance with these conventions, we would modify step A4 above to read as follows:

"A4′. [Reserve $\geq$N.] Set K $\leftarrow$ SIZE(P) − N. If K $< c$ (where c is a small positive constant chosen to reflect an amount of storage we are willing to sacrifice in the interests of saving time), set

$$\texttt{LINK(Q)} \leftarrow \texttt{LINK(P)} \qquad \text{and} \qquad \texttt{L} \leftarrow \texttt{P}.$$

Otherwise set

$$\texttt{SIZE(P)} \leftarrow \texttt{K}, \qquad \texttt{L} \leftarrow \texttt{P} + \texttt{K}, \qquad \text{and} \qquad \texttt{SIZE(L)} \leftarrow \texttt{N}.$$

The algorithm terminates successfully, having reserved an area of length N or more beginning with location L."

A value for the constant c of about 8 or 10 is suggested, although very little theory or empirical evidence exists to compare this with other choices. When the best-fit method is being used, the test of K $< c$ is even *more* important than it is to the first-fit method, because tighter fits (smaller values of K) are much more likely to occur, and the number of available blocks should be kept as small as possible for that algorithm.

B. Liberation. Now let us consider the inverse problem: How should we return blocks to the available space list when they are no longer needed?

It is perhaps tempting to dismiss this problem by using "garbage collection" (see Section 2.3.5); we could follow a policy of simply doing nothing until space

runs out, then searching for all the areas currently in use and fashioning a new AVAIL list.

The idea of garbage collection is not to be recommended, however, for all applications. In the first place, we need a fairly "disciplined" use of pointers if we are to be able to guarantee that all areas currently in use will be easy to locate, and this amount of discipline is often lacking in the applications considered here. Secondly, as we have seen before, garbage collection tends to be slow when the memory is nearly full.

There is another more important reason why garbage collection is not satisfactory, due to a phenomenon which did not confront us in our previous discussion of the technique: Suppose that there are two adjacent areas of memory, both of which are available, but because of the garbage-collection philosophy one of them (shown shaded) is not in the AVAIL list.

(2)

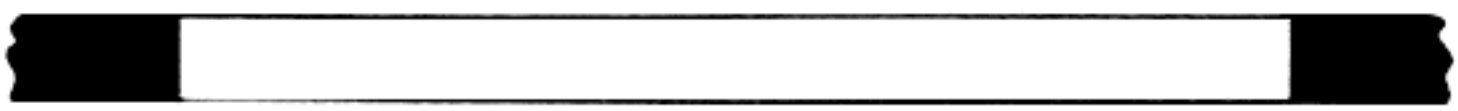
In this diagram, the heavily shaded areas at the extreme left and right are unavailable. We may now reserve a section of the area known to be available:

(3)

If garbage collection occurs at this point, we have two separate free areas,

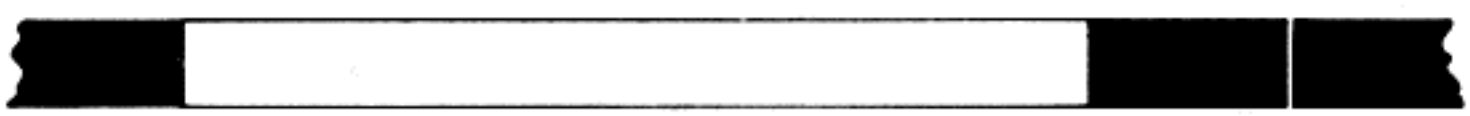
(4)

Boundaries between available and reserved areas have a tendency to perpetuate themselves, and as time goes on the situation gets progressively worse. But if we had used a philosophy of returning blocks to the AVAIL list as soon as they become free, *and collapsing adjacent available areas together*, we would have collapsed (2) into



(5)

and we would have obtained



(6)

which is much better than (4). This phenomenon causes the garbage-collection technique to leave memory more broken up than it should be.

In order to remove this difficulty, it is possible to use garbage collection together with the process of *compacting memory*, i.e., moving all the reserved blocks into consecutive locations, so that all available blocks come together whenever garbage collection is done. The allocation algorithm now becomes completely trivial by contrast with Algorithm A, since there is only one available block at all times. Even though this technique takes time to recopy all the

locations that are in use, and to change the value of the link fields therein, it can be applied with reasonable efficiency when there is a disciplined use of pointers, and when there is a spare link field in each block for use by the garbage collection algorithms. (See exercise 33.)

Since many applications do not meet these requirements for the feasibility of garbage collection, we shall now study methods for returning blocks of memory to the available space list. The only difficulty in these methods is the collapsing problem: two adjacent free areas should be merged into one. In fact, when an area bounded by two available blocks becomes free, all three areas should be merged together into one. In this way *a good balance is obtained in memory even though storage areas are continually reserved and freed over a long period of time.* (For a proof of this fact, see the "fifty-percent rule" below.)

The problem is to determine whether the areas at either side of the returned block are currently available; and if they are, we want to update the AVAIL list properly. (The latter operation is a little more difficult than it sounds.)

The first solution to these problems is to maintain the AVAIL list in order of increasing memory locations.

Algorithm B (*Liberation with sorted list*). Under the assumptions of Algorithm A, with the additional assumption that the AVAIL list is sorted by memory location (i.e., if P points to an available block and LINK(P) $\neq \Lambda$, then LINK(P) $>$ P), this algorithm adds the block of N consecutive cells beginning at location P0 to the AVAIL list. We naturally assume that none of these N cells is already available.

B1. [Initialize.] Set Q $\leftarrow$ LOC(AVAIL). (See the remarks in step A1 above.)

B2. [Advance P.] Set P $\leftarrow$ LINK(Q). If P $= \Lambda$, or if P $>$ P0, go to B3; otherwise set Q $\leftarrow$ P and repeat step B2.

B3. [Check upper bound.] If P0 $+$ N $=$ P (and P $\neq \Lambda$), set N $\leftarrow$ N $+$ SIZE(P) and set LINK(P0) $\leftarrow$ LINK(P). Otherwise set LINK(P0) $\leftarrow$ P.

B4. [Check lower bound.] If Q $+$ SIZE(Q) $=$ P0 [we assume that

$$\texttt{SIZE(LOC(AVAIL))} = 0,$$

so that this test always fails when Q $=$ LOC(AVAIL)], set SIZE(Q) $\leftarrow$ SIZE(Q) $+$ N and LINK(Q) $\leftarrow$ LINK(P0). Otherwise set LINK(Q) $\leftarrow$ P0, SIZE(P0) $\leftarrow$ N. ▮

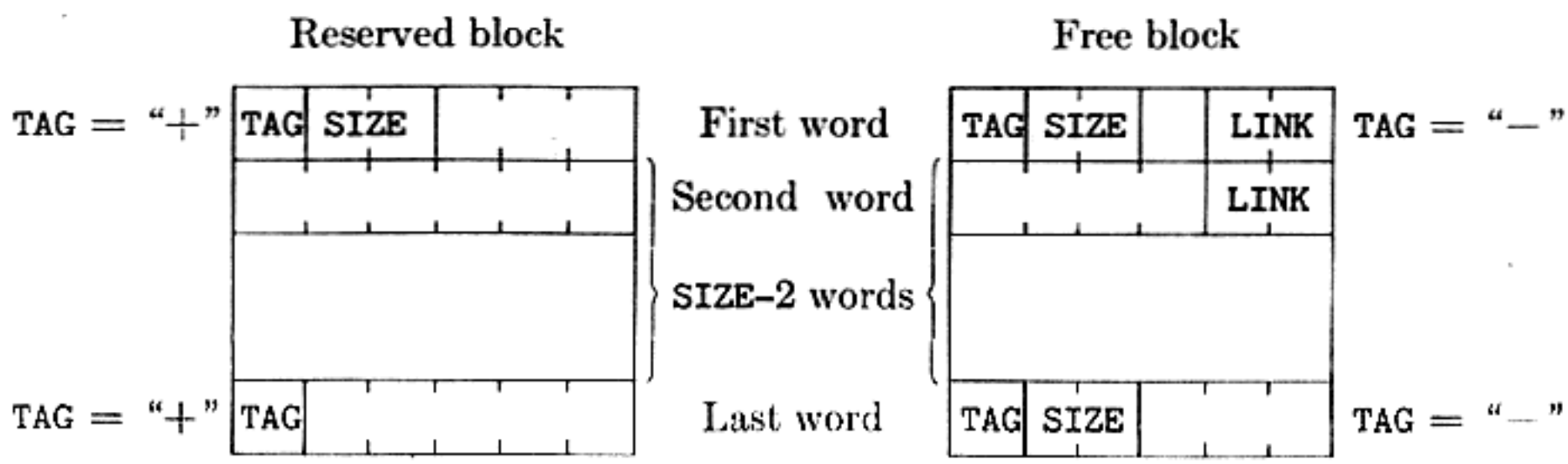
Steps B3 and B4 do the desired collapsing, based on the fact that Q $<$ P0 $<$ P are the beginning locations of three consecutive available areas.

If the AVAIL list is not maintained in order of locations, the reader can see that a "brute force" approach to the collapsing problem would require a complete search through the entire AVAIL list; Algorithm B reduces this to a search through about *half* of the AVAIL list (in step B2) on the average. Exercise 11 shows how Algorithm B can be modified so that, on the average, only about

one-third of the AVAIL list must be searched. But obviously, when the AVAIL list is long, all of these methods are much slower than we want them to be. Isn't there some way to reserve and free storage areas so that we don't need to do extensive searching through the AVAIL list?

We will now consider a method which eliminates all searching when storage is returned and which can be modified, as in exercise 6, to avoid almost all of the searching when storage is reserved. This technique makes use of a TAG field at both ends of each block, and a SIZE field in the first word of each block; this "overhead" is negligible when reasonably large size blocks are being used, although it is perhaps too much of a penalty to pay in situations when the blocks have a very small average size. Another method described in exercise 19 requires only one bit in the first word of each block, at the expense of a little more running time and a slightly more complicated program.

At any rate, let us now assume that we don't mind adding a little bit of control information, in order to save a good deal of time over Algorithm B when the AVAIL list is long. The method we will describe assumes each block has the following form:



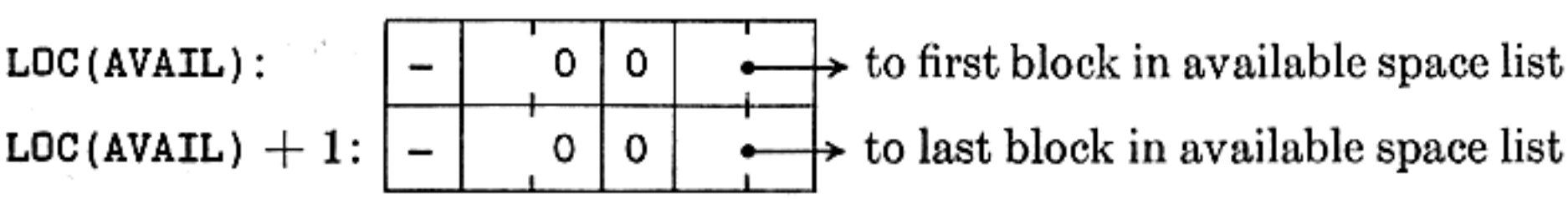
(7)

The idea in the following algorithm is to maintain a doubly linked AVAIL list, so that entries may conveniently be deleted from random parts of the list. The TAG field at either end of a block can be used to control the collapsing process, since we can tell easily whether or not both adjacent blocks are available.

Double linking is achieved in a familiar way, by letting the LINK in the first word point to the next free block in the list, and letting the LINK in the second word point back to the previous block; thus, if P is the address of an available block, we always have

$$\texttt{LINK(LINK(P)} + 1) = \texttt{P} = \texttt{LINK(LINK(P} + 1)). \tag{8}$$

To ensure proper "boundary conditions," AVAIL and the following location are set up as follows:



(9)

A "first-fit" reservation algorithm for this technique may be designed very much like Algorithm A, so we will not give it here (cf. exercise 12). The principal new feature of this method is the way a block can be freed in essentially a fixed amount of time:

Algorithm C (*Liberation with boundary tags*). Assume that blocks of locations have the forms shown in (7), and assume that the AVAIL list is doubly linked, as described above. This algorithm puts the block of locations starting with address P0 into the AVAIL list. If the pool of available storage runs from locations m_0 through m_1, inclusive, the algorithm assumes for convenience that

$$\texttt{TAG}(m_0 - 1) = \texttt{TAG}(m_1 + 1) = \text{“}+\text{”}.$$

C1. [Check lower bound.] If TAG(P0 − 1) = "+", go to C3.

C2. [Delete lower area.] Set P ← P0 − SIZE(P0 − 1), and then set

$$\texttt{P1} \leftarrow \texttt{LINK(P)},\quad \texttt{P2} \leftarrow \texttt{LINK(P + 1)},\quad \texttt{LINK(P1 + 1)} \leftarrow \texttt{P2},$$
$$\texttt{LINK(P2)} \leftarrow \texttt{P1},\quad \texttt{SIZE(P)} \leftarrow \texttt{SIZE(P)} + \texttt{SIZE(P0)},\quad \texttt{P0} \leftarrow \texttt{P}.$$

C3. [Check upper bound.] Set P ← P0 + SIZE(P0). If TAG(P) = "+", go to C5.

C4. [Delete upper area.] Set

$$\texttt{P1} \leftarrow \texttt{LINK(P)},\quad \texttt{P2} \leftarrow \texttt{LINK(P + 1)},\quad \texttt{LINK(P1 + 1)} \leftarrow \texttt{P2},$$
$$\texttt{LINK(P2)} \leftarrow \texttt{P1},\quad \texttt{SIZE(P0)} \leftarrow \texttt{SIZE(P0)} + \texttt{SIZE(P)},\quad \texttt{P} \leftarrow \texttt{P} + \texttt{SIZE(P)}.$$

C5. [Add to AVAIL list.] Set

$$\texttt{SIZE(P − 1)} \leftarrow \texttt{SIZE(P0)},\quad \texttt{LINK(P0)} \leftarrow \texttt{LINK(AVAIL)},$$
$$\texttt{LINK(P0 + 1)} \leftarrow \texttt{LOC(AVAIL)},\quad \texttt{LINK(LINK(AVAIL) + 1)} \leftarrow \texttt{P0},$$
$$\texttt{LINK(AVAIL)} \leftarrow \texttt{P0},\quad \texttt{TAG(P0)} \leftarrow \texttt{TAG(P − 1)} \leftarrow \text{“}-\text{”}. \ \blacksquare$$

The steps of Algorithm C are straightforward consequences of the storage layout (7); a slightly longer algorithm which is a little faster appears in exercise 15.

C. The "buddy system." We will now describe another approach to dynamic storage allocation, suitable for use with binary computers. This method takes one bit of "overhead" in each block, and it requires all blocks to be of length 1, 2, 4, 8, or 16, etc. If a block is not 2^k words long for some integer k, the next higher power of 2 is chosen and extra unused space is allocated accordingly. When this method is applicable it has an advantage of speed, especially in "real-time" situations, since there is a positive guarantee against searches through a long AVAIL list.

The idea of this method is to keep separate lists of available blocks of each size 2^k, $0 \le k \le m$. The entire pool of memory space under allocation consists

of 2^m words, which we will assume for convenience have the addresses 0 through $2^m - 1$. Originally, the entire block of 2^m words is available. Later, when a block of 2^k words is desired, and if none of this size are available, a larger available block is *split* into two equal parts; ultimately, a block of the right size 2^k will appear. When one block splits into two (each of which is half as large as the original), these two blocks are called *buddies*. Later when both buddies are available again, they coalesce back into a single block; thus the process can be maintained indefinitely (unless we run out of space at some point).

The key fact underlying the practical usefulness of this method is that if we know the address of a block (i.e., the memory location of its first word), and if we also know the size of that block, we know the address of its buddy. For example, the buddy of the block of size 16 beginning in binary location 101110010110000 is a block starting in binary location 101110010100000. To see why this must be true, we first observe that as the algorithm proceeds, *the address of a block of size 2^k is a multiple of 2^k*. In other words, the address in binary notation has at least k zeros at the right. This observation is easily justified by induction: if it is true for all blocks of size 2^{k+1}, it is certainly true when such a block is halved.

Therefore a block of size, say, 32 has an address of the form $xx\ldots x00000$ (where the x's represent either 0 or 1); if it is split, the newly formed buddy blocks have the addresses $xx\ldots x00000$ and $xx\ldots x10000$. In general, let $\text{buddy}_k(x)$ = address of the buddy of the block of size 2^k whose address is x; we find that

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, & \text{if} \quad x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{if} \quad x \bmod 2^{k+1} = 2^k. \end{cases} \tag{10}$$

This function is readily computed with the "exclusive or" instruction (sometimes called "selective complement" or "add without carry") usually found on binary computers; cf. exercise 28.

The buddy system makes use of a one-bit TAG field in each block:

$$\begin{aligned} &\texttt{TAG(P)} = 0, && \text{if the block with address P is reserved;} \\ &\texttt{TAG(P)} = 1, && \text{if the block with address P is available.} \end{aligned} \tag{11}$$

Besides this TAG field, which is present in all blocks, *available* blocks also have two link fields, LINKF and LINKB, which are the usual forward and backward links of a doubly linked list; and they also have a KVAL field to specify k when their size is 2^k. The algorithms below make use of the table locations AVAIL[0], AVAIL[1], . . . , AVAIL[m], which serve respectively as the heads of the lists of available storage of sizes 1, 2, 4, . . . , 2^m. These lists are doubly linked, so as usual (see Section 2.2.5) the list heads contain two pointers:

$$\begin{aligned} &\texttt{AVAILF[}k\texttt{]} = \texttt{LINKF(LOC(AVAIL[}k\texttt{]))} = \text{link to rear of AVAIL[}k\text{] list;} \\ &\texttt{AVAILB[}k\texttt{]} = \texttt{LINKB(LOC(AVAIL[}k\texttt{]))} = \text{link to front of AVAIL[}k\text{] list.} \end{aligned} \tag{12}$$

Initially, before any storage has been allocated, we have

$$\begin{aligned} \texttt{AVAILF}[m] &= \texttt{AVAILB}[m] = 0, \\ \texttt{LINKF}(0) &= \texttt{LINKB}(0) = \texttt{LOC}(\texttt{AVAIL}[m]), \\ \texttt{TAG}(0) &= 1,\ \texttt{KVAL}(0) = m \end{aligned} \tag{13}$$

(indicating a single available block of length 2^m, beginning in location 0), and also

$$\texttt{AVAILF}[k] = \texttt{AVAILB}[k] = \texttt{LOC}(\texttt{AVAIL}[k]), \qquad \text{for} \qquad 0 \le k < m \tag{14}$$

(indicating empty lists for available blocks of lengths 2^k for all $k < m$).

From this description of the buddy system, the reader may find it enjoyable to design the necessary algorithms for reserving and freeing storage areas by himself, and to compare his solutions with the algorithms given below. Note the comparative ease with which blocks can be halved in the reservation algorithm.

Algorithm R (*Buddy system reservation*). This algorithm finds and reserves a block of 2^k locations, or reports failure, using the organization of the buddy system as explained above.

R1. [Find block.] Let j be the smallest integer in the range $k \le j \le m$ for which $\texttt{AVAILF}[j] \ne \texttt{LOC}(\texttt{AVAIL}[j])$, that is, for which the list of available blocks of size 2^j is not empty. If no such j exists, the algorithm terminates unsuccessfully, since there are no known available blocks of sufficient size to meet the request.

R2. [Remove from list.] Set

$$\texttt{L} \leftarrow \texttt{AVAILF}[j], \qquad \texttt{AVAILF}[j] \leftarrow \texttt{LINKF}(\texttt{L}),$$
$$\texttt{LINKB}(\texttt{LINKF}(\texttt{L})) \leftarrow \texttt{LOC}(\texttt{AVAIL}[j]), \qquad \text{and} \qquad \texttt{TAG}(\texttt{L}) \leftarrow 0.$$

R3. [Split required?] If $j = k$, the algorithm terminates (we have found and reserved an available block starting at address L).

R4. [Split.] Decrease j by 1. Then set

$$\texttt{P} \leftarrow \texttt{L} + 2^j, \quad \texttt{TAG}(\texttt{P}) \leftarrow 1, \quad \texttt{KVAL}(\texttt{P}) \leftarrow j, \quad \texttt{LINKF}(\texttt{P}) \leftarrow \texttt{LOC}(\texttt{AVAIL}[j]),$$
$$\texttt{LINKB}(\texttt{P}) \leftarrow \texttt{LOC}(\texttt{AVAIL}[j]), \qquad \texttt{AVAILF}[j] \leftarrow \texttt{AVAILB}[j] \leftarrow \texttt{P}.$$

(This splits a large block and enters the unused half in the $\texttt{AVAIL}[j]$ list which was empty.) Go back to step R3. ∎

Algorithm S (*Buddy system liberation*). This algorithm returns a block of 2^k locations starting in address L to free storage, using the organization of the buddy system as explained above.

S1. [Is buddy available?] Set $\texttt{P} \leftarrow \text{buddy}_k(\texttt{L})$. (See Eq. 10.) If $k = m$ or if $\texttt{TAG}(\texttt{P}) = 0$, or if $\texttt{TAG}(\texttt{P}) = 1$ and $\texttt{KVAL}(\texttt{P}) \ne k$, go to S3.

S2. [Combine with buddy.] Set

$$\texttt{LINKF(LINKB(P))} \leftarrow \texttt{LINKF(P)}, \qquad \texttt{LINKB(LINKF(P))} \leftarrow \texttt{LINKB(P)}.$$

(This removes block P from the AVAIL[k] list.) Then set $k \leftarrow k + 1$, and if P $<$ L set L $\leftarrow$ P. Return to S1.

S3. [Put on list.] Set

$$\texttt{TAG(L)} \leftarrow 1, \qquad \texttt{LINKF(L)} \leftarrow \texttt{AVAILF[}k\texttt{]}, \qquad \texttt{LINKB(AVAILF[}k\texttt{])} \leftarrow \texttt{L},$$
$$\texttt{KVAL(L)} \leftarrow k, \qquad \texttt{LINKB(L)} \leftarrow \texttt{LOC(AVAIL[}k\texttt{])}, \qquad \texttt{AVAILF[}k\texttt{]} \leftarrow \texttt{L}.$$

(This puts block L on the AVAIL[k] list.) ▮

D. Comparison of the methods. Hardly any mathematical analysis of these dynamic storage-allocation algorithms has been carried out so far; the only result which we can prove is an interesting phenomenon, the "fifty-percent rule":

If Algorithms A and B are used continually in such a way that the system tends to an equilibrium condition, where there are N reserved blocks in the system, on the average, each with an independent lifetime, and where the quantity K *in Algorithm A takes on nonzero values (or, more generally, values $\geq c$ as in step A4′) with probability p, then the average number of available blocks tends to approximately $\frac{1}{2}pN$.*

This rule tells us approximately how long the AVAIL list will be. When the quantity p is near 1—this will happen if c is very small and if the block sizes are infrequently equal to each other—we have about half as many available blocks as unavailable ones; hence the name "fifty-percent rule."

It is not hard to derive this rule. Consider the following memory map:

This shows the reserved blocks divided into three categories:

A: when freed, the number of available blocks will decrease by one;
B: when freed, the number of available blocks will not change;
C: when freed, the number of available blocks will increase by one.

Now let N be the number of reserved blocks, and let M be the number of available ones; let A, B, and C be the number of blocks of the types identified above. We have

$$\begin{aligned} N &= A + B + C \\ M &= \tfrac{1}{2}(2A + B + \epsilon) \end{aligned} \tag{15}$$

where $\epsilon = 0$, 1, or 2 depending on conditions at the lower and upper boundaries. To derive the fifty-percent rule, we set

probability that M increases by one = probability that M decreases by one

(or, more precisely, the average change in M per unit time is set to zero during equilibrium). This leads to

$$C = A + (1 - p)N;$$

and by (15), with ϵ assumed to be zero (since M and N are assumed to be reasonably large), we get

$$N - 2M + A = A + (1 - p)N. \tag{16}$$

The fifty-percent rule follows. This derivation shows in fact that when M is momentarily less than $\frac{1}{2}pN$, there is higher probability that M will increase than that it will decrease, and conversely.

Besides this interesting rule, our knowledge of the performance of these algorithms is based solely on Monte Carlo experiments. The reader will find it instructive to conduct his own simulation experiments when he is choosing between storage allocation algorithms for a particular machine and a particular application or class of applications. The author carried out several such experiments just before writing this section (and, indeed, the fifty-percent rule was noticed during these experiments before a proof for it was found); let us briefly examine the methods and results of these experiments here.

The basic simulation program ran as follows, with TIME initially zero and with the memory area initially all available:

P1. Advance TIME by 1.

P2. Free all blocks in the system which are scheduled to be freed at the current value of TIME.

P3. Calculate two quantities S (a random size) and T (a random "lifetime"), based on some probability distributions, using the methods of Chapter 3.

P4. Reserve a new block of length S, which is due to be freed at (TIME + T). Return to P1. ▮

Whenever TIME was a multiple of 200, detailed statistics about the performances of the reservation and liberation algorithms were printed. The same sequence of values of S and T was used for each pair of algorithms tested. After TIME advanced past 2000, the system usually had reached a more or less steady state which gave every indication of being maintained indefinitely thereafter. However, depending on the total amount of storage available and on the distributions of S and T in step P3, the allocation algorithms would occasionally fail to find enough space and the simulation experiment was then terminated.

Let M be the total number of memory locations available, and let $\overline{S}$, $\overline{T}$ denote the average values of S and T in step P3. It is easy to see that the expected number of unavailable words of memory at any given time is $\overline{S} \cdot \overline{T}$, once TIME is sufficiently large. When $\overline{S} \cdot \overline{T}$ was greater than about $\frac{2}{3}M$ in the

experiments, memory overflow usually occurred, often before M words of memory were actually needed. The memory was able to become over 90 percent filled when the block size was small compared to M, but when the block sizes were allowed to exceed $\frac{1}{3}M$ (as well as taking on much smaller values) the memory tended to become "full" when less than $\frac{1}{2}M$ locations were in fact needed. Empirical evidence suggests strongly that *block sizes larger than* $\frac{1}{10}M$ *should not be used with dynamic storage allocation* if effective operation is expected.

The experiments were conducted with three size distributions for S:

(*S1*) An integer chosen uniformly between 100 and 2000;
(*S2*) Sizes (1, 2, 4, 8, 16, 32) chosen with respective probabilities ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{32}$);
(*S3*) Sizes (10, 12, 14, 16, 18, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 500, 1000, 2000, 3000, 4000) were selected with equal probability.

The time distribution T was usually a random integer chosen uniformly between 1 and t, for fixed $t = 10$, 100, or 1000.

Experiments were also made by choosing T uniformly between 1 and min ($\lfloor \frac{5}{4}U \rfloor$, 12500) in step P3, where U is the number of time units remaining until the first reserved block now in the system is scheduled to be freed. This time distribution was meant to simulate an "almost-last-in-first-out" behavior: for if T were always chosen $\leq U$, the storage allocation system would degenerate into simply a stack operation where no complex algorithms are necessary. (See exercise 1.) In this case, T is chosen greater than U about 20 percent of the time, so we have almost, but not quite, a stack operation. When this distribution was used, algorithms such as A, B, and C behaved much better than usual; there were rarely, if ever, more than two items in the entire AVAIL list, while there were about 14 reserved blocks. On the other hand, the buddy system algorithms R and S were slower when this distribution was used, because it is necessary to split and to coalesce blocks more frequently in a stack-like operation. The theoretical properties of this time distribution appear to be very difficult to deduce (see exercise 32).

Figure 42, which appeared near the beginning of this section, was the configuration of memory at TIME = 5000, with size distribution (*S1*) and with the time distribution chosen randomly between 1 and 100, using the "first-fit" method just as in Algorithms A and B above. For this experiment, the probability p which enters into the "fifty-percent rule" was essentially 1, so we would expect about half as many available blocks as reserved blocks. Actually Fig. 42 shows 21 available and 53 reserved. This does not disprove the fifty-percent rule: for example, at TIME = 4600 there were 25 available and 49 reserved. The configuration in Fig. 42 merely shows how the fifty-percent rule is subject to statistical variations. The number of available blocks generally ranged between 20 and 30, while the number of reserved blocks was generally between 45 and 55.

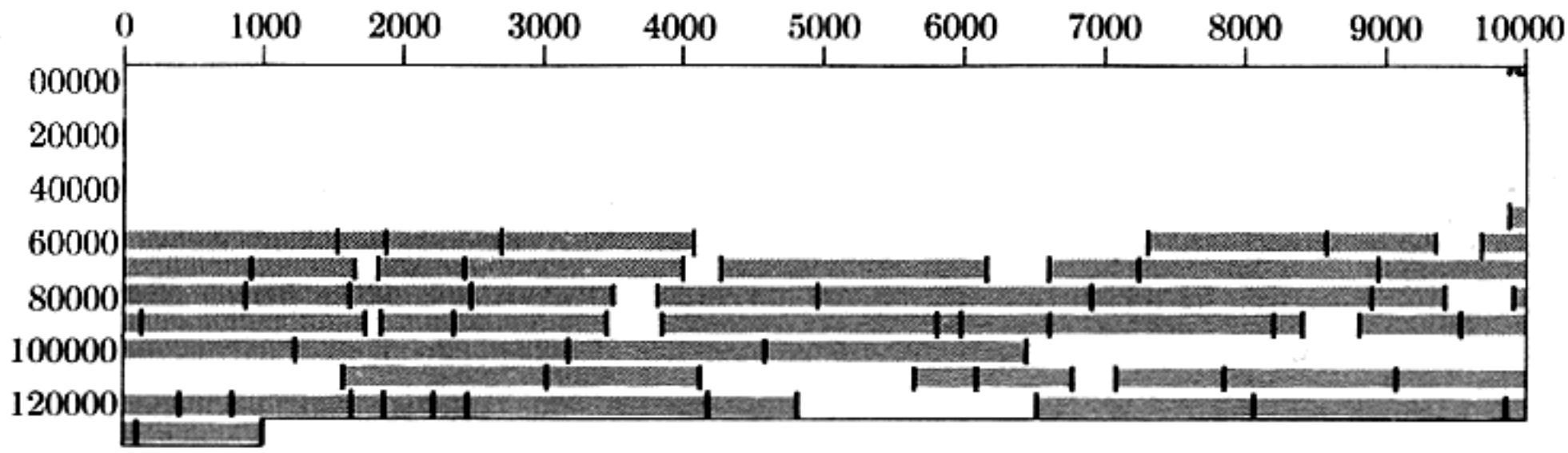


Fig. 43. Memory map obtained with the "best fit" method. (Compare this with Fig. 42, which shows the "first fit" method, and Fig. 44, which shows the "buddy system," for the same sequence of storage requests.)

Figure 43 shows the configuration of memory obtained with *the same data as Fig. 42* but with the "best-fit" method used instead of the "first-fit" method. The constant c in step A4′ was chosen as 16, to eliminate small blocks, and as a result the probability p dropped to about 0.7 and there were fewer available areas.

When the time distribution was changed from 1 to 1000 instead of 1 to 100, situations precisely analogous to those shown in Figs. 42 and 43 were obtained, with all appropriate quantities approximately multiplied by 10. For example, there were 515 reserved blocks; and 240 free blocks in the equivalent of Fig. 42, 176 free blocks in the equivalent of Fig. 43.

In all experiments comparing the best-fit and first-fit methods, the latter always appeared to be superior. When memory size was exhausted, the first-fit method actually stayed in action longer than the best-fit method before memory overflow occurred, in most instances.

The buddy system was also applied to the same data that led to Figs. 42 and 43, and Fig. 44 was the result. Here, all sizes in the range 257 to 512 were treated as 512, those between 513 and 1024 were raised to 1024, etc. On the average this means about four thirds as much memory was requested (see exercise 21); the buddy system, of course, works better on size distributions like that of (*S2*) above, instead of (*S1*). Note that there are available blocks of sizes 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, and 2^{14} in Fig. 44.

Simulation of the buddy system showed that it performs much better than first expected. It is clear that the buddy system will sometimes allow two adjacent areas to be available without merging them into one (if they are not "buddies"); but this situation is not present in Fig. 44 and, in fact, it is rare in practice. In cases where memory overflow occurred, memory was 95 percent packed, and this reflects a surprisingly good allocation balance. Furthermore, it was very seldom necessary to split blocks in Algorithm R, or to merge them in Algorithm S; the tree remained much like Fig. 44 with available blocks on the most commonly used levels. It would be most interesting to see mathematical results which account for this behavior.

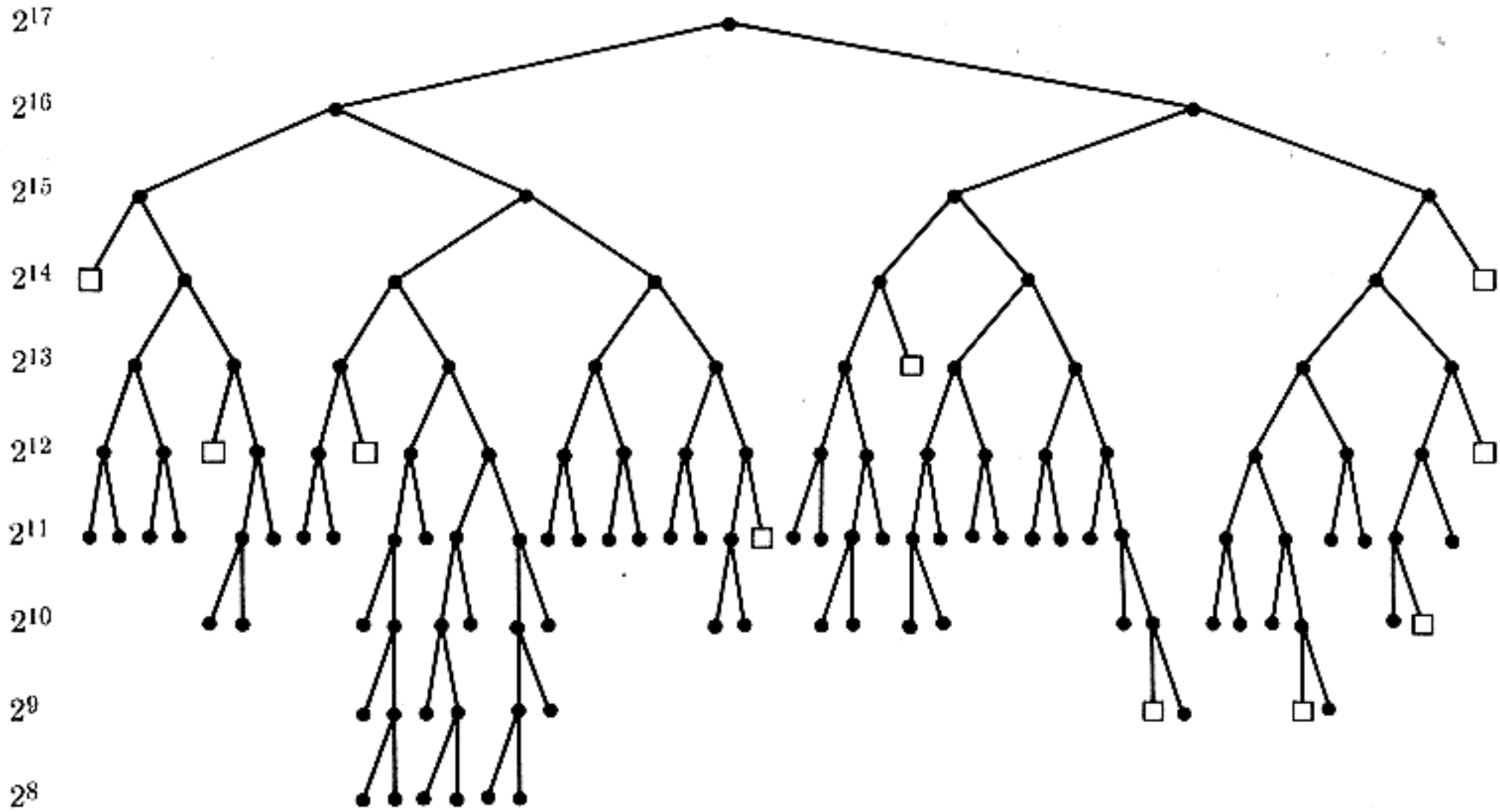


Fig. 44. Memory map obtained with the "buddy system." (The tree structure indicates the division of certain large blocks into "buddies" of half the size. Squares indicate available blocks.)

Another surprise was the excellent behavior of Algorithm A after the modification described in exercise 6; only 2.8 inspections of available block sizes were necessary on the average [using size distribution (*S1*) and times chosen uniformly between 1 and 1000], and more than half of the time only the minimum value, one iteration, was necessary. This was true in spite of the fact that about 250 available blocks were present. The same experiment with Algorithm A unmodified showed about 125 iterations were necessary on the average (so about half of the AVAIL list was being examined each time), and 20 percent of the time 200 or more interations were found to be necessary! This greatly increased efficiency was unexpected since the idea came as an afterthought while examining the output of earlier simulations. [*Note:* Algorithm A unmodified was not nearly so bad on size distribution (*S2*); only six iterations were needed on the average, even when 500 reserved blocks were present. In large part, this is due to the frequent requests for blocks of size 1 and the relatively short AVAIL list.]

The exercises below include MIX programs for the two principal algorithms which are recommended as a consequence of the above remarks—Algorithm A modified as in exercise 12 together with Algorithm C, as compared with the buddy system—and here are the approximate results:

	Time for reservation	Time for liberation
Boundary tag system:	$33 + 7A$	18, 29, 31, or 34
Buddy system:	$19 + 25R$	$27 + 26S$

Here $A \geq 1$ is the number of iterations necessary in searching for an available block which is large enough; $R \geq 0$ is the number of times a block is split in two (the initial difference of $j - k$ in Algorithm R); and $S \geq 0$ is the number of times buddy blocks are reunited during Algorithm S. The simulation experiments indicate that under the stated assumptions with size distribution (*S1*) and time chosen between 1 and 1000, we may take $A = 2.8$, $R = S = 0.04$ on the average. (The average values $A = 1.3$, $R = S = 0.9$ were observed when the "almost-last-in-first-out" time distribution was substituted as explained above.) This shows that both methods are quite fast, with the buddy system slightly faster in MIX's case. Remember that the buddy system requires about 40 percent more memory space when block sizes are not constrained to be powers of 2.

A corresponding time estimate for the garbage collection and compacting algorithm of exercise 33 is about 104 units of time to locate a free node, assuming that garbage collection occurs when the memory is approximately half full, and assuming that the nodes have an average length of 5 words with 2 links per node. The pros and cons of garbage collection are discussed in Section 2.3.5. When the memory is not heavily loaded and when the appropriate restrictions are met, garbage collection and compacting is very efficient; for example, on the MIX computer, the garbage collection method is faster than the other two, if the memory space never gets more than about one-third full, and if the nodes are relatively small.

The same simulation techniques were applied also to some other storage allocation algorithms. The other algorithms were so poor by comparison with the algorithms of this section that they will be given only brief mention here:

a) Separate AVAIL lists were kept for each size. A single free block was occasionally split into two smaller blocks when necessary, but no attempt was made to put such blocks together again. The memory map became fragmented into finer and finer parts until it was in terrible shape; a simple scheme like this might work when only two or three block sizes are required, but it is essentially equivalent to doing separate allocation in disjoint areas, one area for each block (node) size.

b) An attempt was made to do "two-level" allocation: The memory was divided into 32 large sectors. A brute-force allocation method was used to reserve large blocks of 1, 2, or 3 (rarely more) adjacent sectors; each large block such as this was subdivided to meet storage requests until no more room was left within the current large block, and then another large block was reserved for use in subsequent allocations. Each large block was returned to free storage only when *all* space within it became available. This method almost always ran out of storage space very quickly.

Although this particular method of "two level" allocation was a failure for the data considered in the author's simulation experiments, there are other circumstances (which occur not infrequently in practice) when a multiple-level

allocation strategy can be beneficial. For example, this occurs with rather large programs that operate in several stages, where it is known that certain types of nodes are needed only within a certain subroutine. It might also be desirable to use quite different allocation strategies for different classes of nodes in the same program. The idea of allocating storage by "zones," with possibly different strategies employed in each zone and with the ability to free an entire zone at once, is discussed by Douglas T. Ross in his article, "The AED Free Storage Package," *CACM* **10** (1967), 481–492.

E. Overflow. What do we do when no more room is available? Suppose there is a request for, say, n consecutive words, when all available blocks are too small. The first time this happens, there are usually more than n available locations present, but they are not consecutive; "compacting memory" (i.e., moving some of the locations which are in use, so that all the available locations are brought together) would mean we could continue processing. But compacting is slow; and the vast majority of cases in which the "first-fit" method runs out of room actually would soon thereafter run completely out of space anyway, no matter how much compacting and re-compacting is done. Therefore it is generally not worth while to write a compacting program, except under special circumstances in connection with garbage collection, as in exercise 33. If overflow is expected to occur, some method for removing items from memory and storing them on an external memory device can be used, with provision for bringing the information back again when it is needed. This implies that all programs referring to the dynamic memory area must be severely restricted with regard to the allowable references they make to other blocks, and special computer hardware (e.g., interrupt on absence of data, or automatic "paging") is generally required for efficient operation under these conditions.

Some decision procedure is necessary to decide which blocks are the most likely candidates for removal. One idea is to use a memory cell as an artificial "timer": the contents of this cell is increased by one periodically as a program is executed, and although it does not keep strict units of time it is possible to tell that one event has occurred before another if its "time" is smaller. When a block is accessed, the current timer reading is stored in the block, and thus an algorithm can be devised to place those blocks that are not recently active into the auxiliary memory. Instead of such a timer cell, the same effect can be achieved with a doubly linked list of reserved blocks, in which a block is moved up to the front of the list each time it is accessed; then the blocks are effectively sorted in order of their last access, and the block at the rear of the list is the one to remove first.

Further discussion of this problem belongs under the heading of executive control programs ("operating systems"), and so it is beyond the scope of this book. For most applications of dynamic storage allocation, let us hope there is plenty of memory space available.

EXERCISES

1. [*20*] What simplifications can be made to the reservation and liberation algorithms of this section, if storage requests always appear in a "last-in-first-out" manner, i.e., if no reserved block is freed until after all blocks that were reserved subsequently have already been freed?

2. [*HM23*] (E. Wolman.) Suppose that we want to choose a fixed node size for variable length items, and suppose also that when each node has length k and when an item has length l, it takes $\lceil l/(k-b) \rceil$ nodes to store this item. (Here b is a constant, signifying that b words of each node contain control information, such as a link to the next node.) If the average length l of a record is L, what choice of k minimizes the average amount of storage space required? (Assume that the average value of $(l/(k-b)) \bmod 1$ is equal to $\frac{1}{2}$, for any fixed k, as l varies.)

3. [*40*] By computer simulation, compare the best-fit, first-fit, and "worst-fit" methods of storage allocation; in the latter method, the largest available block is always chosen. Is there any significant difference in the memory usage?

4. [*22*] Write a MIX program for Algorithm A, paying special attention to making the inner loop fast. Assume that the SIZE field is (4:5), the LINK field is (0:2), and $A < 0$.

▶ **5.** [*18*] Suppose it is known that N is always 100 or more in Algorithm A. Would it be a good idea to set $c = 100$ in the modified step A4′?

▶ **6.** [*23*] After Algorithm A has been used repeatedly, there will be a strong tendency for blocks of small SIZE to remain at the front of the AVAIL list, so that it will often be necessary to search quite far into the list before finding a block of length N or more. For example, note how the size of the blocks essentially increases in Fig. 42, for both reserved and free blocks, from the beginning of memory to the end. (The AVAIL list used while Fig. 42 was being prepared was kept sorted by order of location, as required by Algorithm B.) Can you suggest a way to modify Algorithm A so that (a) short blocks won't tend to accumulate in a particular area, and (b) the AVAIL list may still be kept in order of increasing memory locations, for purposes of algorithms like Algorithm B?

7. [*10*] The example (1) shows that sometimes "first fit" can definitely be superior to "best fit." Give a similar example which shows a case where "best fit" is superior to "first fit."

8. [*21*] Show how to modify Algorithm A in a simple way to obtain an algorithm for the "best-fit" method, instead of "first fit."

9. [*26*] In what ways could a reservation algorithm be designed using the "best-fit" method, that avoids searching the whole AVAIL list? (Try to think of ways that cut down the necessary search as much as possible.)

10. [*22*] Show how to modify Algorithm B so that the block of N consecutive cells beginning at location P0 is made available, without assuming that none of these N cells is already available; assume, in fact, that the area being freed may actually overlap several blocks that are already free.

11. [*M25*] Show that the improvement to Algorithm A suggested in the answer to exercise 6 also can be used to lead to a slight improvement in Algorithm B, which cuts

the average length of search from half the length of the AVAIL list to one-third this length. (Assume that the block being freed will be inserted into a random place within the sorted AVAIL list.)

▶ **12.** [*20*] Modify Algorithm A so that it follows the conventions of (7), uses the modified step A4′ described in the text, and also incorporates the improvement of exercise 6.

13. [*21*] Write a MIX program for the algorithm of exercise 12.

14. [*21*] What difference would it make to Algorithm C and the algorithm of exercise 12, (a) if the SIZE field were not present in the last word of a free block? or (b) if the SIZE field were not present in the first word of a reserved block?

▶ **15.** [*24*] Show how to speed up Algorithm C at the expense of a slightly longer program, by not changing any more links than absolutely necessary in each of four cases depending on whether TAG(P0−1), TAG(P0 + SIZE(P0)) are plus or minus.

16. [*24*] Write a MIX program for Algorithm C, incorporating the ideas of exercise 15.

17. [*10*] What should be the contents of LOC(AVAIL) and LOC(AVAIL) + 1 in (9) when there are no available blocks present?

▶ **18.** [*20*] Figs. 42 and 43 were obtained using the same data, and essentially the same algorithms (Algorithms A and B), except that Fig. 43 was prepared by modifying Algorithm A to choose "best fit" instead of "first fit." Why did this cause Fig. 42 to have a large available area in the *higher* locations of memory, while in Fig. 43 there is a large available area in the *lower* locations?

▶ **19.** [*24*] Suppose that blocks of memory have the form of (7), except without the TAG or SIZE fields required in the last word of the block. Suppose further that the following simple algorithm is being used to make a reserved block free again: "Q ← LINK(AVAIL), LINK(P0) ← Q, LINK(P0+1) ← LOC(AVAIL), LINK(Q+1) ← P0, LINK(AVAIL) ← P0, TAG(P0) ← "−"." (This algorithm does nothing about collapsing adjacent areas together.)

Show that it is possible to design a reservation algorithm similar to Algorithm A, which does the necessary collapsing of adjacent free blocks while searching the AVAIL list, and at the same time it avoids any unnecessary fragmentation of memory as in (2), (3), and (4).

20. [*00*] Why is it desirable to have the AVAIL[*k*] lists in the buddy system doubly linked, instead of simply having straight linear lists?

21. [*HM25*] Examine the ratio a_n/b_n, where a_n is the sum of the first n terms of $1 + 2 + 4 + 4 + 8 + 8 + 8 + 8 + 16 + 16 + \cdots$, and b_n is the sum of the first n terms of $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + \cdots$, as n goes to infinity.

▶ **22.** [*21*] The text repeatedly states that the buddy system allows only blocks of size 2^k to be used, and exercise 21 shows this can lead to a substantial increase in the storage required. But if an 11-word block is needed in connection with the buddy system, why couldn't we find a 16-word block and divide it into an 11-word piece together with two free blocks of sizes 4 and 1?

23. [*05*] What is the binary address of the buddy of the block of size 4 whose binary address is 011011110000? What would it be if the block were of size 16 instead of 4?

24. [*20*] According to the algorithm in the text, the largest block (of size 2^m) has no buddy, since it represents all of storage. Would it be correct to define $\text{buddy}_m(0) = 0$

(i.e., to make this block its own buddy), and then to simplify step S1 to avoid testing the condition $k = m$?

▶ **25.** [*22*] Criticize the following idea: "Dynamic storage allocation using the buddy system will never reserve a block of size 2^m in practical situations (since this would fill the whole memory), and, in general, there is a maximum size 2^n for which no blocks of greater size are ever to be reserved. Therefore it is a waste of time to start with such large blocks available, and to combine buddies in Algorithm S when the combined block has a size larger than 2^n."

▶ **26.** [*21*] Explain how the buddy system could be used for dynamic storage allocation in memory locations 0 through M − 1 even though M does not have the form 2^m as required in the text.

27. [*24*] Write a MIX program for Algorithm R, and determine its running time.

28. [*25*] Assume that MIX is a binary computer, with a new operation code XOR defined as follows (using the notation of Section 1.3.1): "C = 5, F = 5; for each bit position in location M which equals 1, the corresponding bit position in register A is complemented (changed from 0 to 1 or 1 to 0)."

Write a MIX program for Algorithm S, and determine its running time.

29. [*20*] Could the buddy system be modified to avoid the tag bit in each reserved block?

30. [*M50*] Analyze the properties of the buddy system, in particular the average speed of Algorithms R and S, given reasonable distributions for the sequence of storage requests.

31. [*M40*] Can a storage allocation system analogous to the buddy system be designed using the Fibonacci sequence instead of powers of two? (Thus, we might start with F_m available words, and split an available block of F_k words into two buddies of respective lengths F_{k-1} and F_{k-2}.)

32. [*HM50*] Determine $\lim_{n\to\infty} \alpha_n$, if it exists, where α_n is the mean value of t_n in a sequence defined as follows: Let

$$g_k = \lfloor \tfrac{5}{4} \min\left(10000, f(t_{k-1} - 1), f(t_{k-2} - 2), \ldots, f(t_1 - (k-1))\right) \rfloor,$$

where $f(x) = x$ if $x > 0$, $f(x) = \infty$ if $x \le 0$. The quantity t_k takes on any of the values $1, 2, \ldots, g_k$ with probability $1/g_k$. (*Note:* Some limited empirical tests indicate that α_n will be approximately 14, but this is not guaranteed to be very accurate.)

33. [*28*] (Garbage collection and compacting.) Assume that memory locations 1 through AVAIL − 1 are being used as a storage pool for nodes of varying sizes, having the following form: The first word of NODE(P) contains the fields

SIZE(P) = number of words in NODE(P);
T(P) = number of link fields in NODE(P); T(P) < SIZE(P);
LINK(P) = special link field for use only during garbage collection.

The node immediately following NODE(P) in memory is NODE(P + SIZE(P)). Assume that the only fields in NODE(P) which are used as links to other nodes are LINK(P + 1), LINK(P + 2), . . . , LINK(P + T(P)), and each of these link fields is either Λ or the address of the first word of another node. Finally, assume that there is one further link variable in the program, called USE, and it points to one of the nodes in the storage pool.

Design an algorithm which (a) determines all nodes accessible directly or indirectly from the variable USE, (b) moves these nodes into memory locations 1 through K − 1, for some K, changing all links so that structural relationships are preserved, and (c) sets AVAIL ← K.

For example, consider the following contents of memory, where INFO(L) denotes the contents of location L, excluding LINK(L):

```
 1:  SIZE = 2, T = 1
 2:  LINK = 6, INFO = A
 3:  SIZE = 3, T = 1
 4:  LINK = 8, INFO = B
 5:  CONTENTS = C
 6:  SIZE = 2, T = 0
 7:  CONTENTS = D
 8:  SIZE = 3, T = 2
 9:  LINK = 8, INFO = E
10:  LINK = 3, INFO = F
```

AVAIL = 11, USE = 3. Your algorithm should transform this into

```
1:  SIZE = 3, T = 1
2:  LINK = 4, INFO = B
3:  CONTENTS = C
4:  SIZE = 3, T = 2
5:  LINK = 4, INFO = E
6:  LINK = 1, INFO = F
```

AVAIL = 7, USE = 1.

34. [*29*] Write a MIX program for the algorithm of exercise 33, and determine its running time.

35. [*45*] Design an efficient, general-purpose system oriented towards the editing of fairly long sequences of characters.

36. [*22*] Contrast the dynamic storage allocation methods of this section with the techniques for variable-size sequential lists discussed at the end of Section 2.2.2.

▶ **37.** [*25*] ("Beat the allocator.") Assume that storage is being allocated dynamically by an algorithm that never relocates reserved blocks. Show that, no matter what algorithm is used, it is possible to have a series of requests which overflows a memory of $2n + 1$ cells, although no more than $n + 2$ cells are requested at any one time. (For example if $n = 2$, and if memory locations a_1, a_2, a_3, a_4, a_5 are available, we might first reserve four blocks of one cell each. Then if a_3 is unavailable, we liberate all other cells and request a block of 3 cells in a row. If a_3 is available, we liberate a_1 and a_5 and request a block of size 2. In both cases the request cannot be filled.)

Linear lists and rectangular arrays of information kept in consecutive memory locations were widely used from the earliest days of stored-program computers, and the earliest treatises on programming gave the basic algorithms for traversing these structures. [For example, see J. von Neumann, *Collected Works*, vol. V, 113–116 (written 1947); M. V. Wilkes, D. J. Wheeler, S. Gill, *The Preparation of Programs for an Electronic Digital Computer* (Reading, Mass.: Addison-Wesley, 1951), subroutine V–1.] Before the days of index registers, operations on sequential linear lists were done by performing arithmetic on the machine language instructions themselves, and this type of operation was one of the early motivations for having a computer whose programs share memory space with the data they manipulate.

Techniques which permit variable-length linear lists to share sequential locations, in such a way that they shift back and forth when necessary, as described in Section 2.2.2, were apparently a much later invention. J. Dunlap of Digitek Corporation developed these techniques in 1963 in connection with the design of a series of compiler programs; about the same time the idea independently appeared in the design of a COBOL compiler at IBM Corporation, and a collection of related subroutines called CITRUS was subsequently used at various installations. The techniques remained unpublished until after they had been independently developed by Jan Garwick of Norway; see *BIT* **4** (1964), 137–140.

The idea of having linear lists in *non*sequential locations seems to have originated in connection with the design of computers with drum memories; the first such computer was developed by General Electric Corporation during 1946–1948, and the IBM 650 was the most notable later machine of this type. After executing the instruction in location n, such a computer is usually not ready to get its next instruction from location $n + 1$ because the drum has already rotated past this point. Depending on the instruction being performed, the most favorable position for the next instruction might be $n + 7$ or $n + 18$, etc., and the machine can operate up to six or seven times faster if its instructions are optimally located rather than consecutive. [For a discussion of the interesting problems concerning best placement of these instructions, see the author's article in *JACM* **8** (1961), 119–150.] Therefore the machine design provides an extra address field in each machine language instruction, to serve as a link to the next instruction. Such a machine is called a "one-plus-one-address computer," as distinguished from MIX which is a "one-address computer." The design of one-plus-one-address computers is apparently the first appearance of the linked-list idea within computer programs, although the dynamic insertion and deletion operations which we have used so frequently in this chapter were still unknown.

Linked memory techniques were really born when A. Newell, C. Shaw, and H. Simon began their investigations of heuristic problem-solving by machine.

As an aid to writing programs that searched for proofs in mathematical logic, they designed the first "list-processing" language IPL-II in the spring of 1956. (IPL stands for *Information Processing Language.*) This was a system which made use of links and included important concepts like the list of available space, but the concept of stacks was not yet well developed; IPL-III was designed a year later, and it included "push down" and "pop up" for stacks as important basic operations. (For references to IPL-II see "The Logic Theory Machine," *IRE Transactions on Information Theory* **IT-2** (Sept. 1956), 61–70; see also *Proc. Western Joint Comp. Conf.* (Feb. 1957), 218–240. Material on IPL-III first appeared in course notes given at the University of Michigan in the summer of 1957.)

The work of Newell, Shaw, and Simon inspired many other people to use linked memory (which was often at the time referred to as NSS memory), mostly for problems dealing with simulation of human thought processes. Gradually, the techniques became recognized as basic computer-programming tools; the first article describing the usefulness of linked memory for "down-to-earth" problems was published by J. W. Carr, III, in *CACM* **2** (Feb. 1959), 4–6. Carr pointed out in this article that linked lists can readily be manipulated in ordinary programming languages, without requiring sophisticated subroutines or interpretive systems. See also G. A. Blaauw, *IBM J. Res. and Dev.* **3** (1959), 288–301.

At first, one-word nodes were used for linked tables, but about 1959 the usefulness of several consecutive words per node and "multilinked" lists was gradually being discovered by several different groups of people. The first article dealing specifically with this idea was published by D. T. Ross, *CACM* **4** (1961), 147–150; at that time he used the term "plex" for what has been called a "node" in this chapter, but he subsequently has used the word "plex" in a different sense to denote a class of nodes combined with associated algorithms for their traversal.

Notations for referring to fields within nodes are generally of two kinds: the name of the field either precedes or follows the pointer designation. Thus, while we have written "`INFO(P)`" in this chapter, some other authors write, for example, "`P.INFO`". At the time this chapter was prepared, the two notations seemed to be equally prominent. The notation adopted here has been used in the author's lectures since 1962 and it has also been independently devised by numerous other people, primarily because it is natural to use mathematical functional notation to describe attributes of a node. (For example, see the article by N. Wirth and C. A. R. Hoare, *CACM* **9** (1966), 413–432.) Note that "`INFO(P)`" is pronounced "info of `P`" in conventional mathematical verbalization, just as $f(x)$ is rendered "f of x." The alternative notation `P.INFO` has less of a natural flavor, since it tends to put the emphasis on `P`, although it can be read "`P`'s info"; the reason `INFO(P)` seems to be more natural is apparently the fact that `P` is variable, but `INFO` has a fixed significance when the notation is employed. By analogy, we could consider a vector $A = (A[1], A[2], \ldots, A[100])$ to be

a node having 100 fields named 1, 2, . . . , 100. Now the second field would be referred to as "2(P)" in our notation, where P points to the vector A; but if we are referring to the jth element of the vector, we find it more natural to write $A[j]$, putting the variable quantity "j" second. Similarly it seems most appropriate to put the variable quantity "P" second in the notation INFO(P).

Perhaps the first people to recognize that the concepts "stack" (last-in-first-out) and "queue" (first-in-first-out) are important objects of study were cost accountants interested in reducing income tax assessments; for a discussion of the "LIFO" and "FIFO" methods of pricing inventories, see any intermediate accounting textbook, e.g., C. F. and W. J. Schlatter, *Cost Accounting* (New York: Wiley, 1957), Chapter 7. In 1947 A. M. Turing developed a stack, called Reversion Storage, for use in subroutine linkage (see Section 1.4.5). No doubt simple uses of stacks kept in sequential memory locations were common in computer programming from the earliest days, since a stack is such a simple and natural concept. The programming of stacks in linked form appeared first in IPL, as stated above; the name "stack" stems from IPL terminology (although "pushdown list" was the more official IPL wording), and it was also independently introduced in Europe by E. W. Dijkstra. "Deque" is a term introduced by E. J. Schweppe.

The origin of circular and doubly linked lists is obscure; presumably these ideas occurred naturally to many people. A strong factor in the popularization of these techniques was the existence of general List-processing systems based on them [principally the Knotted List Structures, *CACM* **5** (1962), 161–165, and Symmetric List Processor, *CACM* **6** (1963), 524–544, of J. Weizenbaum].

Various methods for addressing and traversing multidimensional arrays of information were developed independently by clever programmers since the earliest days of computers, and thus another part of the unpublished computer folklore was born. This subject was first surveyed in print by H. Hellerman, *CACM* **5** (1962), 205–207. See also J. C. Gower, *Comp. J.* **4** (1962), 280–286.

Tree structures represented explicitly in computer memory were described for the first time in applications to algebraic formula manipulation. The A-1 compiler language, developed by G. M. Hopper in 1951, used arithmetic expressions written in a three-address code; the latter is equivalent to the INFO, LLINK, and RLINK of a binary tree representation. In 1952, H. G. Kahrimanian developed algorithms for differentiating algebraic formulas represented in the A-1 compiler language; see *Symposium on Automatic Programming* (Washington, D.C.: Office of Naval Research, May 1954), 6–14.

Since then, tree structures in various guises have been studied independently by many people in connection with numerous computer applications, but the basic techniques for tree manipulation (not general List manipulation) have seldom appeared in print except in detailed description of particular algorithms. The first general survey was made in connection with a more general study of all data structures by K. E. Iverson and L. R. Johnson [IBM Corp. research reports RC-390, RC-603, 1961; see Iverson, *A Programming Language* (New York: Wiley, 1962), Chapter 3]. See also G. Salton, *CACM* **5** (1962), 103–114.

The concept of *threaded* trees is due to A. J. Perlis and C. Thornton, *CACM* **3** (1960), 195–204. Their paper also introduced the important idea of traversing trees in various orders, and gave numerous examples of algebraic manipulation algorithms. Unfortunately, this important paper was hastily prepared and it contains many misprints. The threaded lists of Perlis and Thornton actually were only "right-threaded trees" in our terminology; binary trees which are threaded in *both* directions were independently discovered by A. W. Holt, *A Mathematical and Applied Investigation of Tree Structures* (Thesis, U. of Pennsylvania, 1963). Postorder and preorder for the nodes of trees were called "normal along order" and "dual along order" by Z. Pawlak, *Colloquium on the Foundation of Mathematics*, etc. (Tihany, 1962, published by Akadémiai Kiadó, Budapest, 1965), 227–238. Preorder was called "subtree order" by Iverson and Johnson in the references cited above. Graphical ways to represent the connection between tree structures and corresponding linear notations were described by A. G. Oettinger, *Proc. Harvard Symp. on Digital Computers and their Applications* (April, 1961), 203–224. Representation of trees in preorder by degrees, with associated algorithms relating this representation to Dewey decimal notation and other properties of trees, was presented by S. Gorn, *Proc. Symp. Math. Theory of Automata* (Brooklyn: Poly. Inst., 1962), 223–240.

The history of tree structures as mathematical entities, together with a bibliography of the subject, is reviewed in Section 2.3.4.6.

At the time this section was written, the most widespread knowledge about information structures was due to programmers' exposure to List processing systems, which have a very important part in this history. The first widely used system was IPL-V (a descendant of IPL-III, developed late in 1959); IPL-V is an interpretive system in which a programmer learns a machine-like language for List operations. At about the same time, FLPL (a set of FORTRAN subroutines for List manipulation, also inspired by IPL but using subroutine calls instead of interpretive language) was developed by H. Gelernter and others. A third system, LISP, was designed by J. McCarthy, also in 1959. LISP is quite different from its predecessors: programs for it are expressed in mathematical functional notation combined with "conditional expressions" (see Chapter 8), then converted into a List representation. Many List processing systems have come into existence since then, of which the most prominent historically is J. Weizenbaum's SLIP; this is a set of subroutines for use in FORTRAN programs, operating on doubly linked Lists.

An article by Bobrow and Raphael, *CACM* **7** (1964), 231–240, may be read as a brief introduction to IPL-V, LISP, and SLIP, and it gives a comparison of these systems. An excellent introduction to LISP has been given by P. M. Woodward and D. P. Jenkins, *Comp. J.* **4** (1961), 47–53. See also the authors' discussions of their own systems, which are each articles of considerable historical importance: "An introduction to IPL-V" by A. Newell and F. M. Tonge, *CACM* **3** (1960), 205–211; "A FORTRAN-compiled List Processing Language" by H. Gelernter, J. R. Hansen, and C. L. Gerberich, *JACM* **7** (1960), 87–101; "Recursive functions of symbolic expressions and their computation by machine,

I" by John McCarthy, *CACM* **3** (1960), 184–195; "Symmetric List Processor" by J. Weizenbaum, *CACM* **6** (1963), 524–544. The latter article includes a complete description of all of the algorithms used in SLIP. In recent years a number of books about these systems have also been written.

Several *string manipulation* systems have also appeared; these are primarily concerned with operations on variable-length strings of alphabetic information (looking for occurrences of certain substrings, etc.). Historically, the most important of these have been COMIT (V. H. Yngve, *CACM* **6** (1963), 83–84) and SNOBOL (D. J. Farber, R. E. Griswold, and I. P. Polonsky, *JACM* **11** (1964), 21–30). Although string manipulation systems have seen wide use and although they are primarily composed of algorithms such as we have seen in this chapter, they play a comparatively small role in the history of the techniques of information structure representation; users of these systems have largely been unconcerned about the details of the actual internal processes carried on by the computer. For a survey of string manipulation techniques, see S. E. Madnick, *CACM* **10** (1967), 420–424.

The IPL-V and FLPL systems for List-processing did not use either a garbage collection or a reference count technique for the problem of shared Lists; instead, each List was "owned" by one List and "borrowed" by all other Lists which referred to it, and a List was erased when its "owner" allowed it to be. Hence, the programmer was enjoined to make sure no List was still borrowing any Lists that were being erased. The reference counter technique for Lists was introduced by G. E. Collins, *CACM* **3** (1960), 655–657; see also the important sequel to this paper, *CACM* **9** (1966), 578–588. Garbage collection was first described in McCarthy's article cited above; see also *CACM* **7** (1964), 38, and an article by Cohen and Trilling, *BIT* **7** (1967), 22–30.

Unfortunately, too few people have realized that manipulation of links is not a magic, complex process that must be done by fancy List manipulation routines. There is still a widespread tendency to feel that the advantages of link manipulation can be obtained only by using a large List-processing system; actually, as we have seen in many examples throughout this chapter, it is quite easy to design linking algorithms which operate more efficiently than a general List-processing subroutine could possibly do. The author hopes that these detailed examples will help to convey a better-balanced attitude towards the use of structural links in data.

Dynamic storage allocation algorithms were in use several years before published information about them appeared. A very readable discussion is given by W. T. Comfort, *CACM* **7** (1964), 357–362 (an article written in 1961). The "boundary-tag" method, introduced in Section 2.5, was designed by the author in 1962 for use in a control program for the B-5000 computer. The "buddy system" was first used by H. Markowitz in connection with the SIMSCRIPT programming system in 1963, and it was independently discovered and published by K. Knowlton, *CACM* **8** (1965), 623–625; see also *CACM* **9** (1966), 616–625. For further discussion of dynamic storage allocation, see the articles by Iliffe

and Jodeit, *Comp. J.* **5** (1962), 200–209; Bailey, Barnett, and Burleson, *CACM* **7** (1964), 339–346; Berztiss, *CACM* **8** (1965), 512–513; and D. T. Ross, *CACM* **10** (1967), 481–492.

A general discussion of information structures and their relation to programming has been given by Mary d'Imperio, "Data Structures and their Representation in Storage," *Annual Review in Automatic Programming* **5** (Oxford: Pergamon Press, 1969). This paper is also a valuable guide to the history of the topic, since it includes a detailed analysis of the structures used in connection with twelve List processing and string manipulation systems. See also the proceedings of two symposia, *CACM* **3** (1960), 183–234 and *CACM* **9** (1966), 567–643, for further historical details. (Several of the individual papers from these proceedings have already been cited above.)

An excellent annotated bibliography, which is primarily oriented towards applications to symbol manipulation and algebraic formula manipulation but which has numerous connections with the material of this chapter, has been compiled by Jean E. Sammet, *Comput. Rev.* **7** (July–August 1966), B1–B31.

In this chapter we have looked at particular types of information structures in great detail, and (lest we fail to see the forest for the trees) it is perhaps wise to take stock of what we have learned and to briefly summarize the general subject of information structures from a broader perspective: Starting with the basic idea of a *node* as an element of data, we have seen many examples which illustrate the fact that it is convenient to represent structural relationships either implicitly (based on the relative order in which nodes are stored in computer memory) or explicitly (by means of links in the nodes, which point to other nodes). The amount of structural information that ought to be represented within the tables of a computer program depends on the operations that are to be performed on the nodes.

The discussion in this chapter does not cover the entire subject of information structures in full generality; at least three important aspects of the subject have not been treated here:

a) It is often necessary or desirable to search through a table to find a node or set of nodes possessing a certain value, and such an operation often has a profound effect on the structure of the table. This situation is explored in detail in Chapter 6.

b) We have primarily been concerned with the internal representation of structure within a computer, and this is obviously only part of the story, since structure must also be represented in the external input and output data. In simple cases, external structure can essentially be treated by the same techniques we have been considering; but the processes of converting between strings of characters and more complex structures are also very important. These processes are analyzed in Chapters 9 and 10.

c) We have primarily discussed representations of structures within a high-speed random-access memory. When slower memory devices (e.g., disks, drums, tapes) are being used, we find that all of the structural problems are

intensified; it is much more crucial to have efficient algorithms and efficient schemes for data representation. It is often necessary to attempt to place "neighboring" nodes, which link to each other, into nearby areas of the memory, etc.; usually the problems are highly dependent on the characteristics of individual machines, so it is difficult to discuss them in general. Hopefully, the simpler examples treated in this chapter will prepare the reader for solving the more difficult problems which arise in connection with less ideal memory devices.

What are the main implications of the subjects treated in this chapter? Perhaps the most important conclusion we can reach is that the ideas we have encountered are not limited to computer programming alone; they apply more generally to everyday life. A collection of nodes containing fields, some of which point to other nodes, appears to be a very good abstract model for structural relationships of all kinds; it shows how we can build up complicated structures from simple ones, and we have seen that corresponding algorithms for manipulating the structure can be designed in a natural manner.

Therefore it seems appropriate to develop much more theory about linked sets of nodes than we know at this time. Perhaps the most obvious way to start such a theory is to define a new kind of abstract machine or "automaton" which deals with linked structures. For example, such an automaton might be defined informally as follows: There are numbers k, l, r, and s, such that the automaton processes nodes containing k link fields and r information fields; it has l link registers and s information registers, which enable it to control the processes it is performing. The information fields and registers may contain any symbols from some given set of information symbols; each of the link fields and link registers either contains Λ or points to a node. The machine can (i) create new nodes (putting a link to the node into a register), (ii) compare information symbols or link values for equality, and (iii) transfer information symbols or link values between registers and nodes. Only nodes pointed to by link registers are immediately accessible. Suitable restrictions on the machine's behavior will make it equivalent to several older species of automata.

Some of the most interesting problems to solve for such devices would be to determine how fast they can solve certain problems, or how many nodes they need to solve certain problems (e.g., to translate certain formal languages). At the time this chapter was written, several interesting results of this kind have been obtained (notably by J. Hartmanis and R. E. Stearns), but only for special classes of "Turing machines" having multiple tapes and read/write heads, etc.; since the Turing machine model is comparatively unrealistic, these results tend to have little to do with practical problems. It is true that, as the number n of nodes created by a linking automaton approaches infinity, we must admit that we don't know how to build such a device physically, since we expect the machine operations will take the same amount of time regardless of the size of n; if linking is represented by using addresses as in a computer memory, it is necessary to put a bound on the number of nodes, since the link fields have a

fixed size. A multitape Turing machine is therefore a more realistic model when n approaches infinity. Yet it seems reasonable to believe that a linking automaton as described above leads to a more appropriate theory of the complexity of algorithms than Turing machines do, even when asymptotic formulas for large n are considered, because the theory is more likely to be relevant for practical values of n.

EXERCISE

1. [*M50*] Explore the properties of linking automata as defined in the text.

You will, I am sure, agree with me that if page 534 finds us only in the second chapter, the length of the first one must have been really intolerable.

—SHERLOCK HOLMES (*The Valley of Fear*, Chapter 1)

ANSWERS TO EXERCISES

NOTES ON THE EXERCISES

1. An average problem for a mathematically inclined reader.

3. See W. J. LeVeque, *Topics in Number Theory* (Addison-Wesley, 1956), vol. 2, chapter 3. (*Note:* One of the men who read a preliminary draft of the manuscript for this book reported that he had discovered a truly remarkable proof, which the margin of his copy was too small to contain.)

SECTION 1.1

1. $t \leftarrow a,\ a \leftarrow b,\ b \leftarrow c,\ c \leftarrow d,\ d \leftarrow t.$

2. After the first time, the values of the variables m, n are the previous values of n, r, respectively; and $n > r$.

3. **Algorithm F** (*Euclid's algorithm*). Given two positive integers m and n, find their greatest common divisor.

F1. [Remainder m/n.] Divide m by n and let r be the remainder.

F2. [Is it zero?] If $r = 0$, the algorithm terminates with answer n.

F3. [Remainder n/r.] Divide n by r and let m be the remainder.

F4. [Is it zero?] If $m = 0$, the algorithm terminates with answer r.

F5. [Remainder r/m.] Divide r by m and let n be the remainder.

F6. [Is it zero?] If $n = 0$, the algorithm terminates with answer m; otherwise go back to step F1. ▮

4. By algorithm E, $n = 6099,2166,1767,399,171,57$. Answer $= 57$.

5. Not finite nor definite nor effective, perhaps no output; in format, no letter is given before step numbers, no summary phrase appears, and there is no "▮".

6. We try Algorithm E with $n = 5$ and count the number of times step E1 is executed. For $m = 1, 2, 3, 4, 5$, respectively, we get 2, 3, 4, 3, 1 times; the average is $2.6 = T_5$.

7. In all but a finite number of cases, $n > m$. In this case, the first iteration of Algorithm E merely exchanges these numbers; so $U_m = T_m + 1$.

8. Let $A = \{a, b, c\}$, $N = 5$. The algorithm will terminate with the string $a^{\gcd(m,n)}$.

j	θ_j	ϕ_j	b_j	a_j	
0	ab	(empty)	1	2	Remove one a and one b, or go to 2.
1	(empty)	c	0	0	Add c at extreme left, go back to 0.
2	a	b	2	3	Change all a's to b's.
3	c	a	3	4	Change all c's to a's.
4	b	b	0	5	If b's remain, repeat.

9. For example we can say C_2 represents C_1 if there is a function g from I_1 into I_2, and function h from Q_2 into Q_1 taking Ω_2 into Ω_1, and a function j from Q_2 into the positive integers, satisfying the following conditions:

a) If x is in I_1, C_1 produces the output y from x if and only if there exists a y' in Ω_2 for which C_2 produces the output y' from $g(x)$ and $h(y') = y$.
b) If q is in Q_2 then $f_1(h(q)) = h(f_2^{j(q)}(q))$, where $f_2^{j(q)}$ means the function f_2 is to be iterated $j(q)$ times.

For example, let C_1 be as in (2) and let C_2 have $I_2 = \{(m, n)\}$, $\Omega_2 = \{(m, n, d)\}$, $Q_2 = I_2 \cup \Omega_2 \cup \{(m, n, a, b, 1)\} \cup \{(m, n, a, b, r, 2)\} \cup \{(m, n, a, b, r, 3)\} \cup \{(m, n, a, b, r, 4)\}$. Let $f_2(m, n) = (m, n, m, n, 1)$; $f_2(m, n, d) = (m, n, d)$; $f_2(m, n, a, b, 1) = (m, n, a, b, a \bmod b, 2)$; $f_2(m, n, a, b, r, 2) = (m, n, b)$ if $r = 0$, otherwise $(m, n, a, b, r, 3)$; $f_2(m, n, a, b, r, 3) = (m, n, b, b, r, 4)$; $f_2(m, n, a, b, r, 4) = (m, n, a, r, 1)$.

Now let $h(m, n) = (m, n) = g(m, n)$; $h(m, n, d) = (d)$; $h(m, n, a, b, 1) = (a, b, 0, 1)$ if $a = m$, $b = n$, otherwise $(a, b, b, 1)$; $h(m, n, a, b, r, 2) = (a, b, r, 2)$; $h(m, n, a, b, r, 3) = (a, b, r, 3)$; $h(m, n, a, b, r, 4) = h(f_2(m, n, a, b, r, 4))$; $j(m, n, a, b, r, 3) = j(m, n, a, b, r, 4) = 2$, otherwise $j(q) = 1$. Then C_2 represents C_1.

Notes: It is tempting to try to define things in a more simple way, e.g. to let g map Q_1 into Q_2 and to insist that when $x_0, x_1, \ldots$ is a computational sequence in C_1 then $g(x_0), g(x_1), \ldots$ is a subsequence of the computational sequence in C_2 that begins with $g(x_0)$. But this is inadequate, e.g. in the above example C_1 forgets the original values of m and n but C_2 does not.

If C_3 represents C_2 by means of functions g, h, j, and if C_2 represents C_1 by means of functions g', h', j', then C_3 represents C_1 by means of functions g'', h'', j'', where

$$g''(x) = g'(g(x)), \qquad h''(x) = h(h'(x)),$$

and

$$j''(q) = \sum_{0 \le k < j(h'(q))} j'(q_k),$$

if $q_0 = q$, $q_{k+1} = f_3^{j'(q_k)}(q_k)$. Hence the above relation is transitive. We can say C_2 *directly* represents C_1 if the function j is bounded; this relation is also transitive. The relation "C_2 represents C_1" generates an equivalence relation in which two computational methods apparently are equivalent if and only if they compute isomorphic functions of their inputs; the relation "C_2 directly represents C_1" generates a more interesting equivalence relation which perhaps matches the intuitive idea of being "essentially the same algorithm."

SECTION 1.2.1

1. (a) Prove $P(0)$. (b) Prove that $P(0), \ldots, P(n)$ implies $P(n+1)$, for all $n \geq 0$.

2. The theorem has not been proved for $n = 2$; in the second part of the proof, take $n = 1$; we assume there that $a^{-1} = 1$. If this condition is true (i.e. if $a = 1$) the theorem is indeed valid.

3. The correct answer is $1 - 1/n$. The mistake occurs in the proof for $n = 1$, when the formula on the left either may be assumed to be meaningless, or it may be assumed to be zero (since there are $n - 1$ terms).

5. If n is prime, it is trivially a product of primes. Otherwise by definition, n has factors, so $n = km$ for $1 < k, m < n$. Since both k and m are less than n, by induction they can be written as products of primes; hence n is the product of the primes appearing in the representations of k and m.

6. In the notation of Fig. 3, we prove *A5* implies *A6*. This is clear since *A5* implies $(a' - qa)m + (b' - qb)n = (a'm + b'n) - q(am + bn) = c - qd = r$.

7. Solution is $1 + 2 + \cdots + n$; or, $n(n+1)/2$.

8. (a) We will show $(n^2 - n + 1) + (n^2 - n + 3) + \cdots + (n^2 + n - 1)$ equals n^3. The sum is $(1 + 3 + \cdots + (n^2 + n - 1)) - (1 + 3 + \cdots + (n^2 - n - 1)) = ((n^2 + n)/2)^2 - ((n^2 - n)/2)^2 = n^3$. We have used Eq. (2); however, an inductive proof was requested, so another approach should be taken! For $n = 1$, the result is obvious. Let $n \geq 1$; $(n+1)^2 - (n+1) = n^2 - n + 2n$, so the first terms for $n + 1$ are $2n$ larger; thus the sum for $n + 1$ is the sum for n plus $2n + \cdots + 2n$ [n times] $+ (n+1)^2 + (n+1) - 1$; this equals $n^3 + 2n^2 + n^2 + 3n + 1 = (n+1)^3$. (b) We have shown the first term for $(n+1)^3$ is two greater than the last term for n^3. Therefore by Eq. (2), $1^3 + 2^3 + \cdots + n^3$ = sum of consecutive odd numbers starting with $1 = (\text{number of terms})^2 = (1 + 2 + \cdots + n)^2$.

10. Obvious for $n = 10$. If $n \geq 10$, we have $2^{n+1} = 2 \cdot 2^n > (1 + \frac{1}{10})^3 2^n$ and by induction this is greater than $(1 + 1/n)^3 n^3 = (n+1)^3$.

11. $(-1)^n(n+1)/(4(n+1)^2 + 1)$.

12. The only nontrivial part of the extension is the calculation of the integer q in E2. This can be done by repeated subtraction, reducing to the problem of determining whether $u + v\sqrt{2}$ is positive, negative, or zero, and the latter problem is readily solved.

It is easy to show that whenever $u + v\sqrt{2} = u' + v'\sqrt{2}$, we must have $u = u'$ and $v = v'$, since $\sqrt{2}$ is irrational. Now it is clear that 1 and $\sqrt{2}$ have no common divisor, if we define divisor in the sense that $u + v\sqrt{2}$ divides $a(u + v\sqrt{2})$ if and only if a is an integer.

(*Note:* However, if we extend the concept of divisor so that $u + v\sqrt{2}$ is said to divide $a(u + v\sqrt{2})$ if and only if a has the form $u' + v'\sqrt{2}$ for integers u' and v', there *is* a way to extend Algorithm E so that it always will terminate: If in step E2 we have $c = u + v\sqrt{2}$ and $d = u' + v'\sqrt{2}$, compute $c/d = c(u' - v'\sqrt{2})/(u'^2 - 2v'^2) = x + y\sqrt{2}$ where x and y are rational. Now let $q = u'' + v''\sqrt{2}$, where u'' and v'' are the nearest integers to x and y; and let $r = c - qd$. If $r = u''' + v'''\sqrt{2}$, it follows that $|u'''^2 - 2v'''^2| < |u'^2 - 2v'^2|$, hence the computation will terminate. For further information, see "quadratic Euclidean domains" in number theory textbooks.)

13. Add "$n = n_0$" to *A1* and *A2*; "$T \leq 3(n_0 - d) + k$" to the others, where $k = 2, 3, 3, 1$, respectively for *A3*, *A4*, *A5*, *A6*. Also add "$d > 0$" to *A4*.

15. (a) Let $A = S$ in (iii); this implies every nonempty well-ordered set has a "least" element.

(b) Let $x \prec y$ if $|x| < |y|$ or if $|x| = |y|$ and $x < 0 < y$.

(c) No, the subset of all positive reals fails to satisfy (iii). (*Note:* Using the "axiom of choice," a rather complicated argument can be given to show that every set can be well-ordered somehow; but nobody has yet been able to define an explicit relation which well-orders the real numbers.)

(d) To prove (iii) for T_n, use induction on n: Let A be a nonempty subset of T_n and consider A_1, the set of first components of A. Since A_1 is a nonempty subset of S, and S is well-ordered, A_1 contains a smallest element x. Now consider A_x, the subset of A in which the first component equals x; A_x may be considered a subset of T_{n-1} if its first component is suppressed, so by induction A_x contains a smallest element $(x, x_2, \ldots, x_n)$ which in fact is the smallest element of A.

(e) No, although properties (i) and (ii) are valid. If S contains at least two distinct elements a and b, the set (b), (a, b), (a, a, b), (a, a, a, b), (a, a, a, a, b), . . . has no least element.

T can be well-ordered if we define $(x_1, \ldots, x_n) \prec (y_1, \ldots, y_m)$ whenever $n < m$, or $n = m$ and $(x_1, \ldots, x_n) \prec (y_1, \ldots, y_m)$ in T_n.

(f) Let S be well-ordered by $\prec$. If such an infinite sequence exists, the set A consisting of the members of the sequence fails to satisfy property (iii), for no element of the sequence can be smallest. Conversely if $\prec$ is a relation satisfying (i) and (ii) but not (iii), let A be a non-empty subset of S which has no smallest element. Since A is not empty, we can find x_1 in A; since x_1 is not the smallest element of A, there is x_2 in A for which $x_2 \prec x_1$; since x_2 is not the smallest element either, we can find $x_3 \prec x_2$; etc.

(g) Let A be the set of all x for which $P(x)$ is false. If A is not empty, it contains a smallest element, x_0. Hence $P(y)$ is true for all $y \prec x_0$. But this implies $P(x_0)$ is true, so x_0 is not in A (a contradiction). Therefore A must be empty, i.e. $P(x)$ is always true.

SECTION 1.2.2

1. There is none; if r is a positive rational, $r/2$ is smaller.

2. Not if infinitely many nines appear; in that case the decimal expansion of the number is $1 + .24000000\ldots$, according to Eq. (2).

3. $\frac{1}{343}$.

4. 4.

6. The decimal expansion of a number is unique, so $x = y$ if and only if $m = n$, and $d_i = e_i$ for $i = 1, 2, \ldots$. If $x \neq y$, one may compare m vs. n, d_1 vs. e_1, d_2 vs. e_2, etc., and when the first inequality occurs the larger one belongs to the larger of x, y.

7. One may use induction on x, first proving the laws for x positive, and then for x negative. Details are omitted here.

8. By trying $n = 0, 1, 2, \ldots$ we find the value of n for which $n^m \leq u < (n+1)^m$. Then assuming inductively $n, d_1, \ldots, d_{k-1}$ have been determined, d_k is the digit

such that

$$\left(n+\frac{d_1}{10}+\cdots+\frac{d_k}{10^k}\right)^m \le u < \left(n+\frac{d_1}{10}+\cdots+\frac{d_k}{10^k}+\frac{1}{10^k}\right)^m.$$

9. $((b^{p/q})^{u/v})^{qv} = (((b^{p/q})^{u/v})^v)^q = ((b^{p/q})^u)^q = ((b^{p/q})^q)^u = b^{pu}$, hence $(b^{p/q})^{u/v} = b^{pu/qv}$. This proves the second law. We prove the first law using the second: $b^{p/q}b^{u/v} = (b^{1/qv})^{pv}(b^{1/qv})^{qu} = (b^{1/qv})^{pv+qu} = b^{p/q+u/v}$.

10. If $\log_{10} 2 = p/q$, with p and q positive, then $2^q = 10^p$, which is absurd since the righthand side is divisible by 5 but the lefthand side isn't.

11. Infinitely many! No matter how many digits of x are given, we will not know whether $10^x = 1.99999\ldots$ or $2.00000\ldots$. There is nothing mysterious or paradoxical in this; a similar situation occurs in addition, if we are adding $.444444\ldots$ to $.555555\ldots$

12. They are the only values of $d_1, \ldots, d_8$ which satisfy Eq. (6).

13. (a) First prove by induction that if $y > 0$, $1 + ny \le (1+y)^n$. Then set $y = x/n$, and take n-th roots. (b) $x = b - 1$, $n = 10^k$.

14. Set $x = \log_b c$ in the second equation of (4) and take logarithms of both sides of this equation.

15. Prove it, by transposing "$\log_b y$" to the other side of the equation and using (10).

16. $\ln x/\ln 10$.

17. 5; 1; 1; 0; undefined.

18. No, $\log_8 x = \log_2 x/\log_2 8 = \frac{1}{3}\log_2 x$.

19. Yes, since $\log_2 x < (\log_{10} x)/.301 < 14/.301 < 47$.

20. They are reciprocals.

21. $(\ln\ln x - \ln\ln b)/\ln b$.

22. From the tables appearing in Appendix B, $\log_2 x = 1.442695 \ln x$; $\log_{10} x = .4342945 \ln x$. The error is $(1.442695 - 1.4342945)/1.442695 = 0.582\%$.

23. Take the figure of area $\ln y$, and divide its height by x while multiplying its length by x. This deformation preserves its area and makes it congruent to the piece left when $\ln x$ is removed from $\ln xy$: for the height at point $x + xt$ in the diagram for $\ln xy$ is $1/(x + xt) = \big(1/(1+t)\big)/x$.

24. Substitute 2 everywhere 10 appears.

27. Prove by induction on k that

$$x^{2^k}(1-\eta)^{2^{k+1}-1} \le 10^{2^k(n+b_1/2+\cdots+b_k/2^k)}x_k' \le x^{2^k}(1+\epsilon)^{2^{k+1}-1}$$

and take logarithms.

28. **E1.** Set $y \leftarrow 1$, $k \leftarrow 0$.

E2. If $x = 0$, stop.

E3. If $x < \log_b(1 + 2^{-k})$, go to E5.

E4. Set $x \leftarrow x - \log_b(1 + 2^{-k})$, $y \leftarrow y + 2^{-k}y$, go to E2.

E5. Increase k by 1, go to E2. ▮

The computational error arises when we set $x \leftarrow x - \log_b(1 + 2^{-k}) + \eta_j$ and $y \leftarrow y(1 + 2^{-k})(1 + \epsilon_j)$ at the j-th execution of step E4, for certain small errors η_j and ϵ_j. When the algorithm terminates, we have $y = b^x \Pi(1 + \epsilon_j) b^{\Sigma \eta_j}$. Further analysis depends on b and the computer word size. Notice that in both this case and in Ex. 26, it is possible to refine the error estimates somewhat if the base is e, since for most values of k the table entry $\ln(1 \pm 2^{-k})$ can be given with high accuracy: it equals $\pm 2^{-k} - \frac{1}{2} 2^{-2k} \pm \frac{1}{3} 2^{-3k} - \cdots$.

Note: Similar algorithms can be given for trigonometric functions; see J. E. Meggitt, *IBM J. Res. and Dev.* **6** (1962), 210–226, **7** (1963), 237–245.

29. e; 3; 4.

SECTION 1.2.3

1. $a_1 + a_2 + a_3$.

2. $\frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{9} + \frac{1}{11}$; $\frac{1}{9} + \frac{1}{3} + \frac{1}{1} + \frac{1}{3} + \frac{1}{9}$.

3. The rule for $p(j)$ is violated; in the first place, the value 3 is assumed for no n^2, and in the second place the value 4 is assumed for *two* n^2.

4. $(a_{11}) + (a_{21} + a_{22}) + (a_{31} + a_{32} + a_{33})$
$$= (a_{11} + a_{21} + a_{31}) + (a_{22} + a_{32}) + (a_{33}).$$

5. It is only necessary to use the rule $a\sum_{R(i)} x_i = \sum_{R(i)} (ax_i)$:

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

7. Use Eq. (3); the two limits are interchanged and the terms between a_0 and a_c must be transferred from one limit to the other.

8. Let $a_{(i+1)i} = +1$, and $a_{i(i+1)} = -1$, for all $i \geq 0$, and all other a_{ij} zero; let $R(i) = S(i) =$ "$i \geq 0$". The lefthand side is -1, the righthand side is $+1$.

10. No, the two applications of rule (d) assume $n \geq -1$.

11. $n + 1$.

12. $\frac{7}{6}(1 - 1/7^{n+1})$.

13. $m(n - m + 1) + \frac{1}{2}(n - m)(n - m + 1)$; or, $\frac{1}{2}\big(n(n + 1) - m(m - 1)\big)$.

14. $\big(m(n - m + 1) + \frac{1}{2}(n - m)(n - m + 1)\big)\big(r(s - r + 1) + \frac{1}{2}(s - r)(s - r + 1)\big)$.

15, 16. Key steps:

$$\sum_{0\leq j\leq n} jx^j = x\sum_{1\leq j\leq n} jx^{j-1} = x\sum_{0\leq j\leq n-1} (j+1)x^j$$
$$= x\sum_{0\leq j\leq n} jx^j - nx^{n+1} + x\sum_{0\leq j\leq n-1} x^j.$$

17. The number of elements in S.

18. $S'(j) =$ "$1 \leq j < n$". $R'(i, j) =$ "n is a multiple of i and $i > j$".

19. $a_n - a_{m-1}$.

20. $(b-1)\sum_{0\le k\le n}(n-k)b^k + n + 1 = \sum_{0\le k\le n} b^k$; this formula follows from (14) and the result of Ex. 16.

21. Analogous to (3), plus the stipulation that there exists an integer j_0 such that

$$\prod_{\substack{R(j)\\ |j|>j_0}} a_j \neq 0.$$

22. For (5), (7) just change $\sum$ to $\prod$. Also, we have

$$\prod_{R(i)} (b_i c_i) = \left(\prod_{R(i)} b_i\right)\left(\prod_{R(i)} c_i\right); \quad \left(\prod_{R(j)} a_j\right)\left(\prod_{S(j)} a_j\right) = \left(\prod_{\substack{R(j)\\ \text{or } S(j)}} a_j\right)\left(\prod_{\substack{R(j)\\ \text{and } S(j)}} a_j\right).$$

23. $0 + x = x$ and $1 \cdot x = x$. This makes many operations and equations simpler, e.g. rule (d) and its analogue in the previous exercise.

25. First step and last step o.k. Second step, uses i for two different purposes at once. Third step, should probably be $\sum_{1\le i\le n} n$.

26. Key steps:

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

The answer is

$$\left(\prod_{0\le i\le n} a_i\right)^{n+2}.$$

28. $(n+1)/2n$.

29. a) $\sum_{0\le k\le j\le i\le n} a_i a_j a_k$.

b) Let $S_r = \sum_{0\le i\le n} a_i^r$. Solution: $\frac{1}{3}S_3 + \frac{1}{2}S_1S_2 + \frac{1}{6}S_1^3$. The general solution to this problem, as the number of indices gets larger, may be found in Section 1.2.9, Eq. (34).

31. $n\sum_{1\le j\le n} a_j b_j - \left(\sum_{1\le j\le n} a_j\right)\left(\sum_{1\le j\le n} b_j\right)$.

33. This can be proved by induction on n, if we rewrite the formula as

$$\frac{1}{x_n - x_{n-1}}\left(\sum_{1\le j\le n}\frac{x_j^r(x_j - x_{n-1})}{\prod_{1\le k\le n,\, k\neq j}(x_j - x_k)} - \sum_{1\le j\le n}\frac{x_j^r(x_j - x_n)}{\prod_{1\le k\le n,\, k\neq j}(x_j - x_k)}\right).$$

Each of these sums now has the form of the original sum, except on $n-1$ elements, and the values turn out nicely by induction when $0 \le r \le n-1$. When $r = n$, consider the identity

$$0 = \sum_{1\le j\le n}\frac{\prod_{1\le k\le n}(x_j - x_k)}{\prod_{1\le k\le n,\, k\neq j}(x_j - x_k)} = \sum_{1\le j\le n}\frac{x_j^n - (x_1 + \cdots + x_n)x_j^{n-1} + P(x_j)}{\prod_{1\le k\le n,\, k\neq j}(x_j - x_k)}$$

where $P(x_j)$ is a polynomial of degree $n-2$; from the solution for $r = 0, 1, \ldots, n-1$ we obtain the desired answer.

Note: The formulas here are the basis for numerical methods concerning "divided differences." The following alternate method of proof, using complex variable theory, is less elementary but more elegant: By the residue theorem, the value of the given sum is

$$\frac{1}{2\pi i}\int_{|z|=R}\frac{z^r\,dz}{(z-x_1)\cdots(z-x_n)}$$

where $R > |x_1|, \ldots, |x_n|$. The Laurent expansion of the integrand converges uniformly on $|z| = R$; it is

$$z^{r-n}\left(\frac{1}{1-x_1/z}\right)\cdots\left(\frac{1}{1-x_n/z}\right)$$
$$= z^{r-n} + (x_1 + \cdots + x_n)z^{r-n-1} + (x_1^2 + x_1x_2 + \cdots)z^{r-n-2} + \cdots.$$

Integrating term by term, everything vanishes except the coefficient of z^{-1}. This method gives us the *general formula* for an arbitrary integer $r \geq 0$:

$$\sum_{\substack{j_1+\cdots+j_n=r-n+1\\ j_1,\cdots,j_n\geq 0}} x_1^{j_1}\ldots x_n^{j_n}.$$

34. If the reader has tried earnestly to solve this problem, *without* getting the answer, perhaps its purpose has been achieved. The temptation to regard the numerators as polynomials in x rather than as polynomials in k is almost overwhelming. It would undoubtedly be easier to prove the considerably more general result

$$\sum_{1\leq k\leq n}\frac{\prod_{1\leq r\leq n-1}(y_k - z_r)}{\prod_{1\leq r\leq n,\ r\neq k}(y_k - y_r)} = 1,$$

which is an identity in $2n - 1$ variables!

35. If $R(j)$ never holds, the value should be $-\infty$. The stated analogue of rule (a) is based on the identity $a + \max(b, c) = \max(a + b, a + c)$. Similarly if all a_i, b_j are *nonnegative*, we have

$$\sup_{R(i)} a_i \sup_{S(j)} b_j = \sup_{R(i)}\sup_{S(j)} a_ib_j.$$

Rules (b), (c) do not change; for rule (d) we get the simpler form

$$\sup\left(\sup_{R(j)} a_j, \sup_{S(j)} a_j\right) = \sup_{R(j)\text{ or }S(j)} a_j.$$

36. Subtract column one from columns $2, \ldots, n$. Add rows $2, \ldots, n$ to row one. The result is a triangular determinant.

37. Subtract column one from columns $2, \ldots, n$. Then subtract x_1 times row $k - 1$ from row k, for $k = n, n - 1, \ldots, 2$ (in that order). We now factor x_1 out of the first column and factor $x_k - x_1$ out of columns $k = 2, \ldots, n$, obtaining $x_1(x_2 - x_1)\ldots(x_n - x_1)$ times a Vandermonde determinant of order $n - 1$, so the process continues by induction.

Alternate proof, using "higher" mathematics: The determinant is a polynomial in the variables $x_1, \ldots, x_n$ of total degree $1 + 2 + \cdots + n$. It vanishes if $x_j = 0$ or if $x_i = x_j$ $(i < j)$, and the coefficient of $x_1^1x_2^2\ldots x_n^n$ is $+1$. These facts characterize its

value. In general, if two rows of a matrix become equal for $x_i = x_j$, their difference is usually divisible by $x_i - x_j$, and this observation often speeds the evaluation of determinants. (R. W. Floyd.)

38. Subtract column one from columns $2, \ldots, n$, and factor out $(x_1 + y_1)^{-1} \ldots (x_n + y_1)^{-1}(y_1 - y_2) \ldots (y_1 - y_n)$ from rows and columns. Now subtract row one from rows $2, \ldots, n$ and factor out $(x_1 - x_2) \ldots (x_1 - x_n)(x_1 + y_2)^{-1} \ldots (x_1 + y_n)^{-1}$; we are left with the Cauchy determinant of order $n - 1$.

39. Let I = identity matrix, J = matrix of all ones. Since $J^2 = nJ$, we find immediately $(xI + yJ)\big((x + ny)I - yJ\big) = x(x + ny)I$.

40. $$\sum_{1 \le t \le n} b_{it} x_j^t = x_j \prod_{\substack{1 \le k \le n \\ k \ne i}} (x_k - x_j) \Big/ x_i \prod_{\substack{1 \le k \le n \\ k \ne i}} (x_k - x_i) = \delta_{ij}.$$

41. This follows immediately from the observations about the relation of an inverse matrix to cofactors. It may also be interesting to give a direct proof here.

$$\sum_{1 \le t \le n} \frac{1}{x_i + y_t} b_{tj} = \sum_{1 \le t \le n} \frac{\prod_{k \ne t} (x_j + y_k - x) \prod_{k \ne i} (x_k + y_t)}{\prod_{k \ne j} (x_j - x_k) \prod_{k \ne t} (y_t - y_k)}$$

when $x = 0$. This is a polynomial of degree at most $n - 1$ in x. If we set $x = x_j + y_s$, $1 \le s \le n$, the terms are zero except when $s = t$, so the value of this polynomial is

$$\prod_{k \ne i} (-x_k - y_s) \Big/ \prod_{k \ne j} (x_j - x_k) = \prod_{k \ne i} (x_j - x_k - x) \Big/ \prod_{k \ne j} (x_j - x_k).$$

Since these polynomials of degree at most $n - 1$ agree at n distinct points x, they agree also for $x = 0$, hence

$$\sum_{1 \le t \le n} \frac{1}{x_i + y_t} b_{tj} = \prod_{k \ne i} (x_j - x_k) \Big/ \prod_{k \ne j} (x_j - x_k) = \delta_{ij}.$$

42. $n/(x + ny)$.

43. $1 - \prod_{1 \le k \le n} (1 - 1/x_k)$. This is easily verified if any $x_i = 1$, since the inverse of any matrix having a row or column all of ones must have elements whose sum is 1. If none of the x_i equals one, sum the elements of row i as in Ex. 44 and obtain $\prod_{k \ne i} (x_k - 1)/x_i \prod_{k \ne i} (x_k - x_i)$. We can now sum this on i using Ex. 33, with $r = 0$ (multiply numerator and denominator by $(x_i - 1)$).

44. We find

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

after applying Ex. 33. And

$$\sum_{1 \le j \le n} c_j = \sum_{1 \le j \le n} \frac{(x_j^n + (y_1 + \cdots + y_n) x_j^{n-1} + \cdots)}{\prod_{1 \le k \le n,\, k \ne j} (x_j - x_k)}$$

$$= (x_1 + x_2 + \cdots + x_n) + (y_1 + y_2 + \cdots + y_n).$$

45. Let $x_i = i$, $y_j = j - 1$. From Ex. 44, the sum of the elements of the inverse is $(1 + 2 + \cdots + n) + \big((n - 1) + (n - 2) + \cdots + 0\big) = n^2$. From Ex. 38, the

elements of the inverse are

$$b_{ij} = \frac{(-1)^{i+j}(i+n-1)!(j+n-1)!}{(i+j-1)(i-1)!^2(j-1)!^2(n-i)!(n-j)!}.$$

This quantity can be put into several forms involving binomial coefficients, for example

$$\frac{(-1)^{i+j}ij}{i+j-1}\binom{-i}{n}\binom{n}{i}\binom{-j}{n}\binom{n}{j}$$

$$= (-1)^{i+j}j\binom{i+j-2}{i-1}\binom{i+n-1}{i-1}\binom{j+n-1}{n-i}\binom{n}{j}.$$

From the latter formula we see that b_{ij} is not only an integer, it is divisible by $i, j, n, i+j-1, i+n-1$, and $j+n-1$. Perhaps the prettiest formula for b_{ij} is

$$(i+j-1)\binom{i+j-2}{i-1}^2\binom{-(i+j)}{n-i}\binom{-(i+j)}{n-j}.$$

The solution to this problem would be extremely difficult if we had not realized that a Hilbert matrix is a special case of a Cauchy matrix; the more general problem is much easier to solve than its special case! It is frequently wise to generalize a problem to its "inductive closure", i.e. to the smallest generalization such that all subproblems that arise in an attempted proof by mathematical induction belong to the same class. In this case, we see that cofactors of a Cauchy matrix are Cauchy matrices, but cofactors of Hilbert matrices are not Hilbert matrices. [For further information, see J. Todd, *J. Res. Nat. Bur. Stand.* **65** (1961), 19–22.]

46. For any integers $k_1, k_2, \ldots, k_m$, let $\epsilon(k_1, \ldots, k_m) = \text{sign}\left(\prod_{1\le i<j\le m}(k_j - k_i)\right)$. If $(q_1, \ldots, q_m)$ is equal to $(k_1, \ldots, k_m)$ except for the fact that k_i and k_j have been interchanged, we have $\epsilon(q_1, \ldots, q_m) = -\epsilon(k_1, \ldots, k_m)$. Therefore we have the equation $\det(B_{k_1\ldots k_m}) = \epsilon(k_1, \ldots, k_m)\det(B_{j_1\ldots j_m})$, if $j_1 \le \cdots \le j_m$ are the numbers $k_1, \ldots, k_m$ rearranged into non-decreasing order. Now by definition of the determinant,

$$\det(AB) = \sum_{1\le q_1,\ldots,q_m\le m} \epsilon(q_1, \ldots, q_m)\left(\sum_{1\le k\le n} a_{1k}b_{kq_1}\right)\cdots\left(\sum_{1\le k\le n} a_{mk}b_{kq_m}\right)$$

$$= \sum_{1\le k_1,\ldots,k_m\le n} a_{1k_1}\ldots a_{mk_m} \sum_{1\le q_1,\ldots,q_m\le m} \epsilon(q_1, \ldots, q_m)b_{k_1q_1}\ldots b_{k_mq_m}$$

$$= \sum_{1\le k_1,\ldots,k_m\le n} a_{1k_1}\ldots a_{mk_m}\det(B_{k_1\ldots k_m})$$

$$= \sum_{1\le k_1,\ldots,k_m\le n} \epsilon(k_1, \ldots, k_m)a_{1k_1}\ldots a_{mk_m}\det(B_{j_1\ldots j_m})$$

[where the j's are related to the k's as above]

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

Finally, if two j's are equal, $\det(A_{j_1\ldots j_m}) = 0$.

SECTION 1.2.4

1. 1, -2, -1, 0, 5.

2. $\lfloor x \rfloor$.

3. $\lfloor x \rfloor$ is the greatest integer less than or equal to x, by definition; therefore, $\lfloor x \rfloor$ is an integer, $\lfloor x \rfloor \leq x$, and $\lfloor x \rfloor + 1 > x$. The latter properties, plus the fact that when m, n are integers $n < m$ if and only if $n \leq m - 1$, lead to an easy proof of propositions (a) and (b). Similar arguments prove (c) and (d). Finally, (e) and (f) are just combinations of previous parts of this exercise.

4. $x - 1 < \lfloor x \rfloor \leq x$; so $-x + 1 > -\lfloor x \rfloor \geq -x$; hence the result.

5. $\lfloor x + \frac{1}{2} \rfloor$. The value of ($-x$ rounded) will be the same as $-$(x rounded), *except* when $x \bmod 1 = \frac{1}{2}$, when the negative value is rounded towards zero and the positive value is rounded away from zero.

6. (a) is true: $\lfloor \sqrt{x} \rfloor = n$ iff $n^2 \leq x < (n+1)^2$ iff $n^2 \leq \lfloor x \rfloor < (n+1)^2$ iff $\lfloor \sqrt{\lfloor x \rfloor} \rfloor = n$. Similarly, (b) is true. But (c) fails, e.g. for $x = 1.1$.

7. Apply Ex. 3 and Eq. (4). The inequality should be $\geq$ for ceilings, and then equality holds if and only if either x or y is an integer or $x \bmod 1 + y \bmod 1 \geq 1$.

8. 1, 2, 5, -100.

9. -1, 0, -2.

10. 0.1, 0.01, -0.09.

11. $x = y$.

12. All.

13. $+1$, -1.

14. 8.

15. Multiply both sides of Eq. (1) by z; if $y = 0$, the result is also easily verified.

17. As an example, consider the multiplication portion of law A: we have $a = b + qm$, $x = y + rm$ for some integers q, r; so $ax = by + (br + yq + qrm)m$.

18. We have $a - b = kr$ for some integer k, and also $kr \equiv 0$ (modulo s). Hence by law B, $k \equiv 0$ (modulo s), so $a - b = qsr$ for some integer q.

20. Multiply both sides of the congruence by a'.

21. There is at least one such representation, by the previously proved exercise. If there are two representations, $n = p_1 \ldots p_k = q_1 \ldots q_m$, we have $q_1 \ldots q_m \equiv 0$ (modulo p_1); so if none of the q's equals p_1 we could cancel them all by law B and obtain $1 \equiv 0$ (modulo p_1). The latter is impossible since p_1 is not equal to 1 (this is the principal reason we do not allow 1 as a prime number). So some q_j equals p_1, and $n/p_1 = p_2 \ldots p_k = q_1 \ldots q_{j-1} q_{j+1} \ldots q_m$. Either n is prime, when the result is clearly true, or by induction the two factorizations of n/p_1 are the same.

22. If $a = cd$, $m = nd$, then $an \equiv 0$ but $n \not\equiv 0$ (modulo m) if $d > 1$, $m \neq 0$.

24. Law A is always valid for addition and subtraction; Law C is always valid.

26. If b is not a multiple of p, then $b^2 - 1$ is, so one of the factors must be.

27. A number is relatively prime to p^e if and only if it is not a multiple of p. So we count those which are not multiples of p and get $\varphi(p^e) = p^e - p^{e-1}$.

28. If a, b are relatively prime to m, so is (ab mod m), since any prime dividing the latter and m must divide a or b also. Now simply let $x_1, \ldots, x_{\varphi(m)}$ be the numbers relatively prime to m, and observe that $ax_1 \bmod m, \ldots, ax_{\varphi(m)} \bmod m$ are the same numbers in some order, etc.

29. We prove (b): if r, s are relatively prime and if k^2 divides rs, then p^2 divides rs for some prime p, so p divides r (say) and cannot divide s; so p^2 divides r. We see that $f(rs) = 0$ iff $f(r) = 0$ or $f(s) = 0$.

30. Let r, s be relatively prime. The idea is to prove that the $\varphi(rs)$ numbers relatively prime to rs are precisely the $\varphi(r)\varphi(s)$ distinct numbers $(sx_i + ry_j) \bmod (rs)$ where $x_1, \ldots, x_{\varphi(r)}$ and $y_1, \ldots, y_{\varphi(s)}$ are the corresponding values for r and s.

We then find $\varphi(10^6) = \varphi(2^6)\varphi(5^6) = (2^6 - 2^5)(5^6 - 5^5) = 400000$; $\varphi(p_1^{e_1} \ldots p_r^{e_r}) = (p_1^{e_1} - p_1^{e_1-1}) \ldots (p_r^{e_r} - p_r^{e_r-1})$; $\varphi(n) = n\prod_{p\backslash n, p \text{ prime}} (1 - 1/p)$. Another proof is in exercise 1.3.3–30.

31. The divisors of rs may be uniquely written in the form cd where c divides r and d divides s. Similarly, if $f(n) \geq 0$, we find the function $\max_{d\backslash n} f(d)$ is multiplicative. Cf. Ex. 1.2.3–35.

33. Either $n + m$ or $n - m + 1$ is even, so one of the quantities inside the brackets at the left is an integer, so equality holds in Ex. 7.

34. b must be an integer ≥ 2. (Set $x = b$). The sufficiency is proved as in Ex. 6. The same condition is necessary and sufficient for $\lceil \log_b x \rceil = \lceil \log_b \lceil x \rceil \rceil$.

More generally we have the following pretty generalization due to R. McEliece: Let f be a continuous, strictly increasing function defined on an interval A, and assume x in A implies that both $\lfloor x \rfloor$ and $\lceil x \rceil$ are in A. Then the relation $\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor$ holds for all x in A iff the relation $\lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$ holds for all x in A iff we have the following condition: "$f(x)$ is an integer implies x is an integer." The condition is obviously necessary, for if $f(x)$ is an integer and it equals $\lfloor f(\lfloor x \rfloor) \rfloor$ or $\lceil f(\lceil x \rceil) \rceil$ then x must equal $\lfloor x \rfloor$ or $\lceil x \rceil$. Conversely if e.g. $\lfloor f(\lfloor x \rfloor) \rfloor < \lfloor f(x) \rfloor$ then by continuity there is some y with $\lfloor x \rfloor < y \leq x$ for which $f(y)$ is an integer, and y cannot be an integer.

35.
$$\frac{x+m}{n} - 1 = \frac{x+m}{n} - \frac{1}{n} - \frac{n-1}{n} < \frac{\lfloor x \rfloor + m}{n} - \frac{n-1}{n} \leq \left\lfloor \frac{\lfloor x \rfloor + m}{n} \right\rfloor \leq \frac{x+m}{n};$$

apply Ex. 3. Use of Ex. 4 gives a similar result for the ceiling function. Both identities follow as a special case of McEliece's theorem in Ex. 34.

36. Assume first $n = 2t$.

$$\sum_{1 \leq k \leq n} \lfloor k/2 \rfloor = \sum_{1 \leq k \leq n} \lfloor (n+1-k)/2 \rfloor,$$

hence

$$\sum_{1 \leq k \leq n} \lfloor k/2 \rfloor = \frac{1}{2}\left(\sum_{1 \leq k \leq n} (\lfloor k/2 \rfloor + \lfloor (n+1-k)/2 \rfloor) \right)$$
$$= \frac{1}{2} \sum_{1 \leq k \leq n} \lfloor (2t+1)/2 \rfloor = t^2 = n^2/4.$$

(Cf. Ex. 33.) And if $n = 2t + 1$, we have $t^2 + \lfloor n/2 \rfloor = t^2 + t = n^2/4 - \frac{1}{4}$. For the second sum we get, similarly, $\lceil n(n+2)/4 \rceil$.

37. $$\sum_{0 \le k < n} \frac{mk + x}{n} = \frac{m(n-1)}{2} + x.$$

Let $\{y\}$ denote y mod 1; we must subtract

$$S = \sum_{0 \le k < n} \left\{ \frac{mk + x}{n} \right\};$$

S consists of d copies of the same sum, since if $t = n/d$,

$$\left\{ \frac{mk + x}{n} \right\} = \left\{ \frac{m(k+t) + x}{n} \right\}.$$

Let $u = m/d$; then

$$\sum_{0 \le k < t} \left\{ \frac{mk + x}{n} \right\} = \sum_{0 \le k < t} \left\{ \frac{x}{n} + \frac{uk}{t} \right\},$$

and since t and u are relatively prime this sum may be rearranged as

$$\left\{ \frac{x \bmod d}{n} \right\} + \left\{ \frac{x \bmod d}{n} + \frac{1}{t} \right\} + \cdots + \left\{ \frac{x \bmod d}{n} + \frac{t-1}{t} \right\}.$$

Finally, since $(x \bmod d)/n < 1/t$, the brackets in this sum may be removed and we have

$$S = d\left(\frac{t(x \bmod d)}{n} + \frac{t-1}{2} \right).$$

Applying Ex. 4, we get

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

This formula would become symmetric in m and n if it were extended over the range $0 \le k \le n$. (The symmetry can be explained by drawing the graph of the summand as a function of k, then reflecting about the line $y = x$.)

38. Take $m = 1$ in the preceding exercise. The ceiling function does not behave so nicely.

39. Proof of (f): Consider the more general identity $\prod_{0 \le k < n} 2 \sin \pi(x + k/n) = 2 \sin \pi n x$, which can be demonstrated as follows: Since $2 \sin \theta = (e^{i\theta} - e^{-i\theta})/i = (1 - e^{-2i\theta}) e^{i\theta - i\pi/2}$, the identity is a consequence of the two formulas

$$\prod_{0 \le k < n} (1 - e^{-2\pi(x + k/n)}) = 1 - e^{-2\pi n x} \quad \text{and} \quad \prod_{0 \le k < n} e^{\pi(x - (1/2) + (k/n)} = e^{\pi(nx - 1/2))}.$$

The latter is true since the function $x - \frac{1}{2}$ is replicative; and the former is true because we may set $z = 1$ in the factorization of the polynomial $z^n - \alpha^n = (z - \alpha)(z - \omega\alpha) \ldots (z - \omega^{n-1}\alpha)$, $\omega = e^{-2\pi i/n}$.

The identity in (i) is easily verified for $0 \le x < 1/n$, and we also find the function

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

is periodic with period $1/n$.

41. We want $a_n = k$ when

$$\frac{k(k-1)}{2} < n \le \frac{k(k+1)}{2}.$$

The equation $k(k+1)/2 = r$ can be solved for k (the positive root):

$$k = \frac{-1+\sqrt{1+8r}}{2}.$$

Hence

$$\frac{-1+\sqrt{1+8n}}{2} \le k < \frac{-1+\sqrt{1+8n}}{2} + 1;$$

the answer is, by Ex. 3,

$$\left\lceil \frac{-1+\sqrt{1+8n}}{2} \right\rceil.$$

Another correct formula, which is not *obviously* the same, is $\lfloor (1+\sqrt{8n-7})/2 \rfloor$.

42. (a) Cf. Ex. 1.2.7–10. (b) The given sum is $n\lfloor \log_b n \rfloor - S$,

$$S = \sum_{\substack{1 \le k < n \\ k+1 \text{ is power of } b}} k = \sum_{1 \le t \le \log_b n} (b^t - 1) = (b^{\lfloor \log_b n \rfloor + 1} - b)/(b-1) - \lfloor \log_b n \rfloor.$$

43. $\lfloor \sqrt{n} \rfloor \left(n - \dfrac{(2\lfloor \sqrt{n} \rfloor + 5)(\lfloor \sqrt{n} \rfloor - 1)}{6} \right).$

45. $\lfloor mj/n \rfloor = r$ if and only if

$$\left\lceil \frac{rn}{m} \right\rceil \le j < \left\lceil \frac{(r+1)n}{m} \right\rceil,$$

and we find the given sum is therefore

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

The stated result follows by rearranging the latter sum, grouping the terms with a particular value of $\lceil rn/m \rceil$. The second formula is immediate by the substitution

$$f(x) = \binom{x+1}{k}.$$

46. $\sum_{0 \le r < \alpha m} \lceil rn/m \rceil (f(r-1) - f(r)) + \lceil \alpha n \rceil f(\lceil \alpha m \rceil - 1)$; prove like Ex. 45.

47. (a) The numbers $2, 4, \ldots, p-1$ are the even residues (modulo p); since $2kq = p\lfloor 2kq/p \rfloor + (2kq) \bmod p$, the number $(-1)^{\lfloor 2kq/p \rfloor}((2kq) \bmod p)$ will be an even

residue or an even residue minus p, and each even residue clearly occurs just once. Hence $(-1)^\sigma q^{(p-1)/2} 2 \cdot 4 \ldots (p-1) \equiv 2 \cdot 4 \ldots (p-1)$. (b) Let $q = 2$. If $p = 4n+1$, $\sigma = n$; if $p = 4n+3$, $\sigma = n+1$. Hence $(\frac{2}{p}) = 1, -1, -1, 1$ according as $p \bmod 8 = 1, 3, 5, 7$, respectively. (c) For $k < p/4$,

$$\lfloor (p-1-2k)q/p \rfloor = q - \lceil (2k+1)/p \rceil = q - 1 - \lfloor (2k+1)/p \rfloor$$
$$\equiv \lfloor (2k+1)/p \rfloor \text{ (modulo 2)}.$$

Hence we may replace the last terms $\lfloor (p-1)q/p \rfloor$, $\lfloor (p-3)q/p \rfloor$, ... by $\lfloor q/p \rfloor$, $\lfloor 3q/p \rfloor$, etc. (d) $\sum_{0 \le k < p/2} \lfloor kq/p \rfloor + \sum_{0 \le r < q/2} \lceil rp/q \rceil = \lceil p/2 \rceil (\lceil q/2 \rceil - 1) = (p+1)(q-1)/4$. Also $\sum_{0 \le r < q/2} \lceil rp/q \rceil = \sum_{0 \le r < q/2} \lfloor rp/q \rfloor + (q-1)/2$. The idea of this proof goes back to G. Eisenstein, *Journal f. d. reine und angewandte Math.* **28** (1844), 246–248; Eisenstein also gives several other proofs of this and other reciprocity laws in the same volume.

48. (a) This is clearly not always true when $n < 0$; when $n > 0$ it is easy to verify. (b) $\lfloor (n + 2 - \lfloor n/25 \rfloor)/3 \rfloor = \lceil (n - \lfloor n/25 \rfloor)/3 \rceil = \lceil (n + \lceil -n/25 \rceil)/3 \rceil = \lceil \lceil 24n/25 \rceil /3 \rceil =$ (cf. Ex. 35) $\lceil 8n/25 \rceil = \lfloor (8n+24)/25 \rfloor$.

SECTION 1.2.5

1. 52! For the curious, this number is 806 58175 17094 38785 71660 63685 64037 66975 28950 54408 83277 82400 00000 00000. (!)

2. $p_{nk} = p_{n(k-1)}(n - k + 1)$. After the first $n - 1$ objects have been placed, there is only one left, only one choice for the last object. But this does *not* mean that the last object is always the same in all permutations!

3. 53124, 35124, 31524, 31254, 31245; 42351, 41352, 41253, 31254, 31245.

4. There are 2568 digits. The leading digit is 4 (since $\log_{10} 4 = 2 \log_{10} 2 \approx .602$). The least significant digit is zero, and in fact by Eq. (8) the low order 249 digits are all zero! The exact value of 1000! was calculated by H. S. Uhler using a desk calculator and much patience over a period of several years, and the value appears in *Scripta Mathematica* 21 (1955), pp. 266–267. It begins with 402 38726 00770 . . .

5. $(39902)(97/96) \approx 416 + 39902 = 40318$.

6. $2^{18} \cdot 3^8 \cdot 5^4 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$.

8. It is $m^n m!/\big((n+m)!/n!\big) = n!$ times $m^n/(m+1)\ldots(m+n)$. The latter quantity approaches one since $m/(m+k) \to 1$.

9. $\sqrt{\pi}$ and $-2\sqrt{\pi}$. (Ex. 10 used.)

10. Yes, except when $x = 0$ or a negative integer. For we have

$$(x+1) = x \lim_{m\to\infty} \frac{m^x m!}{x(x+1)\cdots(x+m)} \left(\frac{m}{x+m+1}\right).$$

11, 12.
$$\begin{aligned}\mu &= (a_k p^{k-1} + \cdots + a_1) + (a_k p^{k-2} + \cdots + a_2) + \cdots + a_k \\ &= a_k(p^{k-1} + \cdots + p + 1) + \cdots + a_1 \\ &= \big(a_k(p^k - 1) + \cdots + a_0(p^0 - 1)\big)/(p-1) \\ &= (n - a_k - \cdots - a_1 - a_0)/(p-1).\end{aligned}$$

13. For each n, $1 \le n < p$, determine n' as in Ex. 1.2.4–19. We have a unique such n' by law 1.2.4B; and $(n')' = n$. Therefore we can pair off the numbers in groups of two, provided $n' \neq n$. If $n' = n$, we have $n^2 \equiv 1$ (modulo p); hence, as in Ex. 1.2.4–26, $n = 1$ or $n = p - 1$. So $(p-1)! \equiv 1 \cdot 1 \cdot \ldots \cdot (-1)$, since $+1$ and $p - 1$ are the only unpaired elements.

14. Of the numbers $1, 2, \ldots, n$ which are *not* multiples of p, there are $\lfloor n/p \rfloor$ complete sets of $p - 1$ consecutive elements, each with a product congruent to (-1) (modulo p) by Wilson's Theorem. There are also a_0 left over, which are congruent to $a_0!$ (modulo p), so the contribution from those factors which are not multiples of p is $(-1)^{\lfloor n/p \rfloor} a_0!$. The contribution from the factors which *are* multiples of p is the same as the contribution in $\lfloor n/p \rfloor!$; this argument can therefore be repeated to get the desired formula.

15. $(n!)^3$. There are $n!$ terms. Each term has one from each row and each column, so each term has the value $(n!)^2$.

16. The terms do not approach zero, since the coefficients approach $1/e$.

17. Express the Gamma functions as limits by Eq. (15).

18. $$\prod_{n \ge 1} \frac{n}{(n - 1/2)} \frac{n}{(n + 1/2)} = \frac{\Gamma(\frac{1}{2})\Gamma(\frac{3}{2})}{\Gamma(1)\Gamma(1)} = 2\Gamma(\tfrac{3}{2})^2.$$

19. (a) Change of variable $t = mt$. (b) Integration by parts. (c) Induction.

20. [For completeness, we prove the stated inequality. Start with the easily verified inequality $1 + x \le e^x$; set $x = \pm t/n$ and raise to the n-th power to get $(1 \pm t/n)^n \le e^{\pm t}$. Hence

$$\begin{aligned} e^{-t} \ge (1 - t/n)^n &= e^{-t}(1 - t/n)^n e^t \ge e^{-t}(1 - t/n)^n(1 + t/n)^n \\ &= e^{-t}(1 - t^2/n^2)^n \ge e^{-t}(1 - t^2/n) \end{aligned}$$

by Ex. 1.2.1–9.]

Now the given integral minus $\Gamma_m(x)$ is

$$\int_m^\infty e^{-t} t^{x-1}\, dt + \int_0^m \left(e^{-t} - \left(\frac{1-t}{m}\right)^m\right) t^{x-1}\, dt.$$

As $m \to \infty$, the first of these approaches zero, since for large t, $t^{x-1} < e^{t/2}$; and the second is less than

$$\frac{1}{m}\int_0^m t^{x+1} e^{-t} < \frac{1}{m}\int_0^\infty t^{x+1} e^{-t}\, dt \to 0.$$

21. If $c(n, j, k_1, k_2, \ldots)$ denotes the appropriate coefficient, we find

$$\begin{aligned} c(n+1, j, k_1, \ldots) = {} & c(n, j-1, k_1 - 1, k_2, \ldots) \\ & + (k_1 + 1)c(n, j, k_1 + 1, k_2 - 1, k_3, \ldots) \\ & + (k_2 + 1)c(n, j, k_1, k_2 + 1, k_3 - 1, k_4, \ldots) + \cdots, \end{aligned}$$

by differentiation. The equations $k_1 + k_2 + \cdots = j$ and $k_1 + 2k_2 + \cdots = n$ are preserved in this induction relationship. We can easily factor $n!/k_1!(1!)^{k_1} k_2!(2!)^{k_2} \ldots$ out of each term appearing on the righthand side of the equation for $c(n+1, j, k_1, \ldots)$, and we are left with $k_1 + 2k_2 + 3k_3 + \cdots = n + 1$. (In the proof it is convenient to assume there are infinitely many k's, although clearly $k_{n+1} = k_{n+2} = \cdots = 0$.) For a table of these coefficients, see the reference at the end of Section 1.2.9.

The solution just given makes use of standard techniques of summation, but it does not give a satisfactory explanation of *why* the formula has this form, nor how it could have been discovered in the first place. Let us examine this question using a combinatorial argument. Write for convenience $w_j = D_u^j w$, $u_k = D_x^k u$. Then $D_x(w_j) = w_{j+1}u_1$ and $D_x(u_k) = u_{k+1}$. By these two rules and the rule for derivative of a product we find

$$\begin{aligned} D_x^1 w &= w_1 u_1 \\ D_x^2 w &= (w_2 u_1 u_1 + w_1 u_2) \\ D_x^3 w &= \big((w_3 u_1 u_1 u_1 + w_2 u_2 u_1 + w_2 u_1 u_2) + (w_2 u_1 u_2 + w_1 u_3)\big), \text{ etc.} \end{aligned}$$

Analogously we may set up a corresponding tableau of set partitions thus:

$$\begin{aligned} \mathfrak{D}^1 &= \{1\} \\ \mathfrak{D}^2 &= (\{2\}\{1\} + \{2, 1\}) \\ \mathfrak{D}^3 &= \big((\{3\}\{2\}\{1\} + \{3, 2\}\{1\} + \{2\}\{3, 1\}) + (\{3\}\{2, 1\} + \{3, 2, 1\})\big), \text{ etc.} \end{aligned}$$

Formally, if $a_1 a_2 \ldots a_j$ is a partition of the set $\{1, 2, \ldots, n-1\}$, define

$$\begin{aligned} \mathfrak{D}a_1a_2\ldots a_j = {} & (\{n\}a_1a_2\ldots a_j + (a_1 \cup \{n\})a_2\ldots a_j + a_1(a_2 \cup \{n\})\ldots a_j + \cdots \\ & + a_1a_2\ldots(a_j \cup \{n\}). \end{aligned}$$

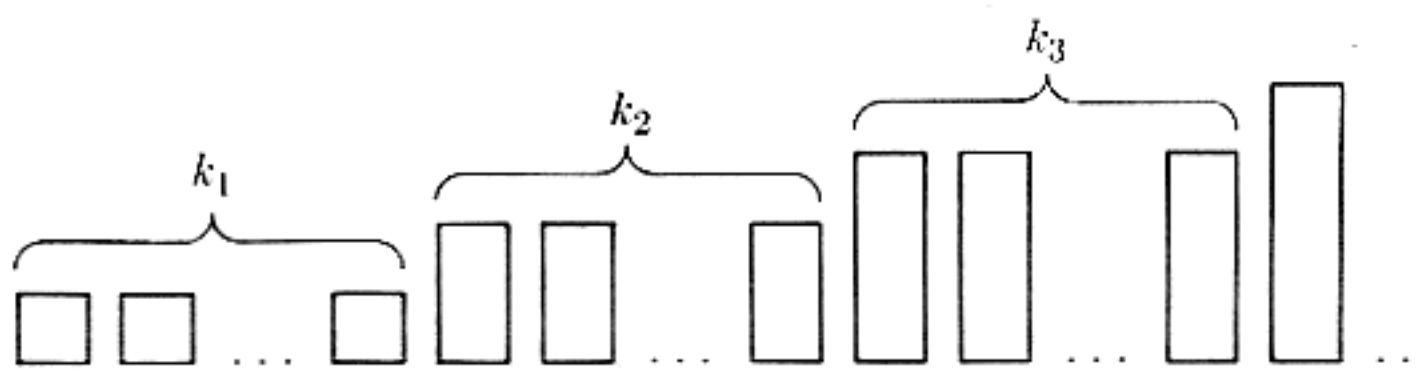
This rule is an exact parallel of the rule

$$\begin{aligned} D_x(w_j u_{r_1} u_{r_2} \ldots u_{r_j}) = {} & w_{j+1} u_1 u_{r_1} u_{r_2} \ldots u_{r_j} + w_j u_{r_1+1} u_{r_2} \ldots u_{r_j} \\ & + w_j u_{r_1} u_{r_2+1} \ldots u_{r_j} + \cdots + w_j u_{r_1} u_{r_2} \ldots u_{r_j+1}, \end{aligned}$$

if we let the term $w_j u_{r_1} u_{r_2} \ldots u_{r_j}$ correspond to a partition $a_1 a_2 \ldots a_j$ with r_t elements in a_t, $1 \le t \le j$. So there is a natural mapping from $\mathfrak{D}^n$ onto $D_x^n w$, and furthermore it is easy to see that $\mathfrak{D}^n$ includes each partition of the set $\{1, 2, \ldots, n\}$ exactly once. (Cf. Ex. 1.2.6–64.)

From these observations we find that if we collect like terms in $D_x^n w$, we obtain a sum of terms $c(k_1, k_2, \ldots) w_j u_1^{k_1} u_2^{k_2} \ldots$, where $j = k_1 + k_2 + \cdots$ and $n = k_1 + 2k_2 + \cdots$, and where $c(k_1, k_2, \ldots)$ is *the number of partitions of* $\{1, 2, \ldots, n\}$ *into* j *parts such that there are* k_t *parts having* t *elements.*

It remains to count these partitions. Consider an array of k_t boxes of capacity t:



The number of ways to put n different elements into these boxes is the multinomial coefficient

$$\binom{n}{1\,1\ldots1\,2\,2\ldots2\,3\,3\ldots3\,4\ldots} = n!/1!^{k_1}2!^{k_2}3!^{k_3}\ldots;$$

to get $c(k_1, k_2, k_3, \ldots)$ we should divide this by $k_1!k_2!k_3!\ldots$ since the boxes in each group of k_t are indistinguishable from each other and may be permuted in $k_t!$ ways without affecting the set partition. (This solution is by R. McEliece.)

Faa di Bruno's formula has been generalized in several ways. For what is perhaps the most extensive generalization of the formula, and a list of references to other related work, see the paper by I. J. Good, *Annals of Mathematical Statistics* **32** (1961), 540–541.

22. The hypothesis that $\lim_{n\to\infty} (n + x)!/n!n^x = 1$ is valid for integers x; for example, if x is positive, the quantity is $(1 + 1/n)(1 + 2/n) \cdots (1 + x/n)$, which certainly approaches one. If we also assume that $x! = x(x - 1)!$, the hypothesis leads us to conclude immediately that

$$1 = \lim_{n\to\infty} \frac{(n+x)!}{n!n^x} = x! \lim_{n\to\infty} \frac{(x+1)\ldots(x+n)}{n!n^x},$$

which is equivalent to the definition given in the text.

SECTION 1.2.6

1. n, since each combination leaves out one item.

2. 1.

3. $\binom{52}{13}$. The actual number is 635013559600.

4. $2^4 5^2 7^2 17 \cdot 23 \cdot 41 \cdot 43 \cdot 47$.

5. $(10+1)^4 = 10000 + 4(1000) + 6(100) + 4(10) + 1$.

6.

$n = -3$:	1	-3	6	-10	15	-21	28	-36	...
$n = -2$:	1	-2	3	-4	5	-6	7	-8	...
$n = -1$:	1	-1	1	-1	1	-1	1	-1	...

7. $\lfloor n/2 \rfloor$; or, alternatively, $\lceil n/2 \rceil$. It is clear from (3) that for smaller values the binomial coefficient is strictly increasing, and afterwards it decreases to zero.

8. The nonzero entries in each row read the same from left to right as from right to left.

9. One if n is positive or zero; zero if n is negative.

10. (a), (b), (f) follow immediately from (e), and (c), (d) follow from (a), (b), and Eq. (9). Thus it suffices to prove (e). Consider $\binom{n}{k}$ as a fraction, given by Eq. (3) with factors in numerator and denominator. The first $k \bmod p$ factors have no p's in the denominator, and in the numerator and denominator these terms are clearly congruent to the corresponding terms of

$$\binom{n \bmod p}{k \bmod p},$$

which differ by multiples of p. (When dealing with non-multiples of p we may work modulo p in both numerator and denominator, since if $a \equiv c$ and $b \equiv d$ and a/b, c/d are integers, then $a/b \equiv c/d$.) There remain $k - k \bmod p$ factors, which fall into $\lfloor k/p \rfloor$ groups of p consecutive values each. Each group contains exactly one multiple of p; the other $(p-1)$ factors in a group are congruent (modulo p) to $(p-1)!$ so they cancel in numerator and denominator. It remains to investigate the $\lfloor k/p \rfloor$ multiples of p in numerator and denominator; we divide each of these by p and are left with the binomial coefficient

$$\binom{\lfloor (n - k \bmod p)/p \rfloor}{\lfloor k/p \rfloor}.$$

If $k \bmod p \le n \bmod p$, this equals

$$\binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}$$

as desired, but if $k \bmod p > n \bmod p$, this equals

$$\binom{\lfloor n/p \rfloor - 1}{\lfloor k/p \rfloor}.$$

However, the other factor

$$\binom{n \bmod p}{k \bmod p}$$

is then zero, so the formula is true in general. [See also N. J. Fine, *AMM* **54** (1947), 589–592.]

11. If

$$\begin{aligned} a &= a_r p^r + \cdots + a_0, \\ b &= b_r p^r + \cdots + b_0, \\ a + b &= c_r p^r + \cdots + c_0, \end{aligned}$$

the value (according to Ex. 1.2.5–12 and Eq. (5)) is

$$(a_0 + \cdots + a_r + b_0 + \cdots + b_r - c_0 - \cdots - c_r)/(p - 1).$$

A carry decreases c_j by p and increases c_{j+1} by 1, giving a net change of $+1$ in this formula.

12. By either of the two previous exercises, n must be one less than a power of 2. More generally, $\binom{n}{k}$ is never divisible by the prime p, $0 \le k \le n$, if and only if $n = ap^m - 1$, $1 \le a < p$, $m \ge 0$.

14. $$24\binom{n+1}{5} + 36\binom{n+1}{4} + 14\binom{n+1}{3} + \binom{n+1}{2}$$
$$= \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30} = \frac{n(n+1)(n+\frac{1}{2})(3n^2+3n-1)}{15}.$$

15. Induction and (9).

17. We may assume r, s are positive integers. Also

$$\sum_n \binom{r+s}{n} x^n = (1+x)^{r+s} = \sum_n \binom{r}{n} x^n \sum_k \binom{s}{k} x^k$$
$$= \sum_n \sum_k \binom{r}{n-k} x^{n-k} \binom{s}{k} x^k = \sum_n \left(\sum_k \binom{r}{n-k} \binom{s}{k} \right) x^n$$

for all x, so the coefficients of x^n must be identical.

21. The lefthand side is a polynomial of degree n, the righthand side is a polynomial of degree $m + n + 1$. We have agreement at $n + 1$ points. This is not enough to prove them equal (although when $m = 0$ it proves that the two sides are multiples of some polynomial; and indeed in the case $m = 0$ we find that the equation *is* an identity in s, since it is Eq. (11)).

22. Assume $n > 0$. The k-th term is

$$\frac{1}{n!}\binom{n}{k}\prod_{0<j<k}(r-tk-j)\prod_{0<j<n-k}(n-1-r+tk-j)$$
$$=\frac{(-1)^{k-1}}{n!}\binom{n}{k}\prod_{0<j<k}(-r+tk+j)\prod_{k\le j<n}(-r+tk+j)$$

and the two products give a polynomial of degree $(n-1)$ in k, so the sum over k is zero by Eq. (35).

24. The proof is by induction on n. If $n \le 0$ the identity is obvious. If $n > 0$, we prove it holds for $(r,\ n-r+nt+m,\ t,\ n)$, by induction on the integer $m \ge 0$, using the previous two exercises and the validity for $n-1$. This establishes the identity (r, s, t, n) for infinitely many s, and being a polynomial in s it holds for all s.

25. Using the ratio test and straightforward estimates for large values of k we can prove convergence. (Alternatively using complex variable theory we know the function is analytic in the neighborhood of $x = 1$.) We have

$$1=\sum_{k,j}(-1)^j\binom{k}{j}\binom{r-jt}{k}\frac{r}{r-jt}w^k=\sum_j(-1)^j\frac{r}{r-jt}\sum_k\binom{k}{j}\binom{r-jt}{k}w^k$$
$$=\sum_j\frac{r(-1)^j}{r-jt}\sum_k\binom{r-jt}{j}\binom{r-jt-j}{k-j}w^k$$
$$=\sum_j(-1)^jA_j(r,t)(1+w)^{r-jt-j}w^j.$$

Now let $x = 1/(1+w)$, $z = -w/(1+w)^{1+t}$. This proof is due to H. W. Gould [*AMM* **63** (1956), 84–91]. See also the simple but less elementary derivation by M. Skalsky, *AMM* **69** (1962), 404–405, and the more general formula in exercise 2.3.4.4–33.

26. We could start with the identity

$$\sum_j(-1)^j\binom{k}{j}\binom{r-jt}{k}=t^k$$

and proceed as above. Another way is to differentiate our formula with respect to z; we get

$$\sum_k kA_k(r,t)z^k=z\frac{d(x^r)}{dz}=\frac{(x^{t+1}-x^t)rx^r}{((t+1)x^{t+1}-tx^t)},$$

hence we can obtain the value of

$$\sum_k\left(1-\frac{t}{r}k\right)A_k(r,t)z^k.$$

27. For Eq. (26), multiply the series for $x^{r+1}/((t+1)x-t)$ by the series for x^s, get a series for $x^{r+s+1}/((t+1)x-t)$ in which coefficients of z may be equated to the coefficients arising from the series for $x^{(r+s)+1}/((t+1)x-t)$.

28. Denoting the lefthand side by $f(r, s, t, n)$, we find

$$\binom{r+s}{n} + t \cdot f(r-t-1, s+t, t, n-1) = f(r, s, t, n)$$

by considering the identity

$$\sum_k \binom{r+tk}{k}\binom{s-tk}{n-k}\frac{r}{r+tk} + \sum_k \binom{r+tk}{k}\binom{s-tk}{n-k}\frac{tk}{r+tk} = f(r, s, t, n).$$

29. $(-1)^k \binom{n}{k} \Big/ n! = (-1)^k/k!(n-k)! = (-1)^n \Big/ \prod_{\substack{0 \le j \le n \\ j \ne k}} (k-j).$

30. Apply (7) and (19) to get

$$\sum_{k \ge 0} \binom{-m-2k-1}{n-m-k}\binom{2k+1}{k+1}\frac{(-1)^{n-m}}{2k+1} = \sum_{k \ge 1} \binom{-m-2k+1}{n-m-k+1}\binom{2k-1}{k}\frac{(-1)^{n-m}}{2k-1}.$$

Now if we add the term for $k = 0$ we can apply Eq. (26) with

$$r = -1, \quad s = m - 2n - 1, \quad t = -2, \quad n = n - m + 1.$$

So we get the result

$$\binom{-m}{n-m+1}(-1)^{n-m+1} + \binom{-m+1}{n-m+1}(-1)^{n-m} = \binom{-m}{n-m}(-1)^{n-m} = \binom{n-1}{n-m}.$$

This result is the same as our previous formula, when n is positive, but when $n = 0$ the answer we have obtained is correct while $\binom{n-1}{m-1}$ is not. Our derivation has a further bonus, since the answer $\binom{n-1}{n-m}$ is valid for $n \ge 0$ and *all* integers m.

31. We have

$$\begin{aligned}
&\sum_k \sum_j \binom{m-r+s}{j}\binom{n+r-s}{n-k}\binom{r}{m+n-j}\binom{k}{j} \\
&= \sum_j \sum_k \binom{m-r+s}{j}\binom{n+r-s}{n-k}\binom{r}{m+n-j}\binom{m-r+s-j}{k-j} \\
&= \sum_j \binom{m-r+s}{j}\binom{r}{m+n-j}\binom{m+n-j}{n-j}.
\end{aligned}$$

Changing

$$\binom{m+n-j}{n-j} \quad \text{to} \quad \binom{m+n-j}{m}$$

and applying (20) again, we get

$$\sum_j \binom{m-r+s}{j}\binom{r}{m}\binom{r-m}{n-j} = \binom{r}{m}\binom{s}{n}.$$

32. Replace x by $-x$ in (40).

33, 34. We have

$$x^{\overline{n}} = n!\binom{x+n-1}{n}.$$

The equation may therefore be transformed into

$$\binom{x+y+n-1}{n} = \sum_k \binom{x+(1-z)k}{k}\binom{y-1+nz+(n-k)(1-z)}{n-k}\frac{x}{x+(1-z)k}$$

which is a case of (26).

35. For example, we prove the first formula:

$$\sum_k (-1)^{n-k}\left((n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}\right) x^k$$

$$= -(n-1)(n-1)!\binom{x}{n-1} + x(n-1)!\binom{x}{n-1} = n!\binom{x}{n}.$$

36. By the binomial formula (assuming n is a nonnegative integer) we get 2^n and δ_{n0}, respectively.

37. When $n > 0$, 2^{n-1}. (The odd and even terms cancel, so each equals half the total sum.)

38. Let $\omega = e^{2\pi i/m}$. Then

$$\sum_{0\le j<m} (1+\omega^j)^n \omega^{-jk} = \sum_t \sum_{0\le j<m} \binom{n}{t} \omega^{j(t-k)}.$$

Now

$$\sum_{0\le j<m} \omega^{rj} = m\delta_{(r \bmod m)0}$$

(it is the sum of a geometric progression), so the righthand sum is

$$m \sum_{t \bmod m = k} \binom{n}{t}.$$

The original sum on the left is

$$\sum_{0\le j<m} (\omega^{-j/2} + \omega^{j/2})^n \omega^{j(n/2-k)} = \sum_{0\le j<m} \left(2\cos\frac{j\pi}{m}\right)^n \omega^{j(n/2-k)}.$$

Since the quantity is known to be real, we may take the real part and obtain the stated formula.

39. $n!$; 0 if $n \ge 2$, ± 1 in the other two cases. (The row sums in the second triangle are not so simple; we will find (Ex. 64) that this gives the number of ways to partition a set of n elements into disjoint sets, i.e. the number of equivalence relations.)

40. Proof of (c): By parts,

$$B(x+1, y) = \left.\frac{t^x(1-t)^y}{y}\right|_0^1 + \frac{x}{y}\int_0^1 t^{x-1}(1-t)^y\,dt.$$

Now use (b).

41. $m^x B(x, m+1) \to \Gamma(x)$ as $m \to \infty$, regardless of whether m runs through integer values or not (by monotonicity). Hence, $(m+y)^x B(x, m+y+1) \to \Gamma(x)$, and $(m/(m+y))^x \to 1$.

42. $1/kB(k, r-k+1)$.

43. $\int_0^1 dt/t^{1/2}(1-t)^{1/2} = 2\int_0^1 du/(1-u^2)^{1/2} = 2 \arcsin u|_0^1 = \pi$.

45. We have for large r,

$$\frac{1}{k\Gamma(k)} \sqrt{\frac{r}{r-k}} \frac{1}{e^k} \frac{(1-k/r)^k}{(1-k/r)^r} \to \frac{1}{\Gamma(k+1)}.$$

46. $\sqrt{\frac{1}{2\pi}\left(\frac{1}{x}+\frac{1}{y}\right)}\left(1+\frac{y}{x}\right)^x \left(1+\frac{x}{y}\right)^y \cdot \binom{2n}{n} \approx 4^n/\sqrt{\pi n}$.

48. This can be proved by induction, using the fact that

$$0 = \sum_k \binom{n}{k}(-1)^k = \sum_k \binom{n}{k} \frac{(-1)^k k}{k+x} + \sum_k \binom{n}{k} \frac{(-1)^k x}{k+x}.$$

Alternatively, we have

$$B(x, n+1) = \int_0^1 t^{x-1}(1-t)^n \, dt = \sum_k \binom{n}{k}(-1)^k \int_0^1 t^{x+k-1} \, dt.$$

(In fact, the stated sum equals $B(x, n+1)$ for non-integer n also, when the series converges.)

49. $\binom{r}{m} = \sum_k \binom{r}{k}\binom{-r}{m-2k}(-1)^{m+k}$, integer m.

50. The k-th summand is $\binom{n}{k}(-1)^{n-k}(x-kz)^{n-1}x$. Apply Eq. (35).

51. The righthand side is

$$\sum_k \binom{n}{n-k} x(x-kz)^{k-1} \sum_r \binom{n-k}{r}(x+y)^r(-x+kz)^{n-k-r}$$

$$= \sum_r \binom{n}{r}(x+y)^r \sum_k \binom{n-r}{n-r-k} x(x-kz)^{k-1}(-x+kz)^{n-k-r}$$

$$= \sum_r \binom{n}{r}(x+y)^r 0^{n-r} = (x+y)^n.$$

The same device may be used to prove Torelli's sum (Ex. 34) as well as the related formula

$$(x+y)^{\underline{n}} = \sum_k \binom{n}{k} x(x-kz-1)^{\underline{k-1}} (y+kz)^{\underline{n-k}},$$

where $x^{\underline{n}} = x(x-1)\cdots(x-n+1)$.

Another neat proof of Abel's formula comes from the fact that it is readily transformed into the more symmetric identity

$$\sum_k \binom{n}{k} x(x+kz)^{k-1} y(y+(n-k)z)^{n-k-1} = (x+y)(x+y+nz)^{n-1};$$

see Ex. 2.3.4.4–29.

Abel's theorem has been even further generalized by A. Hurwitz [*Acta Mathematica* **26** (1902), 199–203] as follows:

$$\sum x(x+\epsilon_1 z_1+\cdots+\epsilon_n z_n)^{\epsilon_1+\cdots+\epsilon_n-1}(y-\epsilon_1 z_1-\cdots-\epsilon_n z_n)^{n-\epsilon_1-\cdots-\epsilon_n} = (x+y)^n$$

where the sum is over all 2^n choices of $\epsilon_1, \ldots, \epsilon_n = 0$ or 1 independently. This is an identity in $x, y, z_1, \ldots, z_n$, and Abel's formula is the special case $z_1 = z_2 = \cdots = z_n$. Hurwitz's formula follows from the result in Ex. 2.3.4.4–30.

54. Insert minus signs in a checkerboard pattern as shown.

$$\begin{pmatrix} 1 & -0 & 0 & -0 \\ -1 & 1 & -0 & 0 \\ 1 & -2 & 1 & -0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

This is equivalent to multiplying a_{ij} by $(-1)^{i+j}$. The result is the desired inverse, by Eq. (34).

55. Insert minus signs as in previous exercise in one triangle, get the inverse of the other. (Eq. (43).)

56. 012 013 023 123 014 024 124 034 134 234 015 025 125 035 135 235 045 145 245 345 016. With c fixed, a and b run through the combinations of c things two at a time; with c, b fixed, a runs through the combinations of b things one at a time. Similarly, we could express all numbers $n = \binom{a}{1} + \binom{b}{2} + \binom{c}{3} + \binom{d}{4}$ with $0 \le a < b < c < d$; the sequence begins 0123 0124 0134 0234 1234 0125 0135 0235 . . . .

59. $(n+1)\binom{n}{k}$.

60. $\binom{n+k-1}{k}$. This formula can be remembered since it is

$$\frac{n(n+1)\cdots(n+k-1)}{k(k-1)\cdots 1},$$

i.e. like Eq. (2) except the numbers in the numerator go up instead of down. A slick way to prove this formula is to note that we want to count the number of integer solutions $(a_1, \ldots, a_k)$ to the relations $1 \le a_1 \le a_2 \le \cdots \le a_k \le n$. This is the same as $0 < a_1 < a_2 + 1 < \cdots < a_k + k - 1 < n + k$; and the number of solutions to $0 < b_1 < b_2 < \cdots < b_k < n + k$ is the number of choices of k distinct things from the set $\{1, 2, \ldots, n+k-1\}$. (This trick is due to H. F. Scherk, *Journal für Math.* **3** (1828), 97; curiously it was also given by W. A. Förstemann in the same journal, vol. 13 (1835), 237, who said "One would almost believe this must have been known long ago, but I have found it nowhere, even though I have consulted many works in this regard."!) The formula may be derived easily with the use of generating functions (cf. Ex. 1.2.9–16).

61. If a_{nm} is the desired quantity, we have by (42), (43), $a_{nm} = na_{(n-1)m} + (-1)^n\delta_{nm}$. Hence the answer is 0 for $n < m$, and $(-1)^m n!/m!$ for $n \geq m$. The same formula is also easily obtained by inversion of (52).

62. Let $A(l, m, n)$ be the sum on the left. The proof proceeds in several steps:

a) $$A(l, m, n+1) = 2\sum (-1)^k \binom{2l}{l+k}\binom{2m}{m+k}\binom{2n+1}{n+k+1}.$$

$$\text{[Replace } \binom{2n+2}{n+k+1} \text{ by } \binom{2n+1}{n+k+1} + \binom{2n+1}{n+k}.\text{]}$$

b) $$(l+n+1)(2m+1)A(l, m, n+1)$$

$$= 2\sum (-1)^k \binom{2l}{l+k}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1}(m+k+1)(l-k+n+k+1)$$

$$= 2\sum (-1)^k \binom{2l-1}{l-k-1}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1}(m+k+1)(2l)$$

$$+ 2\sum (-1)^k \binom{2l}{l+k}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1}(m+k+1)(n+k+1).$$

c) When $l > 0$, $$\sum (-1)^k \binom{2l-1}{l-k-1}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1} = 0.$$

(Replace k by $-k-1$.)

d) Because of (c), the last derived quantity in (b) is symmetric in m and n. Hence $(l+n+1)(2m+1)A(l, m, n+1) = (l+m+1)(2n+1)A(l, m+1, n)$.

e) From (d), the result follows by induction on n. (*Note:* It is hard to say who originated this identity; the case $l = m = n$ was published by A. C. Dixon, *Messenger of Math.* **20** (1891), 79–80. See papers by P. A. MacMahon, *Quarterly Journal of Pure and Applied Math.* **33** (1902), 274–288, and John Dougall, *Proc. Edinburgh Math. Society* **25** (1906), 114–132.)

64. Let $f(n, m)$ be the number of partitions of $\{1, 2, \ldots, n\}$ into m parts. Clearly $f(1, m) = \delta_{1m}$. If $n > 1$, the partitionings are of two varieties: (a) The element n alone forms a set of the partition; there are $f(n-1, m-1)$ ways to construct partitions like this. (b) The element n appears together with another element; there are m ways to insert n into any m-partition of $\{1, 2, \ldots, n-1\}$, hence there are $mf(n-1, m)$ ways to construct partitions like this. We therefore conclude $f(n, m) = f(n-1, m-1) + mf(n-1, m)$, and by induction $f(n, m) = \left\{{n \atop m}\right\}$.

SECTION 1.2.7

1. 0; 1; 3/2.
2. Replace each term $1/(2^m + k)$ by the upper bound $1/2^m$.
3. $S_{2^m}^{(r)} \leq \sum_{0 \leq k \leq m} (1/2^r - 1)^k$. $2^{r-1}/(2^{r-1} - 1)$ is an upper bound.
4. (b) and (c).

5. 9.78760 60360 44382 . . .

6. Induction and Eq. 1.2.6–42.

7. $T(m+1, n) - T(m, n) = 1/(m+1) - 1/(mn+1) - \cdots - 1/(mn+n) < 1/(m+1) - 1/(mn+n) - \cdots - 1/(mn+n) = 1/(m+1) - n/(m+1) \leq 0$. The maximum value occurs at $m = n = 1$, and the minimum is approached when m and n get very large. By Eq. (3) the greatest lower bound is γ, which is never actually attained. A generalization of this result appears in *AMM* **70** (1963), 575–577.

8. By Stirling's approximation, $\ln n!$ is approximately $(n + \frac{1}{2}) \ln n - n + \ln \sqrt{2\pi}$; also $\sum_{1 \leq k \leq n} H_k$ is approximately $(n+1) \ln n - n(1-\gamma) + (\gamma + \frac{1}{2})$; the difference is approximately $\gamma n + \frac{1}{2} \ln n + .158$.

9. $-1/n$.

10. Break left side into two sums, change k to $k+1$ in second sum.

11. $2 - H_{n+1}/n - 1/(n+1)$.

12. 1.000 . . . is correct to over three hundred decimal places!

14. See section 1.2.3, example 2. The second sum is $\frac{1}{2}(H_{n+1}^2 - H_{n+1}^{(2)})$.

15. $\sum_{1 \leq j \leq n} (1/j) \sum_{j \leq k \leq n} H_k$ can be summed by formulas in the text; the answer is $(n+1)H_n^2 - (2n+1)H_n + 2n$.

16. $H_{2n+1} - \frac{1}{2}H_n$.

17. Taking the denominator to be $(p-1)!$, which is a multiple of the true denominator but not a multiple of p, we must show only that the corresponding numerator, $(p-1)!/1 + (p-1)!/2 + \cdots + (p-1)!/(p-1)$, *is* a multiple of p. Modulo p, $(p-1)!/k \equiv (p-1)!k'$, where k' can be determined by the relation $kk' \bmod p = 1$. The set $\{1', 2', \ldots, (p-1)'\}$ is just the set $\{1, 2, \ldots, (p-1)\}$; so the numerator is congruent to $(p-1)!(1 + 2 + \cdots + p - 1) \equiv 0$. In fact the numerator is known to be a multiple of p^2 when $p > 3$; see Hardy and Wright, *The Theory of Numbers*, section 7.8.

18. If $n = 2^k m$ where m is odd, the sum equals $2^{2k} m_1/m_2$ where m_1 and m_2 are both odd. *AMM* **67** (1960), 924–925.

19. Only $n = 0$, $n = 1$. For $n \geq 2$, let $k = \lfloor \log_2 n \rfloor$. There is precisely one term whose denominator is 2^k, so $2^{k-1}H_n - \frac{1}{2}$ is a sum of terms involving only odd primes in the denominator. If H_n were an integer, $2^{k-1}H_n - \frac{1}{2}$ would have a denominator equal to 2.

20. Expand the integrand term by term. See also *AMM* **69** (1962), 239, and an article by H. W. Gould, *Mathematics Magazine* **34** (1961), 317–321.

21. $H_{n+1}^2 - H_{n+1}^{(2)}$.

22. $(n+2)(H_{n+1}^2 - H_{n+1}^{(2)}) - 2(n+1)H_n + 2n$.

23. $\Gamma'(n+1)/\Gamma(n+1) = 1/n + \Gamma'(n)/\Gamma(n)$, since $\Gamma(x+1) = x\Gamma(x)$. Hence $H_n = \gamma + \Gamma'(n+1)/\Gamma(n+1)$. The function $H_n - \gamma$ is called the *psi function* or the *digamma function*.

24. It is

$$x \lim_{n\to\infty} e^{(H_n - \ln n)x} \prod_{1 \leq k \leq n} \left(\left(1 + \frac{x}{k}\right) e^{-x/k} \right) = \lim_{n\to\infty} \frac{x(x+1)\cdots(x+n)}{n^x n!}.$$

Note: The generalization of H_n considered in the previous exercise is therefore equal to $H_x^{(r)} = \sum_{k\geq 0} \left(1/(k+1)^r - 1/(k+1+x)^r\right)$, when $r = 1$; the same idea can be used for larger values of r.

SECTION 1.2.8

1. F_{k+2}; the answer is $F_{14} = 377$ pairs.

2. $\ln(\phi^{1000}/\sqrt{5}) = 1000 \ln \phi - \frac{1}{2} \ln 5 = 480.40711$; $\log_{10} F_{1000}$ is $1/(\ln 10)$ times this, or $208.6\underline{4}$; F_{1000} is therefore a 209-digit number whose leading digit is 4.

4. 0, 1, 5; afterwards F_n increases too fast.

5. 0, 1, 12.

6. Induction. (The equation holds for *negative* n also, cf. Ex. 8.)

7. If d is a proper divisor of n, F_d divides F_n. Now F_d is greater than one and less than F_n provided d is greater than 2. The only non-prime number which has no proper factor greater than 2 is $n = 4$ (since if $d = 2$, $n/d \geq 2$). $F_4 = 3$ is the only exception.

8. $F_{-1} = 1$; $F_{-2} = -1$; $F_{-n} = (-1)^{n+1}F_n$ by induction on n.

9. Not (15). The others are valid, by an inductive argument which proves something true for $n - 1$ assuming it true for n and greater.

10. When n is even, it is greater, and when n is odd, it is less by Eq. (14).

11. Induction; cf. Ex. 9. This can also be observed to be a special case of Ex. 13(a).

12. If $\mathcal{G}(z) = \sum \mathcal{F}_n z^n$, $(1 - z - z^2)\mathcal{G}(z) = z + F_0 z^2 + F_1 z^3 + \cdots = z + z^2 G(z)$. Hence $\mathcal{G}(z) = G(z) + zG(z)^2$; by Eq. (17) we derive $\mathcal{F}_n = \big((3n+3)/5\big)F_n - (n/5)F_{n+1}$.

13. (a) $a_n = rF_{n-1} + sF_n$. (b) Since $(b_{n+2} + c) = (b_{n+1} + c) + (b_n + c)$, we may consider a new sequence $b'_n = b_n + c$. Apply part (a) to b'_n, and we obtain the answer $cF_{n-1} + (c+1)F_n - c$.

14. $a_n = F_{m+1}F_{n-1} + (F_{m+2} + 1)F_n - \binom{n}{m} - \binom{n+1}{m-1} - \cdots - \binom{n+m}{0}$.

15. $c_n = xa_n + yb_n + (1 - x - y)F_n$.

16. F_{n+1}. Induction, and $\binom{n+1-k}{k} = \binom{n-k}{k} + \binom{(n-1)-(k-1)}{k-1}$.

17. $(x^{n+k} - y^{n+k})(x^{m-k} - y^{m-k}) - (x^n - y^n)(x^m - y^m)$
$$= (xy)^n(x^{m-n-k} - y^{m-n-k})(x^k - y^k).$$

Now set $x = \phi$, $y = \hat{\phi}$, and divide by $(\sqrt{5})^2$.

18. It is F_{2n+1}.

19. Let $u = \cos 72°$, $v = \cos 36°$. We have $u = 2v^2 - 1$; $v = 1 - 2\sin^2 18° = 1 - 2u^2$. Hence $u + v = 2(v^2 - u^2)$, i.e. $1 = 2(v - u)$, so $4v^2 - 2v - 1 = 0$. $v = \frac{1}{2}\phi$.

20. $F_{n+2} - 1$.

21. Multiply by $x^2 + x - 1$; the solution is $(x^{n+1}F_{n+1} + x^{n+2}F_n - x)/(x^2 + x - 1)$. If the denominator is zero, x is $1/\phi$ or $1/\hat{\phi}$; then the solution is

$$\big((n+1)x^n F_{n+1} + (n+2)x^{n+1}F_n - 1\big)/(2x+1).$$

22. F_{m+2n}; see next exercise with $t = 2$.

23. $$\frac{1}{\sqrt{5}}\sum_k \binom{n}{k}(\phi^k F_t^k F_{t-1}^{n-k}\phi^m - \hat{\phi}^k F_t^k F_{t-1}^{n-k}\hat{\phi}^m)$$

$$= \frac{1}{\sqrt{5}}\left(\phi^m(\phi F_t + F_{t-1})^n - \hat{\phi}^m(\hat{\phi}F_t + F_{t-1})^n\right) = F_{m+tn}.$$

24. F_{n+1} (expand by cofactors in first row).

25. $2^n\sqrt{5}\,F_n = (1+\sqrt{5})^n - (1-\sqrt{5})^n$.

26. By Fermat's theorem, $2^{p-1} \equiv 1$; now apply the previous exercise and Ex. 1.2.6–10(b).

27. It is true if $p = 2$. Otherwise, modulo p, $F_{p-1}F_{p+1} - F_p^2 = -1$; from the previous exercise and Fermat's theorem, $F_{p-1}F_{p+1} \equiv 0$. Only one of these can be a multiple of p since $F_{p+1} = F_p + F_{p-1}$.

28. $\hat{\phi}^n$. *Note:* the solution to the relation $a_{n+1} = Aa_n + B^n$, $a_0 = 0$, is

$$a_n = (A^n - B^n)/(A - B) \text{ if } A \neq B, \qquad a_n = nA^{n-1} \text{ if } A = B.$$

29.

$\left(\!\binom{n}{0}\!\right)$	$\left(\!\binom{n}{1}\!\right)$	$\left(\!\binom{n}{2}\!\right)$	$\left(\!\binom{n}{3}\!\right)$	$\left(\!\binom{n}{4}\!\right)$	$\left(\!\binom{n}{5}\!\right)$	$\left(\!\binom{n}{6}\!\right)$
1	0	0	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	0
1	2	2	1	0	0	0
1	3	6	3	1	0	0
1	5	15	15	5	1	0
1	8	40	60	40	8	1

(b) follows from (6).

30. By induction on m, the statement being obvious when $m = 1$:

a) $$\sum_k \left(\!\binom{m}{k}\!\right)(-1)^{\lceil (m-k)/2\rceil}F_{n+k}^{m-2}F_k = F_n\sum_k \left(\!\binom{m-1}{k-1}\!\right)(-1)^{\lceil (m-k)/2\rceil}F_{n+k}^{m-2} = 0.$$

b) $$\sum_k \left(\!\binom{m}{k}\!\right)(-1)^{\lceil (m-k)/2\rceil}F_{n+k}^{m-2}(-1)^k F_{m-k}$$

$$= F_n\sum_k \left(\!\binom{m-1}{k}\!\right)(-1)^{\lceil (m-1-k)/2\rceil}F_{n+k}^{m-2}(-1)^n = 0.$$

c) Since $(-1)^k F_{m-k} = F_{k-1}F_m - F_kF_{m-1}$, we conclude from (a), (b) that

$$\sum_k \left(\!\binom{m}{k}\!\right)(-1)^{\lceil (m-k)/2\rceil}F_{n+k}^{m-2}F_{k-1} = 0,$$

since $F_m \neq 0$.

d) Since $F_{n+k} = F_{k-1}F_n + F_kF_{n+1}$ the result follows from (a) and (c). This result may also be proved in slightly more general form by using the "q-binomial theorem" in Ex. 1.2.6–58. See also J. Riordan, *Duke Math J.* **29** (1962), 5–12.

31. Ex. 8 and 11.

32. Modulo F_n the Fibonacci sequence is $0, 1, \ldots, F_{n-1}, 0, F_{n-1}, -F_{n-2}, \ldots$.

33. One can use the properties of Chebyshev polynomials, if they are known. Directly, we find $\cos z = \frac{1}{2}(e^{iz} + e^{-iz}) = -i/2$. Then use the fact that $\sin(n+1)z + \sin(n-1)z = 2\sin(nz)\cos z$.

34. Prove that the only possible value for F_{k_1} is the largest Fibonacci number less than or equal to n; hence $n - F_{k_1}$ is less than F_{k_1-1}, and by induction there is a unique representation of $n - F_{k_1}$. The outline of this proof is quite similar to the proof of the unique factorization theorem.

36. We may consider the infinite string S_∞, since S_n $(n > 1)$ consists of the first F_n letters of S_∞. There are no double a's, no triple b's. S_n contains F_{n-2} a's, F_{n-1} b's. If we express $m - 1$ in the Fibonacci number system as in Ex. 34, the m-th letter of S_∞ is a if and only if $k_r = 2$. The k-th letter of S_∞ is b if and only if $\lfloor (k+1)\phi^{-1} \rfloor - \lfloor k\phi^{-1} \rfloor = 1$; the number of b's in the first k letters is therefore $\lfloor (k+1)\phi^{-1} \rfloor$.

37. [*Fibonacci Quart.* **1** (Dec. 1963), 9–12.] Consider the Fibonacci number system of Ex. 34; if $n = F_{k_1} + \cdots + F_{k_r} > 0$ in that system, let $\mu(n) = F_{k_r}$. Let $\mu(0) = \infty$. We find that (A) If $n > 0$, $\mu(n - \mu(n)) > 2\mu(n)$. *Proof:* $\mu(n - \mu(n)) = F_{k_r-1} \geq F_{k_r+2} > 2F_{k_r}$ since $k_r \geq 2$. (B) If $0 < m < F_k$, $\mu(m) \leq 2(F_k - m)$. *Proof:* Let $\mu(m) = F_j$; $m \leq F_j + F_{j+2} + \cdots + F_{k-1-(k-1-j) \bmod 2} = -F_{j-1} + F_{k-(k-1-j) \bmod 2} \leq -\frac{1}{2}F_j + F_k$. (C) If $0 < m < \mu(n)$, $\mu(n - \mu(n) + m) \leq 2(\mu(n) - m)$. *Proof:* This follows from (B). (D) If $0 < m < \mu(n)$, $\mu(n - m) \leq 2m$. *Proof:* Set $m = \mu(n) - m$ in (C).

Now we will prove that if there are n chips, and if at most q may be taken in the next turn, there is a winning move iff $\mu(n) \leq q$. *Proof:* (a) If $\mu(n) > q$ all moves leave a position n', q' with $\mu(n') \leq q'$. [This follows from (D), above.] (b) If $\mu(n) \leq q$, we can either win on this move (if $q \geq n$) or we can make a move which leaves a position n', q' with $\mu(n') > q'$. [This follows from (A) above, our move is to *take* $\mu(n)$ *chips*.] It can be seen that the set of all winning moves, if $n = F_{k_1} + \cdots + F_{k_r}$, is to remove $F_{k_j} + \cdots + F_{k_r}$, for some j with $1 \leq j \leq r$, provided that $j = 1$ or $F_{k_j-1} > 2(F_{k_j} + \cdots + F_{k_r})$.

If $n = 1000$, the Fibonacci representation is $987 + 13$; the *only* lucky move to force a victory is to take 13 chips. The first player can always win unless n is a Fibonacci number.

The solution to considerably more general games of this type has been obtained by A. Schwenk (to be published).

39. $(3^n - (-2)^n)/5$.

SECTION 1.2.9

1. $1/(1-2z)+1/(1-3z)$.

2. Follows from (6), since $\binom{n}{k} = n!/k!(n-k)!$.

3. $G'(z) = \ln\big(1/(1-z)\big)/(1-z)^2 + 1/(1-z)^2$. From this and the significance of $G(z)/(1-z)$, we have $\sum_{1\le k\le n-1} H_k = nH_n - n$; this agrees with Eq. 1.2.7–8.

4. Put $t = 0$.

5. The coefficient of z^k is, by (11), (22),

$$(n-1)! \sum_{0\le j<k} \left\{ {j \atop n-1} \right\} \binom{k}{j}.$$

Now apply Eqs. 1.2.6–42 and 1.2.6–48.

A simpler method uses differentiation and 1.2.6–42.

6. $(\ln(1/(1-z)))^2$; the derivative is twice the generating function for the harmonic numbers; the sum is therefore $2H_{n-1}/n$.

8. $1/(1-z)(1-z^2)(1-z^3)\cdots$. [This is historically one of the first applications of generating functions. For an interesting account of L. Euler's eighteenth-century researches concerning this generating function, see G. Polya, *Induction and Analogy in Mathematics* (Princeton: Princeton University Press, 1954), Chapter 6.]

9. $\frac{1}{24}S_1^4 + \frac{1}{4}S_1^2S_2 + \frac{1}{8}S_2^2 + \frac{1}{3}S_1S_3 + \frac{1}{4}S_4$.

10. $G(z) = (1+x_1z)\cdots(1+x_nz)$. Taking logarithms as in the derivation of Eq. (34), we have the same formulas except (24) replaces (25), and the answer is exactly the same except $S_2, S_4, S_6, \ldots$ are replaced by $-S_2, -S_4, -S_6$, etc. We have $b_1 = S_1$, $b_2 = \frac{1}{2}S_1^2 - \frac{1}{2}S_2$, $b_3 = \frac{1}{6}S_1^3 - \frac{1}{2}S_1S_2 + \frac{1}{3}S^3$, $b_4 = \frac{1}{24}S_1^4 - \frac{1}{4}S_1^2S_2 + \frac{1}{8}S_2^2 + \frac{1}{3}S_1S_3 - \frac{1}{4}S_4$. (Cf. Ex. 9.) The numbers b_m are called the *elementary symmetric functions* of the x_j, and the formulas derived in this exercise are called *Newton's identities*.

11. We find $z^2G'(z) + zG(z) = G(z) - 1$. The solution to this differential equation is $G(z) = (-1/z)e^{-1/z}(E_1(-1/z) + C)$, where $E_1(x) = \int_x^\infty e^{-t}\,dt/t$ and C is a constant. This function is very ill-behaved in the neighborhood of $z = 0$ and $G(z)$ has no power series expansion. Indeed, since $\sqrt[n]{n!} \approx n/e$ is not bounded, the generating function does not converge in this case; it is, however, an asymptotic expansion for the above function, when $z < 0$.

12. $$\sum_{m,n\ge 0} a_{mn}w^mz^n = \sum_{m,n\ge 0}\binom{n}{m}w^mz^n = \sum_{n\ge 0}(1+w)^nz^n = 1/(1-z-wz).$$

13. $$\int_n^{n+1} e^{-st}f(t)\,dt = \frac{a_0+\cdots+a_n}{s}(e^{-sn} - e^{-s(n+1)}).$$

Adding these together, we find $\mathbf{L}f(s) = G(e^{-s})/s$.

14. Cf. Ex. 1.2.6–38.

15. $G_n(z) = G_{n-1}(z) + zG_{n-2}(z)$, so we find $H(w) = 1/(1-w-zw^2)$. Hence, ultimately, we find

$$G_n(z) = \left(\left(\frac{1+\sqrt{1+4z}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{1+4z}}{2}\right)^{n+1}\right)\Big/\sqrt{1+4z}.$$

16. $$G_{nr}(z) = (1+z+\cdots+z^r)^n = \left(\frac{1-z^{r+1}}{1-z}\right)^n.$$

The case $r = \infty$ is of interest.

17. $\sum_k \binom{-w}{k}(-z)^k = \sum_k \frac{w(w+1)\cdots(w+k-1)}{k(k-1)\cdots 1} z^k = \sum_{n,k} \begin{bmatrix} k \\ n \end{bmatrix} z^k w^n / k!.$

(Alternatively, write it as $e^{w \ln(1/(1-z))}$ and expand first by powers of w.)

18. (a) For fixed n and varying r, the generating function is

$$G_n(z) = (1+z)(1+2z)\cdots(1+nz) = z^{n+1}\left(\frac{1}{z}\right)\left(\frac{1}{z}+1\right)\left(\frac{1}{z}+2\right)\cdots\left(\frac{1}{z}+n\right)$$
$$= \sum_k \begin{bmatrix} n+1 \\ k \end{bmatrix} z^{n+1-k}$$

by Eq. (27). Hence the answer is

$$\begin{bmatrix} n+1 \\ n+1-r \end{bmatrix}.$$

(b) Similarly, the generating function is

$$\frac{1}{1-z}\cdot\frac{1}{1-2z}\cdot\cdots\cdot\frac{1}{1-nz} = \sum_k \begin{Bmatrix} k \\ n \end{Bmatrix} z^{k-n}$$

by Eq. (28), so the answer is

$$\begin{Bmatrix} n+r \\ n \end{Bmatrix}$$

19. $\sum_{n\geq 0}\left(\frac{1}{n+p/q} - \frac{1}{n+1}\right) x^{p+nq}$

$$= x^{p-q}\ln(1-x^q) - \sum_{1\leq k\leq q} \omega^{-kp}\ln(1-\omega^k x) = S_1 + S_2 + S_3,$$

where $\omega = e^{2\pi i/q}$ and where S_1, S_2, S_3 are defined below. Now

$$\lim_{x\to 1-} S_1 = \lim_{x\to 1-} x^{p-q}\ln\left(\frac{1-x^q}{1-x}\right) = \ln q;$$
$$\lim_{x\to 1-} S_2 = \lim_{x\to 1-} (x^{p-q}-1)\ln(1-x) = 0;$$

and

$$\lim_{x\to 1-} S_3 = -\sum_{0<k<q} \omega^{-kp}\ln(1-\omega^k).$$

From the identity

$$\ln(1-e^{i\theta}) = \ln\left(2e^{i(\theta-\pi)/2}\cdot\frac{e^{i\theta/2}-e^{-i\theta/2}}{2i}\right) = \ln 2 + \tfrac{1}{2}i(\theta-\pi) + \ln\sin\frac{\theta}{2},$$

we may write the latter sum as $S_4 + S_5$ where

$$S_4 = -\sum_{0<k<q} \omega^{-kp}\ln\sin\frac{k}{q}\pi = -\sum_{0<k<q/2} (\omega^{-kp}+\omega^{-(q-k)p})\ln\sin\frac{k}{q}\pi$$
$$= -2\sum_{0<k<q/2} \cos\frac{2\pi pk}{q}\ln\sin\frac{k}{q}\pi;$$

and

$$S_5 = -\sum_{0<k<q} \omega^{-kp}\left(\ln 2 - \frac{i\pi}{2} + \frac{ik\pi}{q}\right) = \ln 2 - \frac{i\pi}{2} - \frac{i\pi}{(\omega^{-p}-1)}.$$

Finally,

$$\frac{-i}{2} - \frac{i}{(\omega^{-p}-1)} = -\frac{i}{2}\left(\frac{1+\omega^p}{1-\omega^p}\right) = \frac{i}{2}\left(\frac{\omega^{p/2}+\omega^{-p/2}}{\omega^{p/2}-\omega^{-p/2}}\right) = \tfrac{1}{2}\cot\frac{p}{q}\pi.$$

SECTION 1.2.10

1. $1/n$; this is the probability that $X[n]$ is the largest.

2. $G''(1) = \sum k(k-1)p_k$, $G'(1) = \sum kp_k$.

3. min 0, ave 6.49, max 999, dev 2.42. (Note that $H_n^{(2)}$ is approximately $\pi^2/6$; see Eq. 1.2.7–7.)

4. $\binom{n}{k}p^kq^{n-k}$.

5. Mean is $36/5 = 7.2$; standard deviation is $6\sqrt{2}/5 \approx 1.697$.

7. The probability that $A = k$ is p_{mk}. For we may consider the values to be $1, 2, \ldots, m$. Given any partitioning of the n positions into m disjoint sets, there are $m!$ ways to assign the numbers $1, \ldots, m$ to these sets. Algorithm M treats these values as if only the rightmost element of each set were present; so, p_{mk} is the average for any fixed partitioning. For example, if $n = 5$, $m = 3$, one partition is

$$\{X[1], X[4]\}\,\{X[2], X[5]\}\,\{X[3]\}\,;$$

the arrangements possible are 12312, 13213, 21321, 23123, 31231, 32132. In every partition we get the same percentage of arrangements with $A = k$.

On the other hand, if more information is given the probability distribution changes. If $n = 3$, $m = 2$, the above argument considers the six possibilities 122, 212, 221, 211, 121, 112; if we know there are two 2's and one 1, then only the first three of these possibilities is to be considered. This interpretation is not consistent with the statement of the exercise, however.

8. $M(M-1)\cdots(M-n+1)/M^n = M!/(M-n)!M^n$. The larger M is, the closer this probability gets to one.

9. Let q_{nm} be the probability that exactly m distinct values occur; then from the recurrence

$$q_{nm} = \frac{M-m+1}{M}\,q_{(n-1)(m-1)} + \frac{m}{M}\,q_{(n-1)m}$$

we deduce that

$$q_{nm} = M!\left\{{n \atop m}\right\} \Big/ (M-m)!M^n.$$

See also Ex. 1.2.6–64.

10. This is $q_{nm}p_{mk}$ summed over all m, i.e.

$$\frac{1}{M^n}\sum_m \binom{M}{m}\left\{{n \atop m}\right\}\left[{m \atop k+1}\right].$$

There does not appear to be a simple formula for the average, which is

$$\left(\frac{1}{M^n}\sum_m \binom{M}{m}\left\{{n \atop m}\right\} m!H_m\right) - 1.$$

11. The first identity is obvious by writing out the power series for e^{kt}. For the second, let $u = 1 + M_1t + M_2t^2/2! + \cdots$; when $t = 0$ we have $u = 1$ and $D_t^k u = M_k$. Also, $D_u^j(\ln u) = (-1)^{j-1}(j-1)!/u^j$.

12. Since this is a product, we add the semi-invariants of each term. If $H(z) = z^n$, $H(e^t) = e^{nt}$, so we find $k_1 = n$ and all others are zero. Therefore, mean$(G_1) = n +$ mean (G), and all other semi-invariants are unchanged. (This accounts for the name "semi-invariant.")

13.
$$G_n(z) = \frac{\Gamma(n+z)}{\Gamma(z+1)n!} = \frac{1}{\Gamma(z+1)}\frac{(n+z)^z}{n+z}e^{-z}\left(1+\frac{z}{n}\right)^n\left(1+O\left(\frac{1}{n}\right)\right)$$
$$= \frac{n^{z-1}}{\Gamma(z+1)}\left(1+O\left(\frac{1}{n}\right)\right).$$

Let $z = z_n = \exp(t/\sigma_n)$. Now $z_n \to 1$, so the $\Gamma(z+1)$ term may be ignored as $n \to \infty$. The limit now is

$$\exp\left(-t\mu_n/\sigma_n + (e^{t/\sigma_n} - 1)\ln n\right)$$
$$= \exp\left(t\left(\frac{\ln n - \mu_n}{\sigma_n}\right) + \frac{t^2 \ln n}{2\sigma_n^2} + O\left(\frac{1}{\ln n}\right)\right) \to \exp\left(\frac{t^2}{2}\right).$$

14. $e^{-tpn/\sqrt{pqn}}(q + pe^{t/\sqrt{pqn}})^n = (qe^{-tp/\sqrt{pqn}} + pe^{tq/\sqrt{pqn}})^n$. Expand the exponentials in power series, get $(1 + t^2/2n + O(n^{-3/2}))^n = \exp\left(n\ln(1 + t^2/2n + O(n^{-3/2}))\right) = \exp\left(t^2/2 + O(n^{-1/2})\right) \to \exp(t^2/2)$.

15. $G_n(z)$ in Eq. (8). A generating function for probabilities may always be interpreted as the average value of a quantity, in this way.

16. (a) $\sum_{k\geq 0} e^{-\mu}(\mu z)^k/k! = e^{\mu(z-1)}$. (b) $\ln e^{\mu(e^t-1)} = \mu(e^t - 1)$, so all semi-invariants equal μ. (c) $\exp(-tnp/\sqrt{np}) \times \exp\left(np(t/\sqrt{np} + t^2/2np + O(n^{-3/2})\right) = \exp\left(t^2/2 + O(n^{-1/2})\right)$.

17. (a) The coefficients of $f(z)$, $g(z)$ are non-negative and $f(1) = g(1) = 1$. Clearly $h(z)$ shares these same characteristics since $h(1) = g(f(1))$ and the coefficients of h are polynomials in those of f, g with non-negative coefficients. (b) Let $f(z) = \sum p_k z^k$ where p_k is the probability that some event yields a "score" of k. Let $g(z) = \sum q_k z^k$ where q_k is the probability that the event described by f happens exactly k times (each occurrence of the event being independent of the others). Then $h(z) = \sum r_k z^k$, where r_k is the probability that the sum of the scores of the events that occurred is equal to k. (This is easy to see if we observe that $f(z)^k = \sum s_t z^t$, where s_t is the probability that a total score t is obtained in k independent occurrences of the event.) Example: If f gives the probabilities that a man has k male offspring, and if g gives the probabilities that there are k males in the n-th generation, then h gives the probabilities that there are k males in the $(n+1)$st generation, assuming independence. (c) mean (h) = mean (g) mean (f); var (h) = var (g) mean$^2(f)$ + mean (g) var (f).

18. Consider the choice of $X[1], \ldots, X[n]$ as a process in which we first place all the n's, then place all the $(n-1)$'s among these n's, . . . , finally place the ones among the rest. As we place the r's among the numbers $r+1, \ldots, n$, the number of local maxima from right to left increases by one if and only if we put an r at the extreme right. This happens with probability $k_r/(k_r + k_{r+1} + \cdots + k_n)$.

SECTION 1.2.11.1

1. Zero.

2. Each O symbol represents a different approximate quantity; since the lefthand side may be $f(n) - (-f(n)) = 2f(n)$, the best we can say is $O(f(n)) - O(f(n)) = O(f(n))$. To prove the latter, note that if $|x_n| \le M_1|f(n)|$ and $|y_n| \le M_2|f(n)|$, then $|x_n - y_n| \le |x_n| + |y_n| \le (M_1 + M_2)|f(n)|$. (Signed, J. H. Quick, student.)

3. $n(\ln n) + \gamma n + O(\sqrt{n} \ln n)$.

4. $|x_n| \le \max\,(x_1/f(1), \ldots, x_{n_0}/f(n_0), M)|f(n)|$.

5. (a) Take $M = |c_0|/r^m + |c_1|/r^{m-1} + \cdots + |c_m|$. (Cf. the text following Eq. (3).) (b) Disproof: Let $P(x) = 1$, $m = 1$; then $|1| > Mx$ when $x < 1/M$; the condition therefore fails for all choices of M.

6. A variable number, n, of O-symbols has been replaced by a single O-symbol, falsely implying that a single value of M will suffice for each term $|kn| \le Mn$. The given sum is actually $O(n^3)$, as we know. The last equality, $\sum_{1\le k\le n} O(n) = O(n^2)$, is perfectly valid.

7. If x is positive, the power series tells us $e^x > x^{m+1}/(m+1)!$, hence the ratio of e^x/x^m is unbounded by any M.

8. Replace n by e^n and apply the previous exercise.

9. If $|f(x)| \le M|x|^m$, $e^{f(x)} \le e^{M|x|^m} = 1 + |x|^m(M + M^2|x|^m/2! + M^3|x|^{2m}/3! + \cdots) \le 1 + |x|^m(M + M^2r^m/2! + M^3r^{2m}/3! + \cdots)$.

10. "If $f(x) = O(x^m)$, $|x| \le r$, there exists a positive number r' such that $\ln\,(1+f(x)) = O(x^m)$, $|x| \le r'$." *Proof:* Take r', r'' such that $|f(x)| \le r'' < 1$ when $|x| \le r'$. Then $|\ln\,(1+f(x))| \le |f(x)| + \frac{1}{2}|f(x)|^2 + \frac{1}{3}|f(x)|^3 + \cdots \le |x|^m M(1 + \frac{1}{2}r'' + \frac{1}{3}r''^2 + \cdots)$.

11. Apply Eq. (11) with $m = 1$, $x = \ln n/n$. This is justified since $\ln n/n$ remains bounded by some positive value r (it approaches zero for large n).

12. $\ln a + (\ln a)^2/2n + (\ln a)^3/6n^2 + O(n^{-3})$.

SECTION 1.2.11.2

1. $x = (B_0 + B_1 x + B_2 x^2/2! + \cdots)e^x - (B_0 + B_1 x + B_2 x^2/2! + \cdots)$; apply Eq. 1.2.9–11.

2. The function $B_{m+1}(\{x\})$ must be continuous, for the integration by parts.

3. $|R_{2k}| \leq \left|\dfrac{B_{2k}}{(2k)!}\right| \displaystyle\int_1^n |f^{(2k)}(x)|\, dx.$

4. $$\sum_{0\le k<n} k^m = \frac{1}{1+m}x^m + \sum_{1\le k\le m} \frac{B_k}{k!}\frac{m!}{(m-k+1)!}n^{m-k+1}$$
$$= \frac{1}{m+1}B_{m+1}(n) - \frac{1}{m+1}B_{m+1}.$$

5. It follows that

$$K = \sqrt{2}\lim_{n\to\infty}\frac{2^{2n}(n!)^2}{\sqrt{n}(2n)!};$$

$$K^2 = \lim_{n\to\infty}\frac{2}{n}\frac{n^2(n-1)^2\cdots(1)^2}{(n-\frac{1}{2})^2(n-\frac{3}{2})^2\cdots(\frac{1}{2})^2} = 4\frac{2\cdot2\cdot4\cdot4\cdot}{1\cdot3\cdot3\cdot5\cdot}\cdots = 2\pi.$$

6. Assume $c > 0$ and consider $\sum_{0\le k<n}\ln(k+c)$. We find

$$\ln\big(c(c+1)\cdots(c+n-1)\big) = (n+c)\ln(n+c) - c\ln c - n - \tfrac{1}{2}\ln c$$
$$+ \sum_{1<k\le m}\frac{B_k(-1)^k}{k(k-1)}\left(\frac{1}{(n+c)^{k-1}} - \frac{1}{c^{k-1}}\right) + R_{mn}.$$

Also

$$\ln(n-1)! = (n-\tfrac{1}{2})\ln n - n + \sigma + \sum_{1<k\le m}\frac{B_k(-1)^k}{k(k-1)}\left(\frac{1}{n^{k-1}}\right) - \frac{1}{m}\int_n^\infty \frac{B_m(\{x\})\,dx}{x^m}.$$

Now $\ln\Gamma_{n-1}(c) = c\ln(n-1) + \ln(n-1)! - \ln\big(c\cdots(c+n-1)\big)$; substituting and letting $n\to\infty$, we get

$$\ln\Gamma(c) = -c + (c-\tfrac{1}{2})\ln c + \sigma + \sum_{1<k\le m}\frac{B_k(-1)^k}{k(k-1)c^{k-1}} - \frac{1}{m}\int_0^\infty\frac{B_m(\{x\})\,dx}{(x+c)^m}.$$

This shows that $\Gamma(c+1) = ce^{\ln\Gamma(c)}$ has the same expansion we derived for $c!$.

7. $A\cdot n^{n^2/2+n/2+1/12}e^{-n^2/4}$ where A is a constant. (It is "Glaisher's constant" 1.2824271...) To obtain this result, apply Euler's summation formula to $\sum_{1\le k<n} k\ln k$. A more accurate formula is obtained if we multiply the above answer by

$$\exp\big(-B_4/2\cdot3\cdot4n^2 - \cdots - B_{2t}/(2t-2)(2t-1)(2t)n^{2t-2} + O(1/n^{2t})\big).$$

This formula makes it possible to calculate Glaisher's constant to six decimal places if we let $t = 3$, $n = 4$.

SECTION 1.2.11.3

1. Integrate by parts.

2. Substitute series for e^{-t} in integral.

3. See Eq. 1.2.9–11 and Ex. 1.2.6–48.

4. $1 + 1/u$ is bounded as a function of v, since it goes to zero as v goes from r to infinity. Replace it by M and the resulting integral is Me^{-rx}.

5. $f''(x) = f(x)\left(\frac{(n+\frac{1}{2})(n-\frac{1}{2})}{x^2} - \frac{(2n-1)}{x} + 1\right)$

has constant sign for $0 \le x \le n-1$, so $|R| \le \frac{1}{12}|f'(n-1)| + \frac{1}{12}\int_{n-1}^{n}|f''(x)|\,dx$.

6. It is $n^{n+\beta}\exp\left((n+\beta)(\alpha/n - \alpha^2/2n^2 + O(n^{-3})\right)$, etc.

7. The integrand as a power series in x^{-1} has the coefficient of x^{-n} as $O(u^{2n})$. After integration, terms in x^{-3} are $Cu^7/x^3 = O(x^{-5/4})$, etc. To get $O(x^{-2})$ in the answer, we can discard terms u^n/x^m with $4m - n \ge 9$ (cf. next exercise). Thus expanding $\exp(-u^2/2x)\exp(u^3/3x^2)\ldots$ leads ultimately to the answer

$$yx^{1/4} - \frac{y^3}{6}x^{-1/4} + \frac{y^5}{40}x^{-3/4} + \frac{y^4}{12}x^{-1} - \frac{y^7}{336}x^{-5/4} - \frac{y^6}{36}x^{-3/2} + \left(\frac{y^9}{3456} - \frac{y^5}{20}\right)x^{-7/4} + O(x^{-2}).$$

8. The integrand can be expanded into terms of the form $c_{mn}u^m/x^n$. These terms integrate into $O\big((m+1)r - n\big)$. [Note that if $r > \frac{1}{2}$ the series for $e^{-u^2/2x}$ integrates into series that diverge for large x, hence the assumption $r > \frac{1}{2}$ is necessary.] Multiplying two terms $(u^{m_1}/x^{n_1})(u^{m_2}/x^{n_2})$, if $(m_1+1)r - n_1 \le -s$, we have

$$\begin{aligned}(m_1+m_2+1)r - (n_1+n_2) &= (m_1+1)r - n_1 + (m_2+1)r - n_2 - r \\ &\le -s + r - r = -s.\end{aligned}$$

Therefore we may expand $\exp(-u^2/2x)\exp(u^3/3x^2)\ldots$ and discard all terms with $(m+1)r - n \le -s$ *before* multiplying together the factors; and all terms $\exp(u^p/px^{p-1})$ with $(p+1)r - p + 1 \le -s$ may be suppressed, i.e. those with $p > \lceil (s+2r)/(1-r)\rceil$.

9. We may assume $p \ne 1$, since $p = 1$ is given by Theorem A. We also assume $p \ne 0$ since that case is trivial.

Case 1. $p < 1$. Substitute $t = px(1-u)$ and then $v = -\ln(1-u) - pu$. We have $dv = \big((1-p+pu)/(1-u)\big)\,du$, so the transformation is monotone for $0 \le u \le 1$, and we obtain an integral of the form

$$\int_0^\infty xe^{-xv}\,dv\left(\frac{1-u}{1-p+pu}\right).$$

The parenthesized quantity is

$$\frac{1}{1-p}\left(1 - \frac{v}{(1-p)^2} + \cdots\right).$$

The answer is therefore

$$\frac{p}{1-p}(pe^{1-p})^x\frac{e^{-x}x^x}{\Gamma(x+1)}\left(1 - 1/(p-1)^2x + O(x^{-2})\right).$$

Case 2. $p > 1$. This is $1 - \int_{px}^\infty(\ )$. In the latter integral, substitute $t = px(1+u)$, then $v = pu - \ln(1+u)$, and proceed as in Case 1. The answer turns out to be the same formula as Case 1, plus one. Note that $pe^{1-p} < 1$ so $(pe^{1-p})^x$ is *very* small.

10. $\dfrac{p}{p-1}(pe^{1-p})^x e^{-x} x^x \left(1 - e^{-y} - \dfrac{e^{-y}(e^y - 1 - y - y^2/2)}{x(p-1)^2} + O(x^{-2})\right).$

11. First, $xQ_x(n) + R_{1/x}(n) = n!(x/n)^n e^{n/x}$. (The case $x = -1$ is interesting here!) We get

$$Q_x(n) = \frac{1}{1-x} - \frac{x}{(1-x)^3 n} + O(n^{-2}),$$

$$R_x(n) = \frac{1}{1-x} - \frac{x}{(1-x)^3 n} + O(n^{-2}), \quad \text{if } x < 1;$$

$$Q_x(n) = \frac{1}{x} n! \left(\frac{x}{n}\right)^n e^{nx} - \frac{1}{x-1} + \frac{1}{(x-1)^3 n} + O(n^{-2}),$$

$$R_x(n) = n! \left(\frac{x}{n}\right)^n e^{nx} - \frac{1}{x-1} + \frac{x}{(x-1)^3 n} + O(n^{-2}), \quad \text{if } x > 1.$$

These formulas are quite easily verified when $|x| < 1$ and the relation between $Q_x(n)$ and $R_{1/x}(n)$ extends this to $|x| > 1$. The case $x = -1$ remains; this requires more delicate maneuverings with limits. For further details about the asymptotic expansion, and its connection with Stirling numbers of the second kind, see L. Carlitz, *Proc. Amer. Math. Soc.* **16** (1965), 248–252.

12. $\dfrac{1}{\sqrt{2}} \gamma(\frac{1}{2}, \frac{1}{2}x^2).$

15. Expanding the integrand by the binomial theorem, we find the integral is $1 + Q(n)$.

16. Write $Q(k)$ as a sum, and interchange order of summation using Eq. 1.2.6–49.

SECTION 1.3.1

1. Four; each byte would then contain $3^4 = 81$ different values.

2. Five, since five bytes is always adequate but four is not.

3. (0:2); (3:3); (4:4); (5:5).

4. Presumably index register 4 contains a value greater than or equal to 2000, so that after indexing a valid memory address results.

5. "`DIV -80,3(0:5)`" or simply "`DIV -80,3`".

6. (a) Sets rA to | - | 5 | 1 | 200 | 15 |. (b) Sets rI2 to −200. (c) Sets rX to | + | 0 | 0 | 5 | 1 | ? |. (d) Undefined, since we can't load such a big value into an index register. (e) Sets rX to | - | 0 | 0 | 0 | 0 | 0 |.

7. Let the magnitude |rAX| before the operation be n, and let the magnitude of V be d. After the operation the magnitude of rA is $\lfloor n/d \rfloor$, and the magnitude of rX is $n \bmod d$. The sign of rX afterwards is the previous sign of rA; the sign of rA afterwards is + if the previous signs of rA and V were the same, and it is − otherwise.

8. rA ← | + | 0 | 617 | 0 | 1 | ; rX ← | − | 0 | 0 | 0 | 1 | 0 |.

9. ADD, SUB, DIV, NUM, JOV, JNOV, INCA, DECA, INCX, DECX.

10. CMPA, CMP1, CMP2, CMP3, CMP4, CMP5, CMP6, CMPX. (Also FCMP, floating point.)

11. MOVE, LD1, LD1N, INC1, DEC1, ENT1, ENN1.

12. INC3 0,3.

13. "JOV 1000" makes no difference except time. "JNOV 1001" makes a different setting of rJ in most cases. "JNOV 1000" makes an extraordinary difference, since it may lock the computer in an infinite loop.

14. NOP with anything; ADD, SUB with F = (0:0); HLT (depending on how you interpret the statement of the exercise); any shift with address and index zero; MOVE with F = 0; JSJ with address equal to the location of the instruction plus one; any of the INC or DEC instructions with address and index zero; ENTi 0,i for $1 \le i \le 6$; SLC or SRC with address a multiple of 10.

15. 70; 80; 120. (record size times 5)

16. (a) STZ 0; ENT1 1; MOVE 0(49); MOVE 0(50). If the byte size were known to equal 100, only one MOVE instruction would have been necessary, but we are not allowed to make assumptions about the byte size. (b) Use 100 STZ's.

17. (a)

```
STZ    0,2
DEC2   1
J2NN   3000
```

(b)

```
        STZ    0
        ENT1   1
        JMP    3004
(3003)  MOVE   0(63)
(3004)  DEC2   63
        J2NN   3003
        INC2   63
        ST2    3008(4:4)
(3008)  MOVE   0
```

(Using assembly language, a slightly faster program which uses "bytesize minus 1" instead of 63 could be written; see the following section.)

18. (If you have correctly followed the instructions, an overflow will occur on the ADD, with minus zero in register A afterward.) Answer: Overflow is set on, comparison is set EQUAL, rA is set to | − | 30 | 30 | 30 | 30 | 30 |, rX is set to | − | 31 | 30 | 30 | 30 | 30 |, rI1 is set to +3, and memory locations 0001, 0002 are set to zero.

19. $24u = (2 + 1 + 2 + 2 + 1 + 1 + 1 + 2 + 2 + 1 + 2 + 2 + 3 + 1 + 1)u$.

20.

```
(3990)  ENT1   0
        MOVE   3995      (standard F for MOVE is 1)
        MOVE   0(49)
(3993)  MOVE   0(50)
        JMP    3993
(3995)  HLT    0
```

21. (a) Not unless it can be set to zero by external means (see the "GO-button", Ex. 26), since a program can set rJ = N only by jumping from location $N - 1$.

(b)

```
          LDA  -1,4
          LDX  3004
          STX  -1,4
          JMP  -1,4
(3004)    JMP  3005
(3005)    STA  -1,4
```

22. *Minimum time:* If b is the byte size, the assumption that $|X^{13}| < b^5$ implies that $X^2 < b$, so X^2 can be contained in one byte. The following ingenious solution due to G. MacGinitie makes use of this fact.

```
(3000)  LDA   2000
        STA   3500(0:1)
        MUL   2000
        STX   3501(0:1)
        SRC   1
        MUL   3501
        STA   3502
        MUL   3500
        MUL   3502
        MUL   3502
        SLAX  2
        HLT   0
(3500)  NOP   0
(3501)  NOP   0
(3502)  NOP   0
```

rA					rX				
X^2	0	0	0	0	0	0	0	0	0
X^4		0	0	0	0	0	0	0	0
X^4		0	0	0	0	0	0	0	0
X^5			0	0	0	0	0	0	0
X^9					0	0	0	0	0
0	0	X^{13}					0	0	0
X^{13}					0	0	0	0	0

space = 15; time = $62u$.

At least five multiplications are "necessary" (see Chapter 4); however, see the much better solution on the following page.

Minimum space:

```
(3000)  ENT4  12
        LDA   2000
(3002)  MUL   2000
        SLAX  5
        DEC4  1
        J4P   3002
        HLT   0
```

space = 7; time = $171u$.

True minimum time: As R. W. Floyd points out, the conditions imply $|X| \leq 6$, so the minimum execution time is achieved by referring to a table:

```
(3000)  LD1   2000
        LDA   3500,1
        HLT   0
(3494)  (−6)^13
(3495)  (−5)^13
  ⋮
(3506)  (+6)^13
```

space = 16, time = $4u$.

23. The following solution by R. D. Dixon appears to satisfy all the conditions:

```
(3000)  ENT1  4
(3001)  LDA   200
        SRA   0,1
        SRAX  1
        DEC1  1
        J1NN  3001
        SLAX  5
        HLT   0
```

24. (a) DIV 3500, where 3500 = | + | 1 | 0 | 0 | 0 | 0 |.
(b) SRC 4; SRA 1; SLC 5.

25. Some ideas: (a) Obvious things like faster memory, more input-output devices. (b) The I field could be used for J-register indexing, and/or multiple indexing (specify two different index registers), and/or "indirect addressing" (see Exs. 2.2.2–3, 4, 5). (c) Index registers and J register can be extended to a full five bytes; this means locations with higher addresses can be referred to only by indexing, but this is not so intolerable if multiple indexing is available as in (b). (d) An interrupt capability can be added (when certain conditions occur, all registers are stored in special locations—say locations $-1, -2, \ldots$—and a jump is made to a control program which later restarts the original program by using a new operation code, see Ex. 1.4.4–18). (e) A "real time clock" could be added, in a negative memory address. (f) "Logical operations" could be added to binary versions of MIX (see Ex. 2.5–28 and Chapter 7). (g) An "execute" command, meaning to perform the instruction at location M, could be another variant of C = 5.

26. The following routine is the shortest found so far for which the transfer card does not depend on the byte size. It is tempting to use a (2:5) field to get at cols. 7–10 of the card, but this cannot be done since $2 \cdot 8 + 5 = 21$. Since this routine requires only 28 instructions, it can be adapted for paper tape.

To make the program easier to follow, it is presented here in symbolic language, anticipating the following section of the text.

The transfer card has the format TRANS0nnnn in columns 1–10, where nnnn is the place to jump to start execution.

```
                                                        characters
      BUFF   EQU    28          Buffer area is 28–43    punched on card:
             ORIG   0
00    TEMP1  IN     16(16)      Read in second card.        0 06
01    READ   IN     BUFF(16)    Read next card.             Y 06
02           LD1    0(0:0)      [ENT1 0]                      I
03           ENTA   0                                        B=
04           JBUS   *(16)       Wait for read to finish     D 04
05           LD2N   BUFF+1(1:1) — (count + 30)              Z IQ
06           STZ    BUFF+1(1:1) Clear (1:1) so (2:5)       Z I3
07           LDX    BUFF+1        can be used.             Z EN
08    TEMP   NUM    0                                        E
09           STA    TEMP1       Starting location          EU
10           ENTA   30,2        — count                    0BB=
11    LOOP   STA    TEMP(1:1)                              H IU
12           LD3    TEMP1                                    EJ
13           JAZ    0,3         Transfer card               CA.
14           ENTA   1,3         Increase TEMP1.            ACB=
15           STA    TEMP1                                    EU
```

```
16           LDA    BUFF+3,1(5:5)                          1A–H
17           DECA   25                                     V A=
18           STA    0,3         Store sign.                 CEU
19           LDA    BUFF+2,1                               0AEH
20           LDX    BUFF+3,1                               1AEN
21           NUM                                             E
22           STA    0,3(1:5)    Store magnitude.            CLU
23           MOVE   0,1(2)      [INC1 2!]                   ABG
24           LDA    TEMP(1:1)                              H IH
25           DECA   1           Decrease count.            A A=
26           JAP    LOOP                                   J B.
27           JMP    READ        Ready for new card         A  9
```

SECTION 1.3.2

1. ENTX 1000; STX X.

2. The STJ instruction in line 03 resets this address. (It is conventional to denote the address of such instructions by "*", both because it is simple to write, and because it provides a recognizable test of an error condition in a program, where a subroutine has not been entered properly because of some oversight.)

3. Read in 100 words from tape unit zero; exchange the maximum of these with the last one; exchange the maximum of the remaining 99 with the last of those; etc. Eventually the 100 words will become completely sorted into ascending sequence and the result is then written onto tape unit one.

4. Nonzero locations:

3000:	+	0000	00	18	35
3001:	+	2051	00	05	09
3002:	+	2050	00	05	10
3003:	+	0001	00	00	49
3004:	+	0499	01	05	26
3005:	+	3016	00	01	41
3006:	+	0002	00	00	50
3007:	+	0002	00	02	51
3008:	+	0000	00	02	48
3009:	+	0000	02	02	55
3010:	−	0001	03	05	04
3011:	+	3006	00	01	47
3012:	−	0001	03	05	56
3013:	+	0001	00	00	51
3014:	+	3008	00	06	39
3015:	+	3003	00	00	39
3016:	+	1995	00	18	37
3017:	+	2035	00	02	52
3018:	−	0050	00	02	53
3019:	+	0501	00	00	53
3020:	−	0001	05	05	08

3021:	+	0000		00	01	05
3022:	+	0000		04	12	31
3023:	+	0001		00	01	52
3024:	+	0050		00	01	53
3025:	+	3020		00	02	45
3026:	+	0000		04	18	37
3027:	+	0024		04	05	12
3028:	+	3019		00	00	45
3029:	+	0000		00	02	05
0000:	+					2
1995:	+	06	09	19	22	23
1996:	+	00	06	09	25	05
1997:	+	00	08	24	15	04
1998:	+	19	05	04	00	17
1999:	+	19	09	14	05	22
2024:	+					2035
2049:	+					2010
2050:	+					3
2051:	−					499

(the latter two may be interchanged, with corresponding changes to 3001 and 3002)

6. (a) If n is not prime, by definition n has a divisor d with $1 < d < n$. If $d > \sqrt{n}$, n/d is a divisor with $1 < n/d < \sqrt{n}$. (b) If N is not prime, N has a *prime* divisor d with $1 < d \le \sqrt{n}$. The algorithm has verified that N has no prime divisors $\le p =$ PRIME[K]; also $\mathtt{N} = p\mathtt{Q} + \mathtt{R} < p\mathtt{Q} + p \le p^2 + p < (p+1)^2$. Any prime divisor of N is therefore $> p + 1 > \sqrt{\mathtt{N}}$.

Strictly speaking, we must also prove that there will be a sufficiently large prime less than N when N is prime, so that when $\mathtt{K} \leftarrow \mathtt{K} + 1$ we know PRIME[K] has already been stored. This follows from "Bertrand's postulate," i.e., if p is prime there is a larger prime less than $2p$.

7. (a) It refers to the location of line 29. (b) The program would then fail; line 14 would refer to line 15 instead of line 25; line 24 would refer to line 15 instead of line 12.

8. Prints 100 lines. If the 12000 characters on these lines were arranged end to end, they would reach quite far and would consist of five blanks followed by five A's

followed by ten blanks followed by five A's followed by fifteen blanks . . . followed by $5k$ blanks followed by five A's followed by $5(k+1)$ blanks . . . until 12000 characters have been printed. The second-last line ends with AAAAA and 35 blanks; the final line is all blank. The total effect is one of OP art, as in OP-code.

9. In the table, the (4:4) field is the maximum F setting; (1:2) is the location of a checking routine.

```
B      EQU   1(4:4)
BMAX   EQU   B-1
TABLE  NOP   GOOD(BMAX)
       ADD   FLOAT(5:5)
       SUB   FLOAT(5:5)
       MUL   FLOAT(5:5)
       DIV   FLOAT(5:5)
       HLT   GOOD
       SRC   GOOD
       MOVE  MEMORY(BMAX)
       LDA   FIELD(5:5)
       LD1   FIELD(5:5)
...
       STZ   FIELD(5:5)
       JBUS  MEMORY(19)
       IOC   GOOD(19)
       IN    MEMORY(19)
       OUT   MEMORY(19)
       JRED  MEMORY(19)
       JLE   MEMORY
       JANP  MEMORY
...
       JXNP  MEMORY
       ENNA  GOOD
...
       ENNX  GOOD
       CMPA  FLOAT(5:5)
       CMP1  FIELD(5:5)
...
       CMPX  FIELD(5:5)
```

```
BEGIN  LDA   INST
       CMPA  VALID(3:3)
       JG    BAD                 I field > 6?
       LD1   INST(5:5)
       DEC1  64                  C field > 64?
       J1NN  BAD
       CMPA  TABLE+64,1(4:4)     F field > F max?
       JG    BAD
       LD1   TABLE+64,1(1:2)     Jump to special
       JMP   0,1                   routine.
FLOAT  CMPA  VALID(4:4)          F = 6 allowed on
       JE    GOOD                  arithmetic op
FIELD  ENTA  0
       LDX   INST(4:4)           This is a tricky
       DIV   =9=                   way to check
       STX   *+1(0:2)              for a valid
       INCA  0                     partial field.
       CMPA  =5=
       JG    BAD
MEMORY LDX   INST(3:3)
       JXNZ  GOOD                If I = 0,
       LDX   INST(0:2)             ensure the
       JXN   BAD                   address is a
       CMPX  =3999=                valid memory
       JLE   GOOD                  location.
       JMP   BAD
VALID  CMPX  3999,6(6)
```

10. The catch to this problem is that there may be several places in a row (column) where the minimum (maximum) occurs, and each is a potential saddle point.

Solution 1: In this solution we make a list of all columns in which the row minimum occurs, then check for a column maximum for each column in the list.

rX ≡ current max or min; rI1 traces through the matrix (goes from 72 down to zero unless a saddle point is found); rI2 ≡ column index of rI1; rI3 ≡ size of list of minima. Notice that in all cases the terminating condition for a loop is that an index register is ≤ 0.

```
* SOLUTION 1
A10     EQU   1008
LIST    EQU   1000
START   ENT1  72          Begin at lower right column.
ROWMIN  ENT2  8           Now rI1 is at 8th column of row.
2H      LDX   A10,1       Candidate for row minimum
        ENT3  0           List empty
```

```
4H      INC3  1
        ST2   LIST,3     Put column index in list.
1H      DEC1  1          Go left one.
        DEC2  1
        J2Z   COLMAX     Done with row?
3H      CMPX  A10,1
        JL    1B         Is rX still minimum?
        JG    2B         New minimum?
        JMP   4B         Another appearance of minimum.
COLMAX  LD2   LIST,3     Get column from list.
        INC2  64
1H      CMPX  A10,2
        JL    NO         Is row min < col element?
        DEC2  8
        J2P   1B         Done with column?
YES     INC1  A10+8,2    Yes; rI1 ← address of saddle.
        HLT
NO      DEC3  1          Is list empty?
        J3P   COLMAX     No; try again.
        J1P   ROWMIN     Have all rows been tried?
        HLT              Yes; rI1 = 0, no saddle. ▮
```

Solution 2: The introduction of Mathematics gives a different algorithm.

Theorem. *Let* $R(i) = \min_j a_{ij}$, $C(j) = \max_i a_{ij}$. *The element* $a_{i_0 j_0}$ *is a saddle point if and only if* $R(i_0) = \max_i R(i) = C(j_0) = \min_j C(j)$.

Proof: If $a_{i_0 j_0}$ is a saddle point, then for any fixed i, $R(i_0) = C(j_0) \geq a_{i j_0} \geq R(i)$; so $R(i_0) = \max_i R(i)$. Similarly $C(j_0) = \min_j C(j)$. Conversely, assuming the given condition, $R(i_0) \leq a_{i_0 j_0} \leq C(j_0) = R(i_0)$ implies $a_{i_0 j_0} = R(i_0)$ so we have a saddle point. ▮

(It may be of interest to note that we always have the inequality

$$\max_i \min_j a_{ij} = \min_j a_{i_0 j} \leq \min_j \max_i a_{ij}, \text{ for some } i_0;$$

so there is no saddle point iff $\max R(i) < \min C(j)$, i.e. all the R's are less than all the C's.)

A program based on this theorem finds the smallest column maximum and then searches for an equal row minimum. (Phase 1: rI1 is col index; rI2 runs through matrix. Phase 2: rI1 is possible answer; rI2 runs through matrix; rI3 is row index; rI4 is column index.)

```
* SOLUTION 2
A10     EQU   1008
CMAX    EQU   1000
PHASE1  ENT1  8          Start at column 8.
3H      ENT2  64,1
        JMP   2F
1H      CMPX  A10,2      rX still minimum?
        JGE   *+2
```

```
2H      LDX    A10,2       New maximum in column
        DEC2   8
        J2P    1B
        STX    CMAX+8,2    Store column maximum.
        J2Z    1F          First time?
        CMPA   A10,2       rA still min max?
        JLE    *+2
1H      LDA    A10,2
        DEC1   1           Move left a column.
PHASE2  ENT3   64          At this point rA = min C(j)
3H      ENT2   8,3         Prepare to search a row.
        ENT4   8
1H      CMPA   A10,2       Is a[i,j] ≥ min C(j)?
        JG     NO          No saddle in this row
        JL     2F
        CMPA   CMAX,4      a[i,j] = C(j)?
        JNE    2F
        ENT1   A10,2       Possible saddle point
2H      DEC4   1           Move left in row.
        DEC2   1
        J4P    1B
        HLT                Saddle point found
NO      DEC3   8
        ENT1   0           Try another row.
        J3P    3B
        HLT                rI1 = 0; no saddle. ▌
```

We leave it to the reader to invent a still better solution in which "phase 1" records all possible rows which are candidates for the row search in "phase 2". It is not necessary to search all rows, just those i_0 for which there exists j_0 with $a_{i_0 j_0} = C(j_0) = \min_j C(j)$. Usually this is only one row.

In some trial runs with elements selected at random from $\{0,1,2,3,4\}$, solution 1 required approximately $730u$ to run, while solution 2 took about $540u$. Given a matrix of all zeroes, solution 1 found a saddle point in $137u$, solution 2 in $524u$.

11. Assume an $m \times n$ matrix. (a) By the theorem in the answer to Ex. 10, all saddle points of a matrix have the same value, so (under our assumption of distinct elements) there is at most one saddle point. By symmetry the desired probability is mn times the probability that a_{11} is a saddle point. This latter is $1/(mn)!$ times the number of permutations with $a_{12} > a_{11}, \ldots, a_{1n} > a_{11}, a_{11} > a_{21}, \ldots, a_{11} > a_{m1}$; this is $1/(m+n-1)!$ times the number of permutations of $m+n-1$ elements in which the first is greater than the next $(m-1)$ and less than the remaining $(n-1)$, namely $(m-1)!(n-1)!$. The answer is therefore

$$mn(m-1)!(n-1)!/(m+n-1)! = (m+n) \Big/ \binom{m+n}{n}.$$

In our case this is $17/\binom{17}{8}$, only one chance in 1430. (b) Under the second assumption, an entirely different method must be used since there can be multiple saddle points; in fact either a whole row or whole column must consist entirely of saddle points. The

probability equals the probability that there is a saddle point with value zero plus the probability that there is a saddle point with value one. The former is the probability that there is at least one column of zeroes; the latter is the probability that there is at least one row of ones. The answer is $(1-(1-2^{-m})^n)+(1-(1-2^{-n})^m)$; in our case, 9247447962340362 31/18446744073709551616, about 1 in 19.9. An approximate answer is $n2^{-m}+m2^{-n}$.

13.

```
*CRYPTANALYST PROBLEM (CLASSIFIED)
UNIT   EQU   19            Input unit number
SIZE   EQU   14            Input record size
TABLE  EQU   1000          Table of counts
       ORIG  TABLE           (initially zero
       CON   -1              except entries for
       ORIG  TABLE+46        blank space and
       CON   -1              asterisk)
       ORIG  2000
BUF1   ORIG  *+SIZE        First buffer area
       CON   -1            "Sentinel" at end of buffer
       CON   *+1           Each buffer refers to other
BUF2   ORIG  *+SIZE        Second buffer
       CON   -1            "Sentinel"
       CON   BUF1          Reference to first buffer
BEGIN  IN    BUF1(UNIT)    Input first record.
       ENT6  BUF2
1H     IN    0,6(UNIT)     Input next record.
       LD6   SIZE+1,6      During this input, prepare
       ENT5  0,6             to process previous one.
2H     LDX   0,5           Five chars → rX.
       JXN   1B            End of record?
1H     SLAX  1
       STA   *+1(2:2)      Next char → rI1.        main
       ENT1  0                                     loop,
       LDA   TABLE,1                               should
       JAN   2F            Is character special?   run as
       INCA  1             Update table entry.     fast as
       STA   TABLE,1                               possible
1H     JXP   1B            Any non-blanks in rX?
       INC5  1
       JMP   2B
2H     J1Z   1B            Skip over a blank.
       ENT1  1             Asterisk detected.
2H     LDA   TABLE,1
       JANP  1F            Skip zero answers.
       CHAR                Convert to decimal.
       JBUS  *(19)         Wait for typewriter complete.
       ST1   CHAR(1:1)
       STA   CHAR(4:5)
```

```
        STX   FREQ
        OUT   ANS(19)      Type one answer.
1H      CMP1  =60=
        INC1  1            Up to 60 character
        JL    2B             codes counted
        HLT
ANS     ALF                Output buffer
        ALF
CHAR    ALF   C  NN
FREQ    ALF   NNNNN
        ORIG  ANS+14       Rest of buffer is blank
        END   BEGIN        Literal constant =60= here. ▮
```

For this problem, buffering of *output* is not desirable since it could save at most $7u$ of time per line output, and this is quite insignificant compared to the time required to output the line itself. For information about letter frequencies, see Charles P. Bourne and Donald F. Ford, "A study of the statistics of letters in English words," *Information and Control* 4 (1961), 48–67.

14. To make the problem more challenging, the following solution uses as much *trickery* as possible, in order to reduce execution time. Can the reader squeeze out any more microseconds?

```
*DATE OF EASTER
EASTER  STJ   EASTX
        STX   Y
        ENTA  0            E1.
        DIV   =19=
        INCX  1
        STX   G(0:2)
        LDA   Y            E2.
        MUL   =1//100+1=   (see
        INCA  1              below)
        STA   C(1:4)
        MUL   =3//4+1=     E3.
        STA   XPLUS12(0:2)
        LDA   =8(1:1)=
        MUL   C            rA = 8C
        INCA  680          680 = 5 + 27 · 25
        MUL   =1//25+1=    rA = Z + 32
XPLUS12 DECA  0
        STA   1F(0:2)      Z + 20 − X
        LDA   Y            E4.
        MUL   =1//4+1=
        ADD   Y
        SUB   XPLUS12(0:2)
        INCA  5
        STA   DPLUS3
G       ENTA  0            E5.
```

```
        MUL   =11=
1H      INCX  0
        DIV   =30=
        JXNN  *+2            see Ex. 15
        INCX  30
        CMPX  =24=
        JE    1F
        CMPX  =25=
        JNE   2F
        LDA   G(0:2)
        DECA  11
        JANP  2F
1H      INCX  1
2H      DECX  20             E6. (24 — N)
        CMPX  =3=
        JLE   *+2
        DECX  30
        STX   N(0:2)
        LDAN  N(0:2)         E7.
        ADD   DPLUS3
        SRAX  5
        DIV   =7=
        SLAX  5
N       INCA  0              31 — N
        JANN  1F             E8.
        CHAR
        LDA   APRIL
        JMP   2F
1H      DECA  31
        CHAR
        LDA   MARCH
2H      JBUS  *(18)
        STA   MONTH
        STX   DAY(1:2)
        LDA   Y
        CHAR
        STX   YEAR
        OUT   ANS(18)        Print
EASTX   JMP   *
MARCH   ALF   MARCH
APRIL   ALF   APRIL
ANS     ALF
DAY     ALF   DD
MONTH   ALF   MMMMM
        ALF   ,
YEAR    ALF   YYYYY
C       CON   0              C times byte size
        ORIG  *+20
```

```
BEGIN    LDX    =1950=          "driver"
         JMP    EASTER            routine,
         LDX    Y                 uses the
         INCX   1                 above
         CMPX   =2000=            subroutine
         JLE    EASTER+1
         HLT
         END    BEGIN           ∎
```

A rigorous justification for the change from division to multiplication in several places can be based on the fact that the number in rA is not too large. (Cf. Section 12.4.1.) The program works with all byte sizes.

16. Work with scaled numbers, $R_n = 10^n r_n$. $R_n(1/m) = R$ iff $10^n/(R + \frac{1}{2}) < m \le 10^n/(R - \frac{1}{2})$; thus we find $m_h = \lfloor 2 \cdot 10^n/(2R - 1) \rfloor$.

```
*SUM OF HARMONIC SERIES
BUF     ORIG   *+24
START   ENT2   0
        ENT1   3          [5 — n]
        ENTA   20
OUTER   MUL    =10=
        STX    CONST      [2 · 10^n]
        DIV    =2=
        ENTX   2
        JMP    1F
INNER   STA    R
        ADD    R
        DECA   1
        STA    TEMP       [2R — 1]
        LDX    CONST
        ENTA   0
        DIV    TEMP
        INCA   1
        STA    TEMP       [m_h + 1]
        SUB    M
        MUL    R
        SLAX   5
        ADD    S
        LDX    TEMP
1H      STA    S          Partial sum
        STX    M
        LDA    M
        ADD    M
        STA    TEMP
        LDA    CONST
        ADD    M          Compute
        SRAX   5          ⌊(2 · 10^n + m)/2m⌋.
        DIV    TEMP
```

```
        JAP   INNER     R > 0?
        LDA   S         Answer
        CHAR
        SLAX  0,1       Neat formatting
        SLA   1
        INCA  40        Decimal point
        STA   BUF,2
        STX   BUF+1,2
        INC2  3
        DEC1  1
        LDA   CONST
        J1NN  OUTER
        OUT   BUF(18)
        HLT
        END   START     ▮
```

The output is

```
0006.16        0008.449        0010.7509        0013.05362
```

in 65595u plus output time.

18.

```
FAREY   STJ   9F      Assume rI1 contains n
        ENTA  0       and assume n > 1.
        STA   X       x0 ← 0.
        ENTX  1
        STX   Y       y0 ← 1.
        STX   X+1     x1 ← 1.
        ST1   Y+1     y1 ← n.
        ENT2  2       k ← 2.
1H      LDX   Y-2,2
        INCX  0,1
        ENTA  0
        DIV   Y-1,2
        STA   TEMP    ⌊(y_{k-2} + n)/y_{k-1}⌋
        MUL   Y-1,2
        SLAX  5
        SUB   Y-2,2
        STA   Y,2     y_k
        LDA   TEMP
        MUL   X-1,2
        SLAX  5
        SUB   X-2,2
        STA   X,2     x_k
        CMPA  Y,2     Test if x_k < y_k.
        INC2  1       k ← k + 1.
        JL    1B      If so, continue.
9H      JMP   *       Exit from subroutine. ▮
```

19. (a) Induction. (b) Let $k \geq 0$ and let the righthand side of (*) be denoted by X, Y. By part (a) we have gcd $(X, Y) = 1$ and $X/Y > x_{k+1}/y_{k+1}$. So if

$$X/Y \neq x_{k+2}/y_{k+2},$$

we have, by definition, $X/Y > x_{k+2}/y_{k+2}$. But this implies

$$\begin{aligned}\frac{1}{Yy_{k+1}} &= \frac{Xy_{k+1} - Yx_{k+1}}{Yy_{k+1}} = \frac{X}{Y} - \frac{x_{k+1}}{y_{k+1}} \\ &= \left(\frac{X}{Y} - \frac{x_{k+2}}{y_{k+2}}\right) + \left(\frac{x_{k+2}}{y_{k+2}} - \frac{x_{k+1}}{y_{k+1}}\right) \geq \frac{1}{Yy_{k+2}} + \frac{1}{y_{k+1}y_{k+2}} \\ &= \frac{Y + y_{k+1}}{Yy_{k+1}y_{k+2}} > \frac{n}{Yy_{k+1}y_{k+2}} \geq \frac{1}{Yy_{k+1}}.\end{aligned}$$

For more of the interesting properties of the Farey series, and its history, see G. H. Hardy and E. M. Wright, *The Theory of Numbers*, Oxford, chapter 3.

20.

```
*TRAFFIC SIGNAL PROBLEM
BSIZE    EQU   1(4:4)          Bytesize
2BSIZE   EQU   2(4:4)          Twice bytesize
DELAY    STJ   1F               If rA contains n,
         DECA  6                  this subroutine
         DECA  2                  waits max (n, 7)u
         JAP   *-1                exactly, not including
         JAN   *+2                the jump
         NOP                      to the subroutine
1H       JMP   *
FLASH    STJ   2F          4    This subroutine
         ENT2  8           5      flashes the
         LDA   =49991=     7      appropriate DON'T
1H       JMP   DELAY       8      WALK light
         DECX  0,1         9    Turn light off.
         LDA   =49996=     2
         JMP   DELAY       3
         INCX  0,1         4    "DON'T WALK"
         DEC2  1           1
         J2Z   1F          2    Repeat eight times.
         LDA   =49993=     4
         JMP   1B          5
1H       LDA   =399992=         Set amber 2u after exit.
         JMP   DELAY       5
2H       JMP   *           6
WAIT     JNOV  *                Del Mar green until tripped
TRIP     INCX  BSIZE            DON'T WALK on Del Mar
         ENT1  2BSIZE
         JMP   FLASH
         LDX   BAMBER           Amber on boulevard
         LDA   =799995=
         JMP   DELAY       3    Wait 8 seconds
         LDX   AGREEN      5    Green for avenue
         LDA   =799996=
         JMP   DELAY            Wait 8 seconds.
         INCX  1                DON'T WALK on Berk'ly
```

```
          ENT1  2
          JMP   FLASH        Do flash cycle.
          LDX   AAMBER       Amber on avenue
          JOV   *+1          Cancel redundant trip.
          LDA   =499994=
          JMP   DELAY        Wait 5 seconds.
BEGIN     LDX   BGREEN       Green on boulevard
          LDA   =1799994=
          JMP   DELAY        Wait at least 18
          JMP   WAIT           seconds.
AGREEN    ALF   CABA         Green for avenue
AAMBER    ALF   CBBB         Amber for avenue
BGREEN    ALF   ACAB         Green for boulevard
BAMBER    ALF   BCBB         Amber for boulevard
          END   BEGIN        ▮
```

22.

```
*JOSEPHUS PROBLEM
N    EQU   24
M    EQU   11
X    ORIG  *+N
OH   ENT1  N-1          1          Set each cell to
     STZ   X+N-1        1            number of next man
     ST1   X-1,1      N - 1          in the sequence.
     DEC1  1          N - 1
     J1P   *-2        N - 1
     ENT1  0            1
     ENTA  1            1
1H   ENT2  M-2        N - 1         (assume M > 2)
     LD1   X,1      (M - 2)(N - 1)  Count around
     DEC2  1        (M - 2)(N - 1)    the circle.
     J2P   *-2      (M - 2)(N - 1)
     LD2   X,1        N - 1         rI1 ≡ lucky man
     LD3   X,2        N - 1         rI2 ≡ doomed man
     CHAR             N - 1         rI3 ≡ next man
     STX   X,2(4:5)   N - 1         Store execution number.
     NUM              N - 1
     INCA  1          N - 1
     ST3   X,1        N - 1         Take man from circle.
     ENT1  0,3        N - 1
     CMPA  =N=        N - 1
     JL    1B         N - 1
     CHAR               1           One man left
     STX   X,1(4:5)     1           (he is clobbered too).
     OUT   X(18)        1           Print answer.
     HLT                1
     END   0B                       ▮
```

The last man is in position 11. The total time before output is $(4(N-1)(M+3)+8)u$. Is there a better method?

SECTION 1.3.3

1. (1 2 4)(3 6 5).

2. $a \leftrightarrow c$, $c \leftrightarrow f$; $b \leftrightarrow d$. The generalization to arbitrary permutations is clear.

3. $\begin{pmatrix} a & b & c & d & e & f \\ d & b & f & c & a & e \end{pmatrix}$.

4. $(a\ d\ c\ f\ e)$.

5. 12. (Cf. Ex. 20.)

6. The total time is increased by $4u$ for every blank word with the preceding nonblank word a "(", plus $5u$ for every blank word with the preceding nonblank word a name. Initial blanks and blanks between cycles do not affect the execution time. The position of blanks has no effect whatever on Program B.

7. $X = 2$, $Y = 29$, $M = 5$, $N = 7$, $U = 3$, $V = 1$. Total, by Eq. (18), $2165u$.

8. Yes; we would then keep the inverse of the permutation, i.e. x_i goes to x_j iff $T[j] = i$. (The final cycle form would then be constructed from the T table from right to left.)

9. No. For example, given (6) as input, Program A will produce "(ADG)(CEB)" as output, while Program B produces "(CEB)(DGA)". The answers are equivalent but not identical, due to the non-uniqueness of the cycle notation. The first element chosen for a cycle is the (a) leftmost available name, or (b) last available distinct name to be encountered from right to left, with Program A or B, respectively.

10. (1) Kirchhoff's law yields $A = 1 + C - D$; $B = A + J + P - 1$; $C = B - (P - L)$; $E = D - L$; $F = E + G - H$; $G = F - (J - H)$; $Q = Z$; $W = S$. (2) Interpretations: B = number of words of input $= 16X - 1$; C = number of nonblank words $= Y$; $D = C - M$; $E = D - M$; G = number of unsuccessful comparisons in names table search; $H = N$; $K = M$; $Q = N$; $R = U$; $S = R - V$; $T = N - V$ since each other name gets tagged. (3) Summing up, we have $(5G + 17Y + 80X + 18N - 21M + 10U - 17V)u$, which is somewhat better than Program A since G is certainly less than $16NX$. The time in the stated case is $962u$.

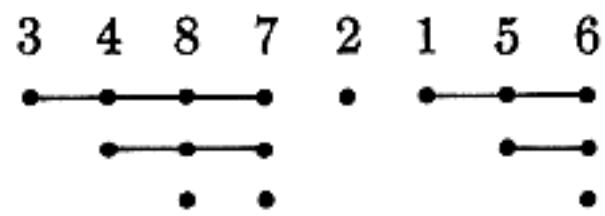
11. "Reflect" it; e.g., the inverse of $(acf)(bd)$ is $(db)(fca)$.

12. (a) The value in cell $L + mn - 1$ is fixed by the transposition, so we may omit it from consideration. Otherwise if $x = n(i - 1) + (j - 1) < mn - 1$, the value in $L + x$ should go to cell $L + (mx) \bmod N = L + \big(mn(i - 1) + m(j - 1)\big) \bmod N = L + m(j - 1) + (i - 1)$, since $mn \equiv 1 (\text{modulo } N)$ and $0 \le m(j - 1) + (i - 1) < N$. (b) If one bit in each memory cell is available (e.g. the sign, or the least significant bit of a floating-point value), we can "tag" elements as we move them, using an algorithm like algorithm I. Thus, (a) set $x \leftarrow N - 1$; (b) if CONTENTS$(L + x)$ has been tagged, go to (f), otherwise tag it and set $y \leftarrow$ CONTENTS$(L + x)$; (c) set $x \leftarrow (mx) \bmod N$; (d) exchange $y \leftrightarrow$ CONTENTS$(L + x)$; (e) if y is untagged, tag it and return to (d); (f) decrease x by 1 and if $x > 0$ return to (b). Reference: *JACM* **5** (1958), 383–384. If there is no room for a tag bit, tag bits can perhaps be kept in an auxiliary table, or else a list of representatives of all non-singleton cycles can be used: For each divisor d of N, we can transpose those elements which are multiples of d separately, since m is prime to N. The length of the cycle containing x, when $\gcd(x, N) = d$, is the smallest integer $r > 0$ such that $m^r \equiv 1$ (modulo N/d). For each d, we want to find $\varphi(N/d)/r$ representatives, one from each of these cycles. Some number-theoretic methods are available for this purpose, but they are not simple enough to be really

satisfactory; if too much space is not required we are probably best off storing a list of cycle representatives somewhere in memory by simply using a tagging method to compute these representatives when the program is first loaded. Reference: Gordon Pall and Esther Seiden, *Math. Comp.* **14** (1960), 189–192. See also J. Boothroyd, *CACM* **10** (1967), 292–3.

13. Show by induction that, at the beginning of step J2, $X[i] = +j$ if and only if $j > m$ and j goes to i under π; $X[i] = -j$ iff i goes to j under π^{k+1}, where k is the smallest non-negative integer such that π^k takes i into a number $\leq m$.

14. Writing the *inverse* of the given permutation in canonical cycle form and dropping parentheses, the quantity $A - N$ is the sum of the number of consecutive elements greater than a given element and immediately to its right. For example, if the original permutation is (165)(3784), the canonical form of the inverse is (3487)(2)(156); set up the array



and the quantity A is the number of "dots," 16. The number of dots below the k-th element is the number of right-to-left minima in the first k elements (i.e. there are 3 dots below 7 in the above, since there are 3 right-to-left minima in 3487). Hence the average is $H_1 + H_2 + \cdots + H_n = (n+1)H_n - n$.

15. If the first character of the linear representation is 1, the last character of the canonical representation is 1. If the first character of the linear representation is $m > 1$, then "... $1m$ ..." appears in the canonical representation. So the only solution is the permutation of order 1.

16. 1324,4231,3214,4213,2143,3412,2413,1243,3421,1324, ...

17. (a) The probability that the cycle is an m-cycle is $n!/m$ divided by $n!H_n$, so $p_m = 1/mH_n$. The average length is $p_1 + 2p_2 + 3p_3 + \cdots = \sum_{1\leq m\leq n} (m/mH_n) = n/H_n$. (b) Since the total number of m-cycles is $n!/m$, the total number of appearances of elements in m-cycles is $n!$. Each element appears as often as any other, by symmetry, so k appears $n!/n$ times in m-cycles. In *this* case, therefore, $p_m = 1/n$ for all k and m; the average is

$$\sum_{1\leq m\leq n} m/n = (n+1)/2.$$

18. See Ex. 22(e).

19. $|P_{n0} - n!/e| = 1/(n+1)! - 1/(n+2)! + \cdots$, an alternating series of decreasing magnitudes, which is less than $1/(n+1)! \leq \frac{1}{2}$.

20. Each m-cycle can be independently written in m ways; there are $\alpha_1 + \alpha_2 + \cdots$ cycles in all, which can be permuted among one another; and so the answer is

$$(\alpha_1 + \alpha_2 + \cdots)!1^{\alpha_1} \cdot 2^{\alpha_2} \cdot 3^{\alpha_3} \cdots .$$

21. $1/(\alpha_1!1^{\alpha_1}\alpha_2!2^{\alpha_2}\cdots)$ if $n = \alpha_1 + 2\alpha_2 + \cdots$; zero otherwise.

Proof: Write out α_1 1-cycles, α_2 2-cycles, etc., in a row, with empty positions; for example if $\alpha_1 = 1$, $\alpha_2 = 2$, $\alpha_3 = \alpha_4 = \cdots = 0$, we would have "(–) (– –) (– –)".

Fill the empty positions in all $n!$ possible ways; we obtain each permutation of the desired form exactly $\alpha_1!1^{\alpha_1}\alpha_2!2^{\alpha_2}\ldots$ times.

22. (a) If $k_1 + 2k_2 + \cdots = n$, the probability in (ii) is $\prod_{j\geq 0} f(w, j, k_j)$ which is assumed to equal $(1 - w)w^n/k_1!1^{k_1}k_2!2^{k_2}\ldots$; hence

$$\left(\prod_{j\geq 0} f(w, j, k_j)\right)^{-1} \prod_{j\geq 0} f(w, j, k_j + \delta_{jm}) = \frac{f(w, m, k_m + 1)}{f(w, m, k_m)} = \frac{w^m}{m(k_m + 1)}.$$

Therefore by induction

$$f(w, m, k) = \frac{1}{k!}\left(\frac{w^m}{m}\right)^k f(w, m, 0).$$

Condition (i) now implies that

$$f(w, m, k) = \frac{1}{k!}\left(\frac{w^m}{m}\right)^k e^{-w^m/m}.$$

[*Note:* Hence α_m is chosen with a "Poisson" distribution, see Ex. 1.2.10–16.]

(b)
$$\sum_{\substack{k_1+2k_2+\ldots=n\\ k_1,k_2,\ldots\geq 0}} \left(\prod_{j\geq 0} f(w, j, k_j)\right) = (1 - w)w^n \sum_{\substack{k_1+2k_2+\ldots=n\\ k_1,k_2,\ldots\geq 0}} P(n; k_1, k_2, \ldots)$$
$$= (1 - w)w^n.$$

Hence the probability that $\alpha_1 + 2\alpha_2 + \cdots \leq n$ is $(1 - w)(1 + w + \cdots + w^n) = 1 - w^{n+1}$.

(c) The average of ϕ is

$$\sum_{n\geq 0}\left(\sum_{k_1+2k_2+\ldots=n} \phi(k_1, k_2, \ldots) \text{ prob } (\alpha_1 = k_1, \alpha_2 = k_2, \ldots)\right)$$
$$= (1 - w)\sum_{n\geq 0} w^n \left(\sum_{k_1+2k_2+\ldots=n} \phi(k_1, k_2, \ldots)/k_1!1^{k_1}k_2!2^{k_2}\ldots\right).$$

(d) Let $\phi(\alpha_1, \alpha_2, \ldots) = \alpha_2 + \alpha_4 + \alpha_6 + \cdots$. Since the α's are independently chosen, the average value of the linear combination ϕ is the sum of the average values of $\alpha_2, \alpha_4, \alpha_6, \ldots$; the average value of α_m is

$$\sum_{k\geq 0} kf(w, m, k) = \sum_{k\geq 1} \frac{1}{(k-1)!}\left(\frac{w^m}{m}\right)^k e^{-w^m/m} = \frac{w^m}{m}.$$

Therefore the average value of ϕ is

$$\frac{w^2}{2} + \frac{w^4}{4} + \cdots = (1 - w)(\tfrac{1}{2}w^2 + \tfrac{1}{2}w^3 + (\tfrac{1}{2} + \tfrac{1}{4})w^4 + \cdots).$$

The desired answer is

$$\sum_{\substack{0\leq k\leq n\\ k \text{ even}}} \frac{1}{k} = \tfrac{1}{2}H_{\lfloor n/2\rfloor}.$$

(e) Let z be a real number; let $\phi(\alpha_1, \alpha_2, \ldots) = z^{\alpha_m}$. The average value of ϕ is

$$\sum_{k\geq 0} f(w, m, k)z^k = \sum_{k\geq 0} \frac{1}{k!}\left(\frac{w^m z}{m}\right)^k e^{-w^m/m} = e^{w^m(z-1)/m} = \sum_{j\geq 0} \frac{w^{mj}}{j!}\left(\frac{z-1}{m}\right)^j$$
$$= (1-w)\sum_{n\geq 0} w^n \left(\sum_{0\leq j\leq n/m} \frac{1}{j!}\left(\frac{z-1}{m}\right)^j\right)$$
$$= (1-w)\sum_{n\geq 0} w^n G_{nm}(z).$$

Hence

$$G_{nm}(z) = \sum_{0\leq j\leq n/m} \frac{1}{j!}\left(\frac{z-1}{m}\right)^j; \; p_{nmk} = \frac{1}{m^k k!} \sum_{0\leq j\leq n/m-k} \frac{(-1/m)^j}{j!};$$

(min 0, ave $1/m$, max n/m, dev $\sqrt{1/m}$).

23. The constant λ is $\int_0^t e^{\mathrm{li}(u)}\,du$, where $\mathrm{li}(x) = \int_0^x dt/(\ln t)$. See *Transactions of the American Math. Society* **121** (1966), 340–357; many other results are proved in this paper, in particular the average length of the *shortest* cycle is approximately $\ln n/e^\gamma$. Further terms of the asymptotic representation of l_n are not yet known. William C. Mitchell has calculated a high-precision value of $\lambda = .62432\ 99885\ 43550\ 87099\ 29363\ 8310\ldots$; no relation between λ and classical mathematical constants is known.

25. One proof, by induction on N, is based on the fact that when the Nth element is a member of s of the sets it contributes exactly

$$\binom{s}{0} - \binom{s}{1} + \binom{s}{2} - \cdots = (1-1)^s = \delta_{s0}$$

to the sum. Another proof, by induction on M, is based on the fact that the number of elements that are in S_M but not in $S_1 \cup \cdots \cup S_{M-1}$ is

$$\|S_M\| - \sum_{1\leq j<M} \|S_j \cap S_M\| + \sum_{1\leq j<k<M} \|S_j \cap S_k \cap S_M\| - \cdots.$$

26. Let $N_0 = N$ and let

$$N_k = \sum_{1\leq j_1<\cdots<j_k\leq M} \|S_{j_1} \cap \cdots \cap S_{j_k}\|.$$

Then the desired formula is

$$N_r - \binom{r+1}{r} N_{r+1} + \binom{r+2}{r} N_{r+2} - \cdots.$$

This may be proved from the principle of inclusion and exclusion itself, or by using the formula

$$\binom{r}{r}\binom{s}{r} - \binom{r+1}{r}\binom{s}{r+1} + \cdots = \binom{s}{r}\binom{s-r}{0} - \binom{s}{r}\binom{s-r}{1} + \cdots = \delta_{sr}$$

as in exercise 25.

27. Let S_j be the multiples of m_j in the stated range and let $N = am_1 \ldots m_t$. Then

$$\|S_j \cap S_k\| = N/m_j m_k, \text{ etc.},$$

so the answer is

$$N - N \sum_{1 \le j \le t} \frac{1}{m_j} + N \sum_{1 \le j < k \le t} \frac{1}{m_j m_k} - \cdots = N\left(1 - \frac{1}{m_1}\right) \cdots \left(1 - \frac{1}{m_t}\right).$$

This provides another proof of exercise 1.2.4–30, if we let $m_1, \ldots, m_t$ be the primes dividing N.

SECTION 1.4.1

1. Calling sequence: JMP MAXN; or, JMP MAX100 if $n = 100$.
   Entry conditions: For the MAXN entrance, rI3 $= n$; assume $n \geq 1$.
   Exit conditions: Same as in (4).

2.
```
MAX50  STJ   EXIT
       ENT3  50
       JMP   2F
```

3. Entry conditions: $n =$ rI1 if rI1 > 0; otherwise $n = 1$.
   Exit conditions: rA and rI2 as in (4); rI1 unchanged; rI3 = min (0, rI1); rJ = EXIT + 1; CI unchanged if $n = 1$, otherwise CI is greater, equal, or less, according as the maximum is greater than $X[1]$, equal to $X[1]$ and rI2 > 1, or equal to $X[1]$ with rI2 $= 1$. (The analogous exercise for subroutine (9) would of course be somewhat more complicated.)

4.
```
SMAX1  ENT1  1      r = 1
SMAX   STJ   EXIT   general r
       JMP   2F     continue as before
...
       DEC3  0,1    decrease by r
       J3P   1B
EXIT   JMP   *      exit.
```

Calling sequence: JMP SMAX; or, JMP SMAX1 if $r = 1$.
Entry conditions: rI3 $= n$, assumed positive; for the SMAX entrance, rI1 $= r$, assumed positive.
Exit conditions: rA $= \max_{0 \leq k < n/r}$ CONTENTS(X $+ n - kr$) $=$ CONTENTS(X + rI2); rI3 $= ((n - 1) \bmod r) + 1 - r$.

5. Any other register can be used. For example,

   Calling sequence: ENTA *+2
   JMP MAX100
   Entry conditions: None.
   Exit conditions: Same as in Ex. 1.

The code is the same as (1) except the first instruction would be

"MAX100 STA EXIT(0:2)".

6. (Solution by Joel Goldberg.)

```
MOVE  STJ   1F
      STJ   2F
      STJ   3F
      STA   4F
      ST2   5F(0:2)
1H    LDA   *(0:3)
      STA   *+1(0:3)
      ENTA  *
2H    LD2N  *(4:4)
      J2Z   1F
      DECA  0,2
      STA   2F(0:2)
2H    LDA   *,2
      STA   0,1
      INC1  1
      INC2  1
      J2N   2B
1H    LDA   4F
5H    ENT2  *
3H    JMP   *
4H    CON   0
```

∎

SECTION 1.4.2

1. If one coroutine calls the other only once, it is nothing but a subroutine; so we need an application in which each coroutine calls the other in at least two distinct places. Even then, it is often easy to set some sort of switch or use some property of the data, so that upon entry to a fixed place within one coroutine it is possible to branch to one of two desired places—so again, nothing more than a subroutine would be required. Coroutines become correspondingly more useful as the number of references between them grows larger.

2. The first character found by `IN` would be lost.

3. *Almost* true, since "`CMPA =10=`" within `IN` is then the only comparison instruction of the program, and since the code for "." is 40. But if the final period were preceded by a replication digit, the test would go unnoticed. (*Note:* The most efficient program would probably remove lines 40, 44, and 48, and would insert "`CMPA PERIOD`" between lines 26 and 27. If the state of the comparison indicator is to be used across coroutines, however, it must be recorded as part of the coroutine characteristics in the documentation of the program.)

4. (a) On the IBM 650, using SOAP assembly language, we would have the calling sequences "LDD A" and "LDD B"; and linkage "A STD BX AX" and "B STD AX BX" (with the two linkage instructions preferably in core). (b) On the IBM 709, using common assembly languages, the calling sequences would be "TSX A,4" or "TSX B,4"; the linkage instructions would be

```
A   SXA  BX,4                    B   SXA  AX,4
AX  AXT  1-A1,4                  BX  AXT  1-B1,4
    TRA  1,4                         TRA  1,4
```

(c) On the CDC 1604, the calling sequences would be "return jump" (SLJ 4) to A or B, and the linkage would be, e.g.,

```
B:  SLJ  A1;  ALS  0
A:  SLJ  B1;  SLJ  B.
```

(d) Most other machines are similar to one of these three. For example, System/360 would be analogous to the 709, or we could use BALR r,r in short coroutines.

5. "STA HOLDAIN; LDA HOLDAOUT" between OUT and OUTX, and "STA HOLDAOUT; LDA HOLDAIN" between IN and INX.

6. Within A write "JMP AB" to activate B, "JMP AC" to activate C. Similarly locations BA, BC, CA, and CB would be used within B and C. The linkage is:

```
AB  STJ  AX                  BC  STJ  BX                  CA  STJ  CX
BX  JMP  B1                  CX  JMP  C1                  AX  JMP  A1
CB  STJ  CX                  AC  STJ  AX                  BA  STJ  BX
    JMP  BX                      JMP  CX                      JMP  AX
```

(*Note:* With n coroutines, $2(n-1)n$ cells would be required for the linkage. If n is large, a "centralized" routine for linkage could of course be used; a method with $3n+2$ cells would not be hard to invent. But in practice the faster method above requires just $2m$ cells, where m is the number of pairs (i, j) such that coroutine i jumps to coroutine j. When there are many coroutines each independently jumping to others, we usually have a situation in which the sequence of control is under external influence, as discussed in Section 2.2.5.)

SECTION 1.4.3.1

1. It is used only twice, both times immediately followed by a call on **MEMORY**, so it would be slightly more efficient to make it a special entrance to the **MEMORY** subroutine, and also to make it put $-R$ in rI2.

2.

```
SHIFT  J5N   ADDRERROR
       DEC3  5
       J3P   FERROR
       LDA   AREG
       LDX   XREG
       LD1   1F,3(4:5)
       ST1   2F(4:5)
       J5Z   CYCLE
```

```
2H      SLA   1
        DEC5  1
        J5P   2B
        JMP   STOREAX
        SLA   1
        SRA   1
        SLAX  1
        SRAX  1
        SLC   1
1H      SRC   1          ▮
```

3.

```
MOVE    J3Z   CYCLE
        JMP   MEMORY
        SRAX  5
        LD1   I1REG
        LDA   SIGN1
        JAP   *+2
        J1NZ  MEMERROR
        CMP1  =BEGIN=
        JGE   MEMERROR
        STX   0,1
        LDA   CLOCK
        INCA  2
        STA   CLOCK
        INC1  1
        ST1   I1REG
        INC5  1
        DEC3  1
        JMP   MOVE  ▮
```

4. Just insert "`IN 0(16)`" and "`JBUS *(16)`" between lines 03 and 04. (Of course on another computer this would be considerably different since it would be necessary to convert to `MIX` character code.)

5. Central control time is $34u$, plus $15u$ if indexing is required; the `GETV` subroutine takes $52u$, plus $5u$ if $L \neq 0$; extra time to do the actual loading is $11u$ for `LDA` or `LDX`, $13u$ for `LDi`, $21u$ for `ENTA` or `ENTX`, $23u$ for `ENTi` (add $2u$ to the latter two times if `M` = 0). Summing up, we have a total time of $97u$ for `LDA` and $55u$ for `ENTA`, plus $15u$ for indexing, and plus $5u$ or $2u$ in certain other circumstances. It would seem that simulation in this case is causing roughly a 50:1 ratio in speeds. (Results of a test run which involved $178u$ of simulated time required $8422u$ of actual time, a 47:1 ratio.)

7. Execution of `IN` or `OUT` sets a variable associated with the appropriate input device to the time when transmission is desired. The "`CYCLE`" control routine interrogates these variables on each cycle, to see if `CLOCK` has exceeded either (or both) of them; if so, the transmission is carried out and the variable is set to "infinity". (When more than two I/O units must be handled in this way, there will be so many variables it will be preferable to keep them in a sorted list using linked memory techniques; see Section 2.2.5.)

1. Change lines 49 and 50 to the following sequence:

```
LEAVE  STX   3F
       ST1   4F
       LD1   JREG
       LDA   -1,1
       LDX   1F
       STX   -1,1
       JMP   -1,1
1H     JMP   *+1
       STA   -1,1
2H     LD1   4F
       LDX   3F
       LDA   AREG
LEAVEX JSJ   *
3H     CON   0
4H     CON   0
```

The operator "JSJ" here is, of course, particularly crucial.

2.

```
. . . . . . . . . . . lines 01–04
       ST1    I1REG
. . . . . . . . . . . lines 05–07
BUF    LD1    BUF2
       STA    1,1
. . . . . . . . . . . lines 08–12
       STA    2,1
       LDA    AREG
       STA    3,1
       STX    4,1
       LDA    I1REG
       STA    5,1
       ST2    6,1
       ST3    7,1
       ST4    8,1
       ST5    9,1
       ST6    10,1
       LDA    JREG
       STA    11,1
       STZ    12,1
       ENTA   1
       JNOV   1F
       STA    12,1
       LDA    BIG
       ADD    BIG
1H     STZ    13,1
       ST1    BUF(0:2)
. . . . . . . . . . . lines 13–30
```

```
        LD1    I1REG
. . . . . . . . . . . . lines 31–34
        ST1   I1REG
. . . . . . . . . . . . lines 35–49
        LD1    I1REG
. . . . . . . . . . . . lines 50–53
BIG     CON    50(1:1)
BUF1    CON    *+14
        ORIG   *+13
BUF2    CON    BUF1
        ORIG   *+13
```

Note: Since only 13/100 of each tape record gets useful information, this program is not using tape very efficiently, and since the tape time is the limiting factor the speed of tracing could also be improved if we put several 13-word groups into each tape record. The program could be readily modified to have 14 buffers, with a tape record written only after seven buffers are full (linking the buffers together and using a sentinel); this would make the tape length and execution time $\frac{1}{7}$ as long, but it would require much more memory.

3. Tape is faster; and the editing of this information into characters while tracing would consume far too much space.

4, 5. A true trace, as desired in Ex. 6, would not be obtained, since restriction (a) mentioned in the text is violated. Early examination leads to speculation that the first attempt to trace CYCLE would cause a loop back to ENTER.

6. Suggestion: keep a table of values of each memory location within the trace area that has been changed by the outer program.

SECTION 1.4.4

1. (a) No, the input operation may not yet be complete. (b) No, the input operation may be going just a little faster, and this is much too risky.

2.
```
ENT1  2000
JBUS  *(6)
MOVE  1000(50)
MOVE  1050(50)
OUT   2000(6)
```

3.
```
WORDOUT  STJ   1F
         INC5  1
         LDX   0,5
         JXZ   2F
         OUT   -100,5(V)
         LD5   1,5
         ENT1  0,5
         MOVE  -1,1(50)
         MOVE  -1,1(50)
         ST5   CURRENT(0:2)
```

```
2H        STA   0,5
1H        JMP   *
* BUFFER AREAS
          CON   0
OUTBUF1   ORIG  *+100
          CON   -1
          CON   *+2
          CON   0
OUTBUF2   ORIG  *+100
          CON   -1
          CON   OUTBUF1
```

At the beginning of the program, give the instruction "ENT5 OUTBUF1-1". At the end of the program, put

```
CURRENT  OUT   *(V)          Write out last record.
         OUT   OUTBUF1(V)  } (optional; writes an extra record in case of
         IOC   0(V)        } later input buffering, and rewinds the tape)
```

4. If the calculation time exactly equals the I/O time (which is the most favorable situation), both the computer and peripheral device running simultaneously will take half as long as if they ran separately. Formally, let C be the calculation time for the entire program, and let T be the total I/O time required; then the best possible running time with buffering is max (C, T), while the running time without buffering is $C + T$; and of course $\frac{1}{2}(C + T) \leq \max(C, T) \leq C + T$. However, there are some devices which have a "shutdown penalty" which causes an extra amount of time to be lost if too long an interval occurs between references to that unit; in such a case, better than 2:1 ratios are possible.

5. Best ratio is $(n + 1):1$.

6. $\left\{\begin{matrix}\text{IN} & \text{INBUF1(U)} \\ \text{ENT6} & \text{INBUF2+99}\end{matrix}\right\}$ or $\left\{\begin{matrix}\text{IN} & \text{INBUF2(U)} \\ \text{ENT6} & \text{INBUF1+99}\end{matrix}\right\}$

(possibly preceded by IOC 0(U) to rewind the tape just in case it is necessary).

7. One way is to use coroutines:

```
INBUF1    ORIG  *+100
          CON   *+1
INBUF2    ORIG  *+100
          CON   INBUF1
1H        LDA   INBUF2+100,6
          JMP   MAIN
          INC6  1
          J6N   1B
WORDIN1   IN    INBUF2(U)
          ENN6  100
2H        LDA   INBUF1+100,6
          JMP   MAIN
          INC6  1
          J6N   2B
```

```
          IN     INBUF1(U)
          ENN6   100
          JMP    1B
WORDIN    STJ    MAINX
WORDINX   JMP    WORDIN1
MAIN      STJ    WORDINX
MAINX     JMP    *
```

Adding a few more instructions to take advantage of special cases will make this routine actually faster than (4).

8. At the time shown in Fig. 23, the two red buffers have been filled with line images, and the one indicated by NEXTR is being printed. At the same time, the program is computing between RELEASE and ASSIGN. When the program ASSIGNs, the green buffer indicated by NEXTG becomes yellow; NEXTG moves clockwise and the program begins to fill the yellow buffer. When the output operation is complete, NEXTR moves clockwise, the buffer that has just been printed turns green, and the remaining red buffer begins to be printed. Finally, the program RELEASEs the yellow buffer and it too is ready for subsequent printing.

9, 10, 11:

time	*action* ($N = 1$)	*action* ($N = 2$)	*action* ($N = 4$)
0	ASSIGN(BUF1)	ASSIGN(BUF1)	ASSIGN(BUF1)
1000	RELEASE, OUT BUF1	RELEASE, OUT BUF1	RELEASE, OUT BUF1
2000	ASSIGN(wait)	ASSIGN(BUF2)	ASSIGN(BUF2)
3000		RELEASE	RELEASE
4000		ASSIGN(wait)	ASSIGN(BUF3)
5000			RELEASE
6000			ASSIGN(BUF4)
7000			RELEASE
8000			ASSIGN(wait)
8500	BUF1 assigned, output stops	BUF1 assigned, OUT BUF2	BUF1 assigned, OUT BUF2
9500	RELEASE, OUT BUF1	RELEASE	
10500	ASSIGN(wait)	ASSIGN(wait)	
15500			RELEASE

and so on. Total time when $N = 1$ is $110000u$; when $N = 2$ it is $89000u$; when $N = 3$ it is $81500u$; and when $N \geq 4$ it is $76000u$.

12. The following code should be inserted before "LD5 -1,5" in program B:

```
STA    2F
LDA    3F
CMPA   15,5(5:5)
LDA    2F
```

Then the instruction "JMP 1B" should be changed to

```
JNE    1B
JMP    COMPUTE
JMP    *-1        [or JMP COMPUTEX]
```

```
2H   CON   0
3H   ALF        .      ▮
```

13.

```
     JRED  CONTROL(U)
     J6NZ  *-1         ▮
```

14. If $N = 1$ the process would loop indefinitely; otherwise the construction will have the effect that there are two yellow buffers. This can be useful if the computational program wants to refer to two buffers at once, although it ties up buffer space. In general, the excess of ASSIGNs over RELEASEs should be nonnegative and not greater than N.

15.

```
U        EQU   0
V        EQU   1
BUF1     ORIG  *+100
BUF2     ORIG  *+100
BUF3     ORIG  *+100
TAPECPY  ENT1  99
         IN    BUF1(U)
1H       IN    BUF2(U)
         OUT   BUF1(V)
         IN    BUF3(U)
         OUT   BUF2(V)
         IN    BUF1(U)
         OUT   BUF3(V)
         DEC1  3
         J1P   1B
         OUT   BUF1(V)
         HLT
         END   TAPECPY   ▮
```

This is a special case of the algorithm indicated in Fig. 26.

18. Partial solution: in the algorithms below, t is a variable which is set to 0 when the I/O device is active, and $t = 1$ when it is idle.

Algorithm A (ASSIGN, a normal state subroutine).
This algorithm is unchanged from Algorithm 1.4.4A.

Algorithm R (RELEASE, a normal state subroutine).

R1. Increase n by one.
R2. If $t = 0$, cause an interrupt (using the INT operator) which should go to step B2. ▮

Algorithm B (Buffer control routine, which processes interrupts).

B1. If $n = 0$, set $t \leftarrow 0$ and restart main program.
B2. Set $t \leftarrow 1$, and initiate I/O from the buffer area specified by NEXTR.
B3. Restart the main program; an "I/O Complete" condition will interrupt to step B4.
B4. Advance NEXTR to the next clockwise buffer.
B5. Decrease n by one, and go to step B1. ▮

SECTION 2.1

1. (a) TAG(NEXT(TOP)) = TAG(NEXT(242)) = TAG(386) = 0. (b) Λ.

2. Whenever V is a link variable (else CONTENTS(V) makes no sense) whose value is not Λ. It is wise to *avoid* using LOC in contexts like this.

3. Set NEWCARD $\leftarrow$ TOP, and if TOP $\neq \Lambda$ set TOP $\leftarrow$ NEXT(TOP).

4. **C1.** Set X $\leftarrow$ LOC(TOP). (For convenience we make the reasonable assumption that TOP = NEXT(LOC(TOP)). This assumption is compatible with program (5).)

C2. If NEXT(X) $\neq \Lambda$, set X $\leftarrow$ NEXT(X) and repeat this step.

C3. Set NEXT(X) $\leftarrow$ NEWCARD, NEXT(NEWCARD) $\leftarrow \Lambda$, TAG(NEWCARD) $\leftarrow$ 1. ▮

5. **D1.** Set X $\leftarrow$ LOC(TOP), Y $\leftarrow$ TOP. (See step C1 above. By hypothesis, Y $\neq \Lambda$. Throughout the algorithm which follows, X trails one step behind Y in the sense that Y = NEXT(X).)

D2. If NEXT(Y) $\neq \Lambda$, set X $\leftarrow$ Y, Y $\leftarrow$ NEXT(Y), and repeat this step.

D3. (Now NEXT(Y) = Λ, so Y points to the bottom card; also X points to the next-to-last card.) Set NEXT(X) $\leftarrow \Lambda$, NEWCARD $\leftarrow$ Y. ▮

6. (b) and (d). *Not* (a)! CARD is a node, not a link to a node.

7. Sequence (a) gives NEXT(LOC(TOP)), which in this case is identical to TOP; sequence (b) is correct. Note the following important distinction, when X is a link variable: In an algorithm, the symbol X refers to the value of variable X; but in assembly language it refers to the *location* of X. There is no need for confusion; consider the analogous example when X is a numeric variable: To bring X into register A, we write LDA X, not ENTA X.

8. Let rA $\equiv$ N, rI1 $\equiv$ X.

```
ENTA   0            N ← 0.
LD1    TOP          X ← TOP.
J1Z    *+4          Is X = Λ?
INCA   1            N ← N + 1.
LD1    0,1(NEXT)    X ← NEXT(X).
J1NZ   *-2                ▮
```

9. Let rI2 $\equiv$ X.

```
PRINTER  EQU   18            Unit number for printer
TAG      EQU   1:1
NEXT     EQU   4:5           Definition of fields
NAME     EQU   0:5
PBUF     ALF   PILE          Message printed in case
         ALF   EMPTY           pile is empty
         ORIG  PBUF+24
BEGIN    LD2   TOP           Set X ← TOP.
         J2Z   2F            Is the pile empty?
1H       LDA   0,2(TAG)      rA ← TAG(X).
         ENT1  PBUF          Get ready for MOVE instruction.
         JBUS  *(PRINTER)    Wait until printer is ready.
         JAZ   *+3           Is TAG = 0 (is card face up)?
```

```
         MOVE   PAREN(3)          No: set parentheses.
         JMP    *+2
         MOVE   BLANKS(3)         Yes: set blanks.
         LDA    1,2(NAME)         rA ← NAME(X).
         STA    PBUF+1
         LD2    0,2(NEXT)         Set X ← NEXT(X).
2H       OUT    PBUF(PRINTER)     Print the line.
         J2NZ   1B                If X ≠ A, repeat the print loop.
DONE     HLT
PAREN    ALF        (
BLANKS   ALF
         ALF    )
         ALF                      ▮
```

SECTION 2.2.1

1. Yes (consistently insert all items at one of the two ends).

2. To obtain 325641, do SSSXXSSXSXXX (in the notation of the following exercise). The order 154623 cannot be achieved, since 2 can precede 3 only if it is removed from the stack before 3 has been inserted.

3. An admissible sequence is one in which the number of X's never exceeds the number of S's if we read from the left to the right.

Two different admissible sequences must give a different result, since if the two sequences agree up to a point where one has S and the other has X, the latter sequence outputs a symbol which cannot possibly be output before the symbol just inserted by the S of the former sequence.

4. This problem is equivalent to many other interesting problems, such as the enumeration of binary trees, the number of ways to insert parentheses into a formula, and the number of ways to divide a polygon into triangles, and it appeared as early as 1759 in notes by Euler and Segner (see Section 2.3.4.6). For further references, see A. Erdélyi and I. M. H. Etherington, *Edinburgh Mathematical Notes*, **32** (1940), 1–12.

The following elegant solution is due to D. André (1878): There are obviously $\binom{2n}{n}$ sequences of S's and X's that contain n of each. It remains to evaluate the number of *inadmissible* sequences (which contain the right number of S's and X's but which violate the other condition). In any inadmissible sequence, locate the first X for which the X's outnumber the S's. Then in the partial sequence leading up to and including this X, replace all X's by S and all S's by X. The result is a sequence with $(n + 1)$ S's and $(n - 1)$ X's. Conversely for every sequence of the latter type we can reverse the process and find the inadmissible sequence of the former type which leads to it. For example, the sequence XXSXSSSXXSX must have come from SSXSXXXXXSX. This correspondence shows that the number of inadmissible sequences is $\binom{2n}{n-1}$. Hence $a_n = \binom{2n}{n} - \binom{2n}{n-1}$.

Using the same idea, we can solve the more general "ballot problem" of probability theory, which essentially is the enumeration of all partial admissible sequences with a given number of S's and X's. For the history of the ballot problem and some generalizations, see papers by A. Dvoretzky and T. Motzkin, *Duke Math. Journal* **14** (1947),

305–313; L. Takács, *Journal of the American Statistical Association* **57** (1962), 327; John Riordan, *Annals of Math. Statistics* **35** (1964), 369–379; and exercise 2.3.4.4–32.

We present here a new method for solving the ballot problem with the use of double generating functions, since this method lends itself to the solution of more difficult problems such as Ex. 11.

Let g_{nm} be the number of sequences of S's and X's of length n, in which the number of X's never exceeds the number of S's if we count from the left, and in which there are m more S's than X's in all. Then $a_n = g_{(2n)0}$. Obviously g_{nm} is zero unless $m + n$ is even. The recurrence relation satisfied by these numbers is easily found to be

$$g_{(n+1)m} = g_{n(m-1)} + g_{n(m+1)}, \quad m \geq 0, \; n \geq 0; \; g_{0m} = \delta_{0m}.$$

We set up the double generating function $G(x, z) = \sum_{n,m} g_{nm} x^m z^n$, and let $g(z) = G(0, z)$. The recurrence relation above transforms into

$$\left(x + \frac{1}{x}\right) G(x, z) = \frac{1}{x} g(z) + \frac{1}{z}\left(G(x, z) - 1\right), \quad \text{i.e.} \quad G(x, z) = \frac{zg(z) - x}{z(x^2 + 1) - x}.$$

This equation unfortunately tells us nothing if we set $x = 0$, but we can proceed by factoring the denominator as $z(1 - r_1(z)x)(1 - r_2(z)x)$ where

$$r_1(z) = \frac{1}{2z}(1 + \sqrt{1 - 4z^2}), \qquad r_2(z) = \frac{1}{2z}(1 - \sqrt{1 - 4z^2}).$$

(Note that $r_1 + r_2 = 1/z$; $r_1 r_2 = 1$.) We now proceed heuristically; the problem is to find some value of $g(z)$ such that $G(x, z)$ as given by the formula above has an infinite power series expansion in x and z. Note that $r_2(z)$ has such an expansion, and $r_2(0) = 0$; and for fixed z, the value $x = r_2(z)$ causes the denominator of $G(x, z)$ to vanish. This suggests that we might choose $g(z)$ so that the numerator also vanishes when $x = r_2(z)$, i.e. take $zg(z) = r_2(z)$. The equation for $G(x, z)$ now simplifies to

$$G(x, z) = \frac{r_2(z)}{z(1 - r_2(z)x)} = \sum_{n \geq 0} (r_2(z))^{n+1} x^n z^{-1}.$$

Since this is a power series expansion which satisfies the original equation, we must have found the right choice of $g(z)$.

The coefficients of $g(z)$ are the solution to our problem. Actually we can go further and derive a simple form for all the coefficients of $G(x, z)$: By the binomial theorem,

$$r_2(z) = \sum_{k \geq 0} z^{2k+1} \binom{2k+1}{k} \frac{1}{2k+1}.$$

Let $w = z^2$ and $r_2(z) = zf(w)$. Then

$$f(w) = \sum_{k \geq 0} A_k(1, -2) w^k$$

in the notation of Ex. 1.2.6–25; hence

$$f(w)^r = \sum_{k \geq 0} A_k(r, -2) w^k.$$

We now have

$$G(x, z) = \sum_{n,m} A_m(n, -2)x^n z^{2m+n},$$

so the general solution is

$$g_{(2n)(2m)} = \binom{2n+1}{n-m}\frac{2m+1}{2n+1} = \binom{2n}{n-m} - \binom{2n}{n-m-1};$$

$$g_{(2n+1)(2m+1)} = \binom{2n+2}{n-m}\frac{2m+2}{2n+2} = \binom{2n+1}{n-m} - \binom{2n+1}{n-m-1};$$

$$a_n = \binom{2n+1}{n}\frac{1}{2n+1}.$$

5. If $j < k$ and $p_j < p_k$, we must have taken p_j off the stack before p_k was put on; if $p_j > p_k$, we must have left p_k on the stack until after p_j was put on. Combining these two rules, the condition $i < j < k$ and $p_j < p_k < p_i$ is impossible since it means p_j must go off before p_k and after p_i, yet p_i appears after p_k.

Conversely, the desired permutation can be obtained by using the algorithm "For $j = 1, 2, \ldots, n$ input zero or more items (as many as necessary) until p_j first appears in the stack, then output p_j." This algorithm can fail only if we reach a j for which p_j is not at the top of the stack but it is covered by some element p_k for $k > j$. Since the contents of the stack is always monotone increasing, we have $p_j < p_k$. This element p_k could have gotten there only if it is less than p_i for some $i < j$.

6. Only the trivial one, $1\ 2 \ldots n$, by the nature of a queue.

7. An input-restricted deque which first outputs n must simply put the values $1\ 2 \ldots n$ on the deque in order as its first n operations. An output-restricted deque which first outputs n must put the values $p_1\ p_2 \ldots p_n$ on its deque as its first n operations. Therefore we find the answers (a) 4132 (b) 4213 (c) 4231. By trial, these answers are unique.

8. When $n = 4$, no; when $n = 5$, there are four: 51324, 51342, 52314, 52341.

9. By operating in reverse, we can get the inverse of any input-restricted permutation with an output-restricted deque, and conversely. This sets up a one-to-one correspondence between the two sets of permutations.

10. (i) There should be n X's and n combined S's and Q's. (ii) The number of X's must never exceed the combined number of S's and Q's, if we read from the left. (iii) Whenever the number of X's equals the combined number of S's and Q's (reading from the left), the next character must be a Q. (iv) The two operations XQ must never be adjacent in this order.

Clearly rules (i) and (ii) are necessary. The extra rules (iii) and (iv) are added to remove ambiguity, since S is the same as Q when the scroll is empty, and since XQ can always be replaced by QX. Thus, any obtainable sequence corresponds to at least one admissible sequence.

To show that two admissible sequences give different permutations, consider sequences which are identical up to a point, and then one sequence has an S while the other has an X or Q. Since by (iii) the deque is not empty, clearly different permutations (relative to the order of the element moved on by S) are obtained by the two

sequences. The remaining case is where sequences A, B agree up to a point and then sequence A has Q, sequence B has X. Sequence B may have further X's at this point, and by (iv) they must be followed by an S, so again the permutations are different.

11. Proceeding as in Ex. 4, we let g_{nm} be the number of partial admissible sequences of length n, leaving m elements on the deque, *not* ending in the symbol X; h_{nm} is defined analogously, for those sequences that *do* end with X. We have $g_{(n+1)m} = 2g_{n(m-1)}$ $(+h_{n(m-1)}$ if $m > 1)$; $h_{(n+1)m} = g_{n(m+1)} + h_{n(m+1)}$. Define $G(x, z)$ and $H(x, z)$ analogously to the definition in Ex. 4; we have

$$G(x, z) = xz + 2x^2z^2 + 4x^3z^3 + (8x^4 + 2x^2)z^4 + (16x^5 + 8x^3)z^5 + \cdots;$$

$$H(x, z) = z^2 + 2xz^3 + (4x^2 + 2)z^4 + (8x^3 + 6x)z^5 + \cdots.$$

If $h(z) = H(0, z)$, we find

$$\frac{1}{z} G(x, z) = 2xG(x, z) + x\big(H(x, z) - h(z)\big) + x,$$

$$\frac{1}{z} H(x, z) = \frac{1}{x} G(x, z) + \frac{1}{x}\big(H(x, z) - h(z)\big).$$

Consequently

$$G(x, z) = \frac{xz\big(x - z - xh(z)\big)}{x - z - 2x^2z + xz^2}.$$

As in Ex. 4, we try choosing $h(z)$ so the numerator cancels with a factor of the denominator. We find $G(x, z) = xz/\big(1 - 2r_2(z)\big)$ where

$$r_2(z) = \frac{1}{4z}\left(z^2 + 1 - \sqrt{(z^2 + 1)^2 - 8z^2}\right).$$

Using the convention $b_0 = 1$, the desired generating function comes to

$$\tfrac{1}{2}(3 - z - \sqrt{1 - 6z + z^2}) = 1 + z + 2z^2 + 6z^3 + 22z^4 + 90z^5 + \cdots.$$

By differentiation we find a recurrence relation that is handy for calculation: $nb_n = 3(2n - 3)b_{n-1} - (n - 3)b_{n-2}$, $n \geq 2$.

12. If $0 < \alpha < 1$, the coefficient of z^n in $\sqrt{1 - z}\sqrt{1 - \alpha z}$ is

$$(-1)^n \binom{\frac{1}{2}}{n}\left(\sqrt{1 - \alpha} + O\left(\frac{1}{n}\right)\right)$$

which by Stirling's approximation is asymptotically

$$-\frac{1}{2}\sqrt{\frac{1 - \alpha}{\pi n^3}}.$$

Now $1 - 6z + z^2 = \big(1 - (3 + \sqrt{8})z\big)\big(1 - (3 - \sqrt{8})z\big)$. Substitute $w = (3 + \sqrt{8})z$ and apply the above formulas; we find $a_n \sim 4^n/\sqrt{\pi n^3}$; $b_n \sim c(3 + \sqrt{8})^n n^{-3/2}$, where $c = \frac{1}{2}\sqrt{(3\sqrt{2} - 4)/\pi} \approx 0.139$.

SECTION 2.2.2

1. M — 1 (*not* M). If we allowed M items, as (6) and (7) do, it would be impossible to distinguish an empty queue from a full one by examination of R and F, since only M possibilities can be detected. It is better to give up one storage cell than to make the program overly complicated!

2. Delete from rear: if R = F then UNDERFLOW; Y ← X[R]; if R = 1 then R ← M, otherwise R ← R — 1. Insert at front: (Assume that initially R = F = 1 instead of R = F = 0.) Set X[F] ← Y; if F = 1 then F ← M, otherwise F ← F — 1; if F = R then OVERFLOW.

3. (a) LD1 I; LDA BASE,7:1. This takes 5 cycles instead of 4 or 8 as in (8).

(b) *Solution 1:* LDA BASE,2:7 where each base address is stored with $I_1 = 0$, $I_2 = 1$. *Solution 2:* If it is desired to store the base addresses with $I_1 = I_2 = 0$, we could write LDA X,7:1 where location X contains NOP BASE,2:7. The second solution takes one more cycle, but allows the base table to be used with any index registers.

(c) This is equivalent to "LD4 X(0:2)", and takes the same execution time.

4. (i) NOP *,7. (ii) LDA X,7:7. (iii) This is impossible, because 7:7 has been disallowed in an indirect location; see Ex. 5. (iv) LDA X,7:1 with the auxiliary constants

```
X   NOP   *+1,7:2
    NOP   *+1,7:3
    NOP   *+1,7:4
    NOP   0,5:6
```

Execution time is 6 units. (v) INC6 X,7:6 where X contains NOP 0,6:6.

5. (a) Consider the instruction ENTA 1000,7:7 with the memory configuration

location	ADDRESS	I_1	I_2
1000:	1001	7	7
1001:	1004	7	1
1002:	1002	2	2
1003:	1001	1	1
1004:	1005	1	7
1005:	1006	1	7
1006:	1008	7	7
1007:	1002	7	1
1008:	1003	7	2

and with rI1 = 1, rI2 = 2. We find that 1000,7,7 = 1001,7,7,7 = 1004,7,1,7,7 = 1005,1,7,1,7,7 = 1006,7,1,7,7 = 1008,7,7,1,7,7 = 1003,7,2,7,1,7,7 = 1001,1,1,2,7,1,7,7 = 1002,1,2,7,1,7,7 = 1003,2,7,1,7,7 = 1005,7,1,7,7 = 1006,1,7,1,7,7 = 1007,7,1,7,7 = 1002,7,1,1,7,7 = 1002,2,2,1,1,7,7 = 1004,2,1,1,7,7 = 1006,1,1,7,7 = 1007,1,7,7 = 1008,7,7 = 1003,7,2,7 = 1001,1,1,2,7 = 1002,1,2,7 = 1003,2,7 = 1005,7 = 1006,1,7 = 1007,7 = 1002,7,1 = 1002,2,2,1 = 1004,2,1 = 1006,1 = 1007. (A perhaps faster way to do this derivation by hand would be to evaluate successively the addresses specified in locations 1002, 1003, 1007, 1008, 1005, 1006, 1004, 1001, 1000 in this order, but it would seem that a computer would need to go about the evaluation essentially as shown.) The author tried out several fancy schemes for changing the contents of

memory while evaluating the address, yet designed so that everything would be restored again by the time the final address has been obtained. Similar algorithms appear in Section 2.3.5. However, these attempts were unfruitful and it appears there is just not enough room to store the necessary information.

(b, c) Let H, C be auxiliary registers and let N be a counter. To get the effective address M, for the instruction in location L, do the following:

A1. [Initialize.] Set $H \leftarrow 0$, $C \leftarrow L$, $N \leftarrow 0$. (C will be the "current" location, H is used to add together the contents of various index registers, and N measures the "depth" of indirect addressing.)

A2. [Examine address.] Set $M \leftarrow \text{ADDRESS}(C)$. If $I_1(C) = j$, $1 \le j \le 6$, set $M \leftarrow M + rIj$. If $I_2(C) = j$, $1 \le j \le 6$, set $H \leftarrow H + rIj$. If $I_1(C) = I_2(C) = 7$, set $N \leftarrow N + 1$, $H \leftarrow 0$.

A3. [Indirect?] If either $I_1(C)$ or $I_2(C)$ equals 7, set $C \leftarrow M$ and go to A2. Otherwise set $M \leftarrow M + H$, $H \leftarrow 0$.

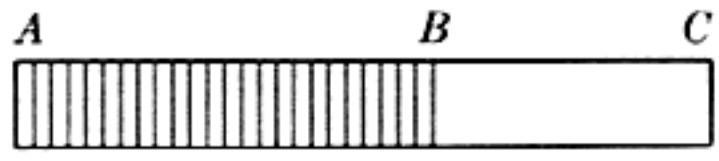
A4. [Reduce depth.] If $N > 0$, set $C \leftarrow M$, $N \leftarrow N - 1$, and go to A2. Otherwise M is the desired answer. ▮

This algorithm will handle any situation correctly except those in which $I_1 = 7$ and $1 \le I_2 \le 6$ *and* the evaluation of the address in ADDRESS involves a case with $I_1 = I_2 = 7$. The effect is as if I_2 were zero. To understand the operation of algorithm A, consider the notation of part (a); the state "L,7,7,7,7,7,1,2,3,5" is represented in the above algorithm by C or M = L, N = 5 (the number of 7's), and H = (rI1) + (rI2) + (rI3) + (rI5) (the post-indexing). In a solution to part (b) of this exercise, the counter N will always be either 0 or 1.

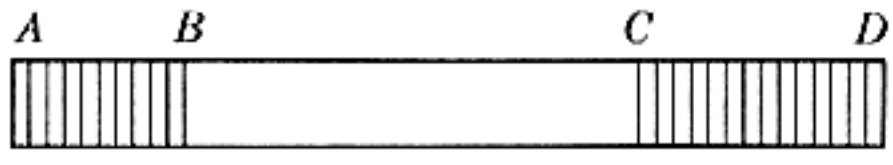
6. (c) causes OVERFLOW. (e) causes UNDERFLOW, and if the program resumes it causes OVERFLOW on the final I_2.

7. No, since TOP[i] must be greater than OLDTOP[i].

8. With a stack, the useful information appears at one end with the vacant information at the other:



where A = BASE[j], B = TOP[j], C = BASE[$j + 1$]. With a queue or deque, the useful information appears at the ends with the vacant information somewhere in the middle:



or in the middle with the vacant information at the ends:

where A = BASE[j], B = REAR[j], C = FRONT[j], D = BASE[$j + 1$]. The two cases are distinguished by the conditions $B \le C$, $B \ge C$, respectively. The algorithms are therefore to be modified in an obvious way so as to widen or narrow the gaps of vacant

information. (Thus in case of overflow, i.e. when $B = C$, we make empty space between B and C by moving one part and not the other.)

9. Given any sequence specification $a_1, a_2, \ldots, a_m$ there is one move operation required for every pair (j, k) such that $j < k$ and $a_j > a_k$. The number of such pairs is therefore the number of moves required. Now imagine all n^m specifications written out, and for each of the $\binom{m}{2}$ pairs (j, k) with $j < k$ count how many specifications have $a_j > a_k$. Clearly this is $\binom{n}{2}$, the number of choices for a_j and a_k, times n^{m-2}, the number of ways to fill in the remaining places. Hence the total number of moves among all specifications is $\binom{m}{2}\binom{n}{2}n^{m-2}$. Divide this by n^m to get the average, Eq. (12).

10. As in Ex. 9 we find the expected value is

$$\binom{m}{2} \sum_{1 \le j < k \le n} p_j p_k = \frac{1}{2}\binom{m}{2}((p_1 + \cdots + p_n)^2 - (p_1^2 + \cdots + p_n^2))$$
$$= \frac{1}{2}\binom{m}{2}(1 - (p_1^2 + \cdots + p_n^2)).$$

For this model, it makes *absolutely no difference* what the relative order of the lists is! (A moment's reflection explains why; if we consider all possible permutations of a given sequence $a_1, \ldots, a_m$ we find the total number of moves summed over all these permutations depends only on the number of pairs of distinct elements $a_j \neq a_k$.)

11. Counting as before, we find the expected number is

$$\frac{1}{n^m}\binom{n}{2} \sum_{0 \le s < m} \sum_{r \ge t} \binom{s}{r}(n-1)^{s-r} n^{m-s-2}(m-s-1).$$

Here s respresents $j - 1$ in the terminology of the above answer, and r is the number of entries in $a_1, a_2, \ldots, a_s$ which equal a_j. This formula can be simplified, e.g. by writing generating functions which correspond to it, until we arrive at Eq. (13). Is there a simpler way yet to give the answer? Apparently not, since the generating function is

$$\sum_m E_{tm} z^m = \binom{n}{2} \frac{1}{(1 - nz)^3} \left(\frac{z}{1 - (n-1)z}\right)^t.$$

12. If $m = 2k$, the average is 2^{-2k} times

$$\binom{2k}{0} 2k + \binom{2k}{1}(2k-1) + \cdots + \binom{2k}{k} k + \binom{2k}{k+1}(k+1) + \cdots + \binom{2k}{2k} 2k.$$

The latter sum is

$$\binom{2k}{k} k + 2\left(\binom{2k-1}{k} 2k + \cdots + \binom{2k-1}{2k-1} 2k\right) = \binom{2k}{k} k + 4k \cdot \tfrac{1}{2} \cdot 2^{2k-1}.$$

A similar argument may be used when $m = 2k + 1$. The answer is

$$\frac{m}{2} + \frac{m}{2^{m+1}}\binom{m}{m/2}$$

when m is even,

$$\frac{m}{2}+\frac{m}{2^m}\binom{m-1}{(m-1)/2}$$

when m is odd.

14. Let $k_j = n/m + \sqrt{n}\, x_j$. (This idea was suggested by N. G. de Bruijn.) Stirling's approximation implies that

$$n^{-m}\frac{m!}{k_1!\ldots k_n!}\max(k_1,\ldots k_n)$$
$$= \sqrt{2\pi}^{1-n} n^{n/2}\left(\frac{m}{n}+\sqrt{m}\max(x_1,\ldots,x_n)\right)$$
$$\times \exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)\sqrt{m}^{1-m}\left(1+O\left(\frac{1}{\sqrt{m}}\right)\right),$$

when $k_1+\cdots+k_n = m$ and when the x's are uniformly bounded. The sum of the latter quantity over all nonnegative $k_1,\ldots,k_n$ satisfying this condition is an approximation to a Riemann integral; we may deduce that the asymptotic behavior of the sum is $a_n^{\prime}(m/n) + c_n\sqrt{m} + O(1)$, where

$$a_n = \sqrt{2\pi}^{1-n} n^{n/2}\int_{x_1+\ldots+x_n=0}\exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)dx_2\ldots dx_n,$$

$$c_n = \sqrt{2\pi}^{1-n} n^{n/2}\int_{x_1+\ldots+x_n=0}\max(x_1,\ldots,x_n)$$
$$\times\exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)dx_2\ldots dx_n,$$

since it is possible to show that the corresponding sums come within ϵ of a_n and c_n for any ϵ.

We know that $a_n = 1$, since the corresponding sum can be evaluated explicitly. The integral which appears in the expression for c_n equals nI_1, where

$$I_1 = \int_{\substack{x_1+\ldots+x_n=0\\ x_1\geq x_2,\ldots,x_n}} x_1\exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)dx_2\ldots dx_n.$$

We may make the substitution

$$x_1 = \frac{1}{n}(y_2+\cdots+y_n),\quad x_2 = x_1 - y_2,\quad x_3 = x_1 - y_3,\quad \ldots,\quad x_n = x_1 - y_n;$$

then we find $I_1 = I_2/n$, where

$$I_2 = \int_{y_2,\ldots,y_n\geq 0}(y_2+\cdots+y_n)\exp\left(-\frac{Q}{2}\right)dy_2\ldots dy_n,$$

$Q = n(y_2^2+\cdots+y_n^2) - (y_2+\cdots+y_n)^2$. Now by symmetry, I_2 is $(n-1)$ times

the same integral with $(y_2 + \cdots + y_n)$ replaced by y_2; hence $I_2 = (n-1)I_3$, where

$$I_3 = \int_{y_2,\ldots,y_n \geq 0} (ny_2 - (y_2 + \cdots + y_n)) \exp\left(-\frac{Q}{2}\right) dy_2 \ldots dy_n$$
$$= \int_{y_3,\ldots,y_n \geq 0} \exp\left(-\frac{Q_0}{2}\right) dy_3 \ldots dy_n;$$

here Q_0 is Q with y_2 replaced by zero. [When $n = 2$, let $I_3 = 1$.] Now let $z_j = \sqrt{n}\, y_j - (y_3 + \cdots + y_n)/(\sqrt{2} + \sqrt{n})$, $3 \leq j \leq m$. Then $Q_0 = z_3^2 + \cdots + z_n^2$, and we deduce that $I_3 = I_4/n^{(n-3)/2}\sqrt{2}$, where

$$I_4 = \int_{y_3,\ldots,y_n \geq 0} \exp\left(-\frac{z_3^2 + \cdots + z_n^2}{2}\right) dz_3 \ldots dz_n$$
$$= \alpha_n \int \exp\left(-\frac{z_3^2 + \cdots + z_n^2}{2}\right) dz_3 \ldots dz_n$$
$$= \alpha_n (\sqrt{2\pi})^{n-2},$$

where α_n is the "solid angle" in $(n-2)$-dimensional space which is spanned by the vectors $(n + \sqrt{2n}, 0, \ldots, 0) - (1, 1, \ldots, 1), \ldots, (0, 0, \ldots, n + \sqrt{2n}) - (1, 1, \ldots, 1)$, divided by the total solid angle of the whole space. Hence

$$c_n = \frac{(n-1)\sqrt{n}}{2\sqrt{\pi}} \alpha_n.$$

We have

$$\alpha_2 = 1, \quad \alpha_3 = \tfrac{1}{2}, \quad \alpha_4 = \frac{1}{\pi} \arctan \sqrt{2} \approx .304,$$

and

$$\alpha_5 = \frac{1}{8} + \frac{3}{4\pi} \arctan \frac{1}{\sqrt{8}} \approx .206.$$

[The value of c_3 was found by Robert M. Kozelka, *Annals of Math. Stat.* **27** (1956), 507–512, but the solution to this problem for higher values of n apparently has never appeared in the literature.]

17. Not unless the queues meet the restrictions which apply to the primitive method (4), (5).

SECTION 2.2.3

1. OVERFLOW is implicit in the operation P ⇐ AVAIL.

2.

```
INSERT  STJ  1F          Store location of "NOP T".
        STJ  9F          Store exit location.
        LD1  AVAIL       rI1 ⇐ AVAIL.
        J1Z  OVERFLOW
        LD3  0,1(LINK)
```

```
        ST3   AVAIL
        STA   0,1(INFO)     INFO(rI1) ← Y.
1H      LD3   *(0:2)        rI3 ← LOC(T).
        LD2   0,3           rI2 ← T.
        ST2   0,1(LINK)     LINK(rI1) ← T.
        ST1   0,3           T ← rI1.
9H      JMP   *                  ▮
```

3.
```
DELETE  STJ   1F            Store location of "NOP T".
        STJ   9F            Store exit location.
1H      LD2   *(0:2)        rI2 ← LOC(T).
        LD3   0,2           rI3 ← T.
        J3Z   9F            Is T = Λ?
        LD1   0,3(LINK)     rI1 ← LINK(T).
        ST1   0,2           T ← rI1.
        LDA   0,3(INFO)     rA ← INFO(rI1).
        LD2   AVAIL         AVAIL ⇐ rI3.
        ST2   0,3(LINK)
        ST3   AVAIL
        ENT3  2             Prepare for second exit.
9H      JMP   *,3                ▮
```

4.
```
OVERFLOW STJ   9F           Store setting of rJ.
         ST1   8F(0:2)      Save rI1 setting.
         LD1   POOLMAX
         INC1  C
         ST1   POOLMAX      Increment POOLMAX.
         CMP1  SEQMIN
         JGE   TOOBAD       Has storage been exceeded?
         ST1   AVAIL        Set AVAIL to new location.
         STZ   0,1(LINK)    Set LINK(AVAIL).
9H       ENT1  *            Take rJ setting.
         DEC1  2            Subtract 2.
         ST1   *+2(0:2)     Store exit location.
8H       ENT1  *            Restore rI1.
         JMP   *            Return. ▮
```

5. Inserting at the front is essentially like the basic insertion operation (8), with an additional test for empty queue: P ⇐ AVAIL, INFO(P) ← Y, LINK(P) ← F, if F = Λ then R ← P, F ← P.

To delete from the rear, we would have to find which node links to NODE(R), and that is necessarily inefficient since we have to search all the way from F. This could be done, for example, as follows:

a) If F = Λ then UNDERFLOW, otherwise set P ← LOC(F).
b) if LINK(P) ≠ R then set P ← LINK(P) and repeat this step until LINK(P) = R.
c) Set Y ← INFO(R), AVAIL ⇐ R, R ← P, LINK(P) ← Λ.

6. We could remove the operation LINK(P) ← Λ from (14), if we delete the commands "F ← LINK(P)" and "if F = Λ then set R ← LOC(F)" from (17); the latter are to be replaced by "if F = R then F ← Λ and R ← LOC(F), otherwise set F ← LINK(P)".

The effect of these changes is that the LINK field of the rear node in the queue will contain spurious information which is never interrogated by the program. A trick like this saves execution time and it is quite useful in practice, although it violates one of the basic assumptions of garbage collection (see Section 2.3.5) so it cannot be used in conjunction with such algorithms.

7. (Make sure your solution works for empty lists.)

I1. Set P ← FIRST, Q ← Λ.
I2. If P ≠ Λ, set R ← Q, Q ← P, P ← LINK(Q), LINK(Q) ← R, and repeat this step.
I3. Set FIRST ← Q. ▮

8.

```
        LD1    FIRST        1   P ≡ rI1 ← FIRST.
        ENT2   0            1   Q ≡ rI2 ← Λ.
        J1Z    2F           1   If list is empty, jump.
1H      ENTA   0,2          n   R ≡ rA ← Q.
        ENT2   0,1          n   Q ← P.
        LD1    0,2(LINK)    n   P ← LINK(Q).
        STA    0,2(LINK)    n   LINK(Q) ← R.
        J1NZ   1B           n   Is P ≠ Λ?
2H      ST2    FIRST        1   FIRST ← Q. ▮
```

The time is $(7n + 6)u$.

9. (a) Yes. (b) Yes if true parenthood is considered; no if legal parenthood is considered (a man's daughter might marry his father, as in the song "I'm My Own Grampa"). (c) No ($-1 \prec 1$ and $1 \prec -1$). (d) Let us hope so, or else there is a circular argument. (e) $1 \prec 3$ and $3 \prec 1$. (f) The statement is ambiguous. If we take the position that the subroutines called by y are dependent upon which subroutine calls y, we would have to conclude that the transitive law does not hold. (For example, a general input/output subroutine might call on different processing routines for each I/O device present, but usually not all these processing subroutines are needed in a single program. This is a problem that plagues many automatic programming systems.)

10. For (i) there are three cases: $x = y$; $x \subset y$ and $y = z$; $x \subset y$ and $y \subset z$. For (ii) there are two cases: $x = y$; $x \neq y$. Each of these subcases is handled trivially, as is (iii).

11. "Multiply out" the following to get all 52 solutions: $13749(25 + 52)86 + (1379 + 1397 + 1937 + 9137)(4258 + 4528 + 2458 + 5428 + 2548 + 5248 + 2584 + 5284)6 + (1392 + 1932 + 1923 + 9123 + 9132 + 9213)7(458 + 548 + 584)6$.

12. For example, (a) list all sets with k elements (in any order) before all sets with $k + 1$ elements, $0 \leq k < n$. (b) Represent a subset by a sequence of 0's and 1's showing which elements are in the set. This gives a correspondence between all subsets and (via the binary number system) the integers 0 through $2^n - 1$. The order of correspondence is a topological sequence.

14. If $a_1\, a_2 \ldots a_n$ and $b_1\, b_2 \ldots b_n$ are two possible topological sorts, let j be minimal such that $a_j \neq b_j$; then $a_k = b_j$ and $a_j = b_m$ for some $k, m > j$. Now $b_j \not\leq a_j$ since $k > j$, and $a_j \not\leq b_j$ since $m > j$, hence (iv) fails. Conversely if there is only one topological sort $a_1\, a_2 \ldots a_n$, we must have $a_j \leq a_{j+1}$ for $1 \leq j < n$, since otherwise a_j and a_{j+1} could be interchanged. This and transitivity imply (iv).

Note: The following alternative proofs work also for infinite sets. (a) Every partial ordering can be embedded in a linear ordering. For if we have two elements

with $x_0 \not\leq y_0$ and $y_0 \not\leq x_0$ we can generate another partial ordering by the rule "$x \leq y$ or $x \leq x_0$ and $y_0 \leq y$". The latter ordering "includes" the former and has $x_0 \leq y_0$. Now apply Zorn's lemma or transfinite induction in the usual way to complete the proof. (b) Obviously a linear ordering cannot be embedded in any different linear ordering. (c) A partial ordering which has incomparable elements x_0 and y_0 as in (a) can be extended to two linear orderings in which $x_0 \leq y_0$ and $y_0 \leq x_0$, so at least two linear orderings exist.

15. If S is finite, we can list all relations $a \prec b$ that are true in the given partial ordering. By successively removing, one at a time, any relations that are implied by others, we arrive at an irredundant set. The problem is to show there is just one such set, no matter in what order we go about removing redundant relations. If there were two irredundant sets α and β, in which "$a \prec b$" appears in α but not in β, there are $k+1$ relations $a \prec c_1 \prec \cdots \prec c_k \prec b$ in β for some $k \geq 1$. But it is possible to deduce $a \prec c_1$ and $c_1 \prec b$ from α, *without* using the relation $a \prec b$ (since $b \not\leq c_1$ and $c_1 \not\leq a$), hence the relation $a \prec b$ is redundant in α.

The result is false for infinite sets S, when there is *at most* one irredundant set of relations. For example if S denotes the integers plus the element ∞ and we define $n \prec n+1$ and $n \prec \infty$ for all n, there is no irredundant set of relations which characterizes this partial ordering.

16. Let S be topologically sorted $x_{p_1} x_{p_2} \ldots x_{p_n}$ and apply this permutation to both rows and columns.

17. If k increases from 1 to n in step T4, the output is 1932745860. If k decreases from n to 1 in step T4, as it does in Program T, the output is 9123745860.

18. They link together the items in sorted order: `QLINK[0]` is the first, `QLINK[QLINK[0]]` is the second, and so on; `QLINK[last] = 0`.

19. This would fail in certain cases; when the queue contains only one element in step T5, this would set `F = 0` (thereby emptying the queue), but other entries could be placed in the queue in step T6. This modification would therefore require an additional test of `F = 0` in step T6.

20. At present, items are output only when they come to the front of the queue. A stack *could* be used, in the following way:

Step T4. Set `T` $\leftarrow 0$. For $1 \leq k \leq n$ if `COUNT[`k`]` is zero do the following: Output k, set `SLINK[`k`]` $\leftarrow$ `T`, `T` $\leftarrow k$. (`SLINK[`k`]` $\equiv$ `QLINK[`k`]`.)

Step T5. If `T = 0`, go to T8; otherwise set `P` $\leftarrow$ `TOP[T]`, `T` $\leftarrow$ `SLINK[T]`.

Step T6. Same as before, except when `COUNT[SUC(P)]` goes down to zero we output `SUC(P)`, set `SLINK[SUC(P)]` $\leftarrow$ `T` and `T` $\leftarrow$ `SUC(P)`.

Step T7. Go to T5.

Step T8. Output "0" and proceed as before. ▮

To compare these two algorithms, they are essentially the same except the output now occurs in three places, not in just one. This is less convenient when the machine implementation is considered, since the output is being packed into a buffer.

21. Repeated relations only make the algorithm a little slower and take up more space in the storage pool. A relation "$j < j$" would be treated like a loop (e.g. an arrow from a box to itself in the corresponding diagram).

22. To make the program "fail-safe" we should (a) check that $0 < n <$ (some appropriate maximum); (b) check each relation $j \prec k$ for the conditions $0 < j, k \leq n$; (c) make sure the number of relations doesn't overflow the storage pool area.

23. At the end of step T5, add "TOP[F] ← Λ". (Then at all times TOP[1], . . . , TOP[n] point to all the relations not yet cancelled.) In step T8, if N > 0, print "LOOP DETECTED IN INPUT:", and set QLINK[k] ← 0 for $1 \leq k \leq n$. Now add the following steps:

T9. For $1 \leq k \leq n$ set P ← TOP[k], TOP[k] ← 0, and perform step T10. (This will set QLINK[j] to one of the predecessors of object j, for each j not yet output.) Then go to T11.

T10. If P ≠ Λ, and QLINK[SUC(P)] = 0, set QLINK[SUC(P)] ← k. If P ≠ Λ set P ← NEXT(P) and repeat this step.

T11. Find a k with QLINK[k] ≠ 0.

T12. Set TOP[k] ← 1 and k ← QLINK[k]. Now if TOP[k] = 0, repeat this step.

T13. (We have found the start of a loop.) Print the value of k, set TOP[k] ← 0, k ← QLINK[k], and if TOP[k] = 1 repeat this step.

T14. Print the value of k (the beginning and end of the loop) and stop. (*Note:* The loop has been printed backwards; if it is desired to print the loop in forward order, an algorithm like that in Ex. 7 should be used between steps T12 and T13.) ▮

24. Insert three lines in the program of the text:

```
08a   PRINTER  EQU   18
14a            ST6   N
60a            STZ   X,1(TOP)          TOP[F] ← Λ.
```

Replace lines 75–77 by the following:

```
75             J6Z   DONE
76             OUT   LINE1(PRINTER)    Print indication of loop.
77             LD6   N
78             STZ   X,6(QLINK)        QLINK[k] ← 0.
79             DEC6  1
80             J6P   *-2               n ≥ k ≥ 1.
81             LD6   N
82     T9      LD2   X,6(TOP)          P ← TOP[k].
83             STZ   X,6(TOP)          TOP[k] ← 0.
84             J2Z   T9A               Is P = Λ?
85     T10     LD1   0,2(SUC)          rI1 ← SUC(P).
86             LDA   X,1(QLINK)
87             JANZ  *+2               If QLINK[rI1] = 0,
88             ST6   X,1(QLINK)          set it to k.
89             LD2   0,2(NEXT)         P ← NEXT(P).
90             J2P   T10               Is P ≠ Λ?
91     T9A     DEC6  1
92             J6P   T9                n ≥ k ≥ 1.
93     T11     INC6  1
94             LDA   X,6(QLINK)
95             JAZ   *-2               Find k with QLINK[k] ≠ 0.
```

```
96              ENTX  1
97      T12     STX   X,6(TOP)          TOP[k] ← 1.
98              LD6   X,6(QLINK)        k ← QLINK[k].
99              LD1   X,6(TOP)
100             J1Z   T12               Is TOP[k] = 0?
101     T13     ENTA  0,6
102             CHAR                    Convert k to alpha.
103             JBUS  *(PRINTER)
104             STX   VALUE             Print.
105             OUT   LINE2(PRINTER)
106             J1Z   DONE              Stop when TOP[k] = 0.
107             STZ   X,6(TOP)          TOP[k] ← 0.
108             LD6   X,6(QLINK)        k ← QLINK[k].
109             LD1   X,6(TOP)
110             JMP   T13
111     LINE1   ALF   LOOP              Title line
112             ALF   DETEC
113             ALF   TED I
114             ALF   N INP
115             ALF   UT:
116     LINE2   ALF                     Succeeding lines
117     VALUE   EQU   LINE2+3
118             ORIG  LINE2+24
119     DONE    HLT                     End of computation.
120     N       CON   0
121     X       EQU   *
122             END   TOPSORT           ▌
```

Note: When the relations $10 \prec 1$, $6 \prec 10$, $1 \prec 9$ were added before the data (18), this program printed out "1,10,6,8,5,9,1" as the loop.

26. One solution is to proceed in two phases as follows:

Phase 1. (We use the X table as a (sequential) stack as we mark B = 1 or 2 for each subroutine that needs to be used.)

A0. For $1 \le$ J $\le$ N set B(X[J]) ← B(X[J]) + 2, if B(X[J]) $\le$ 0.
A1. If N = 0, go to phase 2; otherwise P ← X[N] and decrease N by 1.
A2. If |B(P)| = 1, go to A1, otherwise set P ← P + 1.
A3. If B(SUB1(P)) $\le$ 0, set N ← N + 1, B(SUB1(P)) ← B(SUB1(P)) + 2, X[N] ← SUB1(P). If SUB2(P) $\neq$ 0 and B(SUB2(P)) $\le$ 0, do a similar set of actions with SUB2(P). Go to A2. ▌

Phase 2. (We go through the table and allocate memory.)

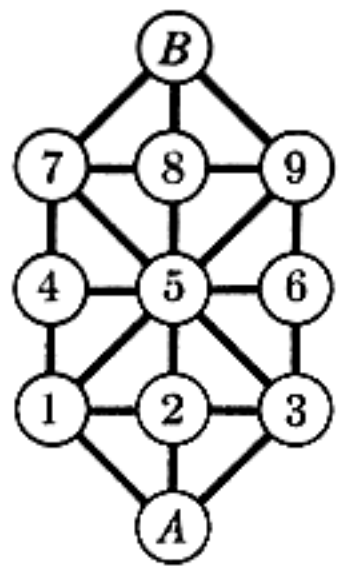
B1. Set P ← FIRST.
B2. If P = Λ, set N ← N + 1, BASE(LOC(X[N])) ← −MLOC, SUB(LOC(X[N])) ← 0, and terminate the algorithm.
B3. If B(P) > 0, set N ← N + 1, BASE(LOC(X[N])) ← MLOC, SUB(LOC(X[N])) ← P, MLOC ← MLOC + SPACE(P).
B4. Set P ← LINK(P) and return to B2. ▌

27. Comments on the following code are left to the reader.

```
B       EQU   0:1
SPACE   EQU   2:3
LINK    EQU   4:5
SUB1    EQU   2:3
SUB2    EQU   4:5
BASE    EQU   0:3
SUB     EQU   4:5
A0      LD2   N
        J2Z   B1
1H      LD3   X,2
        LDA   0,3(B)
        JAP   *+3
        INCA  2
        STA   0,3(B)
        DEC2  1
        J2P   1B
        LD1   N
A1      J1Z   B1
        LD2   X,1
        DEC1  1
A2      LDA   0,2(1:1)
        DECA  1
        JAZ   A1
        INC2  1
A3      LD3   0,2(SUB1)
        LDA   0,3(B)
        JAP   9F
        INC1  1
        INCA  2
        STA   0,3(B)
        ST3   X,1
9H      LD3   0,2(SUB2)
        J3Z   A2
        LDA   0,3(B)
        JAP   A2
        INC1  1
        INCA  2
        STA   0,3(B)
        ST3   X,1
        JMP   A2
B1      ENT2  FIRST
        LDA   MLOC
        JMP   1F
B3      LDX   0,2(B)
        JXNP  B4
        INC1  1
        ST2   X,1(SUB)
```

```
        ADD    0,2(SPACE)
1H      STA    X+1,1(BASE)
B4      LD2    0,2(LINK)
B2      J2NZ   B3
        ENNX   1
        STX    X+1,1(0:0)
        STZ    X+1,1(SUB)    ▮
```

28. We give here only a few comments related to the military game. Let A be the player with three men whose pieces start on nodes A13; let B be the other player. In this game, A must "trap" B, and if B can cause a position to be repeated for a second time we can consider him the winner. To avoid keeping the entire past history of the game as an integral part of the positions, however, we should modify the algorithm in the following way: Start by marking the positions 157–4, 789–B, 359–6 with B to move as "lost" and apply the suggested algorithm. Now the idea is for player A to move only to B's "lost" positions. But he must also take additional precautions against repeating prior moves. A "good" computer game-playing program will use a random number generator to select between several winning moves when more than one is present, so an obvious technique would be to have the computer playing A just choose randomly among those moves which take him to a "lost" position for B.



Board for "The Military Game."

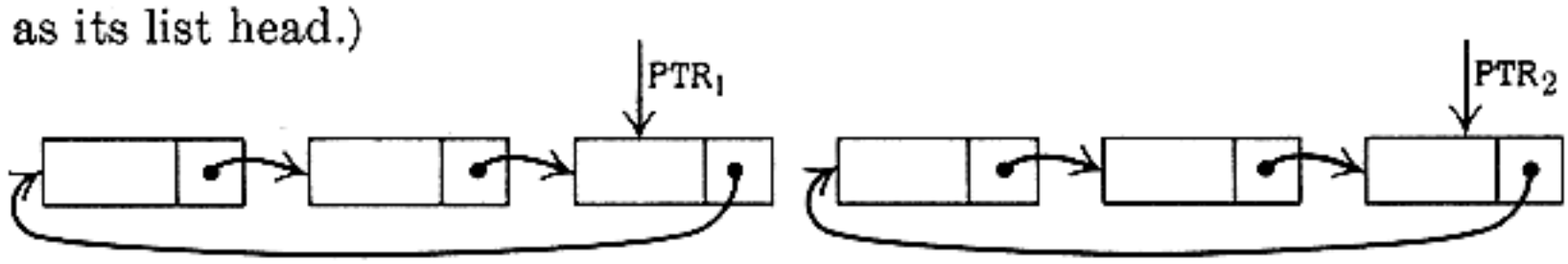
But there are interesting situations which make this plausible procedure fail! For example, consider position 258–7 with A to move; this is a "won" position. From this position player A might try moving to 158–7 (which is a "lost" position for B, according to the algorithm). But then B plays to 158-B, and this forces A to play to 258–B, after which B plays back to 258–7; he has won, since the former position has been repeated! This example shows that the algorithm must be re-invoked after every move has been made, starting with each position that has previously occurred marked "lost" (if A is to move) or "won" (if B is to move).

The author has found that this game makes a very satisfactory computer demonstration program.

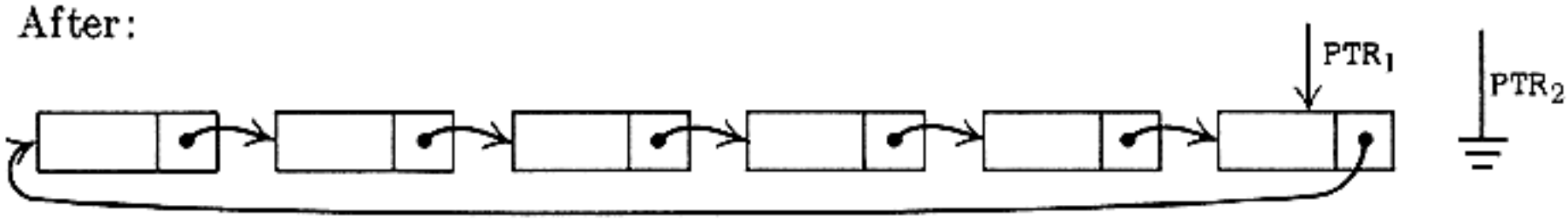
29. (a) If FIRST = Λ, do nothing; otherwise set P $\leftarrow$ FIRST, and then repeatedly set P $\leftarrow$ LINK(P) zero or more times until LINK(P) = Λ. Finally set LINK(P) $\leftarrow$ AVAIL and AVAIL $\leftarrow$ FIRST (and probably also FIRST $\leftarrow \Lambda$). (b) If F = Λ, do nothing; otherwise set LINK(R) $\leftarrow$ AVAIL and AVAIL $\leftarrow$ F (and probably also F $\leftarrow \Lambda$, R $\leftarrow$ LOC(F)).

1. No it does not help, it seems to hinder (if anything). (The stated convention is *not* especially consistent with the circular list philosophy, unless we put NODE(LOC(PTR)) into the list as its list head.)

2. Before:



After:



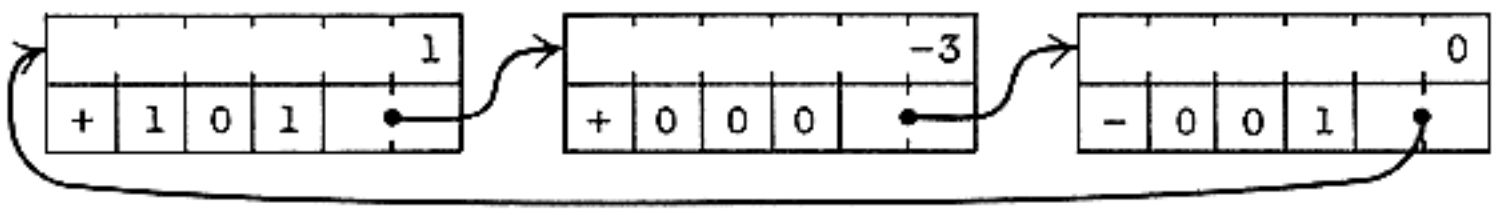
3. If $PTR_1 = PTR_2$, the only effect is $PTR_2 \leftarrow \Lambda$. If $PTR_1 \neq PTR_2$, the exchange of links breaks the list into two parts, as if a circle had been broken in two by cutting at two points; the second part of the operation then makes PTR_1 point to a circular list that consists of the nodes that would have been traversed if, in the original list, we followed the links from PTR_1 to PTR_2.

4. Let HEAD be the address of the list head. To push down Y onto the stack: set P ⇐ AVAIL, INFO(P) ← Y, LINK(P) ← LINK(HEAD), LINK(HEAD) ← P. To pop up the stack onto Y: if LINK(HEAD) = HEAD then UNDERFLOW, otherwise set P ← LINK(HEAD), LINK(HEAD) ← LINK(P), Y ← INFO(P), AVAIL ⇐ P.

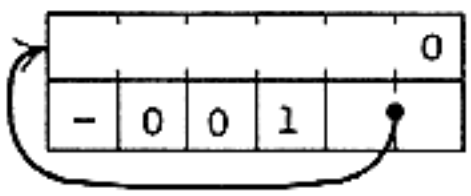
5. If PTR ≠ Λ, start with P ← PTR, Q ← LINK(PTR); then set R ← LINK(Q), LINK(Q) ← P, P ← Q, Q ← R, and repeat this operation until P = PTR.

6.

a)



b)



7. Matching terms in the polynomial are located in one pass over the list, instead of requiring repeated random searches. Also, *increasing* order would be incompatible with the "−1" sentinel.

8. We must know what node points to the current node of interest, if we are going to delete that node or to insert another one ahead of it. There are alternatives, however: we could delete NODE(Q) by setting Q2 ← LINK(Q) and then setting NODE(Q) ← NODE(Q2), AVAIL ⇐ Q2; we could insert a NODE(Q2) in front of NODE(Q) by first interchanging NODE(Q2) ↔ NODE(Q), then setting LINK(Q) ← Q2, Q ← Q2. These clever tricks allow the deletion and insertion *without* knowing which node links to NODE(Q); they were used in early versions of IPL. But they have the disadvantage that the sentinel node at the end of a polynomial will occasionally move, and other link variables may be pointing to this node.

9. Algorithm A with P = Q simply doubles polynomial(Q), as it should. Algorithm M with P = M also gives the expected result. Algorithm M with P = Q sets polynomial(P) ← polynomial(P) times $(1 + t_1)(1 + t_2) \cdots (1 + t_k)$ if $M = t_1 + t_2 + \cdots + t_k$ (although this is not immediately obvious). When M = Q, Algorithm M surprisingly gives the expected result, polynomial(Q) ← polynomial(Q) + polynomial(Q) × polynomial (P), except that the computation blows up when the constant term of polynomial (P) is −1.

10. No changes at all. The only possible difference would be in step M2, removing error checks that A, B, or C might individually overflow (these error checks have not been specified because we have assumed they are not necessary). In other words, the algorithms in this section may be regarded as operations on $f(x^{b^2}, x^b, x)$ instead of on $f(x, y, z)$.

11.

```
COPY  STJ   9F          (comments are left
      ENT3  9F            to the reader)
      LDA   1,1
1H    LD6   AVAIL
      J6Z   OVERFLOW
      LDX   1,6(LINK)
      STX   AVAIL
      STA   1,6
      LDA   0,1
      STA   0,6
      ST6   1,3(LINK)
      ENT3  0,6
      LD1   1,1(LINK)
      LDA   1,1
      JANN  1B
      LD2   8F
      ST2   1,3(LINK)
9H    JMP   *
8H    CON   0  ▮
```

12. Let the polynomial copied have p terms. Program A takes $(29p + 13)u$, and to make it a fair comparison we should add the time to create a zero polynomial, e.g. $18u$ with Ex. 14. The program of Ex. 11 takes $(21p + 31)u$, about 75% as much.

13.

```
ERASE  STJ   9F
       LDX   AVAIL
       LDA   1,1(LINK)
       STA   AVAIL
       STX   1,1(LINK)
9H     JMP   *          ▮
```

14.

```
ZERO  STJ   9F
      LD1   AVAIL
      J1Z   OVERFLOW
      LDX   1,1(LINK)
      STX   AVAIL
      ENT2  0,1
      MOVE  1F(2)
```

```
                ST2   1,2(LINK)
        9H      JMP   *
        1H      CON   0
                CON   -1(ABC)    ▮
```

15.

```
MULT  STJ   9F          Entrance to subroutine
      LDA   5F          Change settings of switches.
      STA   SW1
      LDA   6F
      STA   SW2
      STA   SW3
      JMP   *+2
2H    JMP   ADD         M2. Multiply cycle.
1H    LD4   1,4(LINK)   M1. Next multiplier. M ← LINK(M).
      LDA   1,4
      JANN  2B          To M2 if ABC(M) ≥ 0.
8H    LDA   7F          Restore settings of switches.
      STA   SW1
      LDA   8F
      STA   SW2
      STA   SW3
9H    JMP   *           Return.
5H    JMP   *+1         New setting of SW1
      LDA   0,1         COEF(P)
      MUL   0,4           × COEF(M) → rX.
      LDA   1,1(ABC)    ABC(P)
      JAN   *+2
      ADD   1,4(ABC)      + ABC(M), if ABC(P) ≥ 0
      SLA   2           Move into 0:3 field of rA.
      STX   0F          Save rX for use in SW2 and SW3.
      JMP   SW1+1
6H    LDA   0F          New setting of SW2, SW3
7H    LDA   1,1         Usual setting of SW1
8H    LDA   0,1         Usual setting of SW2, SW3
0H    CON   0           Temp storage ▮
```

16. Let r be the number of terms in polynomial(M). The subroutine requires $21pr + 38r + 29 + 29\sum m' + 19\sum m'' + 29\sum p' + 8\sum q'$, where the latter summations refer to the corresponding quantities during the r activations of program A. The number of terms in polynomial(Q) goes up by $p' - m'$ each activation of program A. If we make the not unreasonable assumption that $m' = 0$ and $p' = \alpha p$ where $0 < \alpha < 1$, we get the respective sums equal to 0, $(1 - \alpha)pr$, αpr, and $rq_0' + \alpha p(r(r-1)/2)$, where q_0' is the value of q' in the first iteration. The grand total is $4\alpha pr^2 + 40pr + 6\alpha pr + 8q_0'r + 38r + 29$. This analysis indicates that the multiplier ought to have fewer terms than the multiplicand, since we have to skip over unmatching terms in polynomial(Q) more often.

17. There actually is very little advantage; addition and multiplication routines with either type of list would be virtually the same. The efficiency of the ERASE subroutine (see Ex. 13) is apparently the only important difference.

18. Let the link field of node x_i contain $\texttt{LOC}(x_{i+1}) \oplus \texttt{LOC}(x_{i-1})$, where "$\oplus$" denotes either subtraction or "exclusive or." Two adjacent list heads are included in the circular list, to help get things started properly. (The origin of this ingenious technique is unknown.)

SECTION 2.2.5

1. Insert Y at the left: P ⇐ AVAIL; INFO(P) ← Y; LLINK(P) ← Λ; RLINK(P) ← LEFT; if LEFT ≠ Λ then LLINK(LEFT) ← P else RIGHT ← P; LEFT ← P. Set Y to left and delete: if LEFT = Λ then UNDERFLOW; P ← LEFT; LEFT ← RLINK(P); if LEFT = Λ then RIGHT ← Λ else LLINK(LEFT) ← Λ; AVAIL ⇐ P.

2. Consider the case of several deletions (at the same end) in succession. After each deletion we must know what to delete next. This implies the links in the list point away from that end of the list. So deletion at both ends implies the links must go both ways.

3. To show the independence of CALLUP from CALLDOWN, notice for example that in Table 1 the elevator did not stop at floors 2 or 3 at time 0393–0444 although there were people waiting; these people had pushed CALLDOWN, but if they had pushed CALLUP the elevator would have stopped.

To show the independence of CALLCAR from the others, notice in Table 1 when the doors start to open at time 1398 the elevator has already decided to be GOINGUP. Its state would have been NEUTRAL at that point if CALLCAR[1] = CALLCAR[2] = CALLCAR[3] = CALLCAR[4] = 0, according to step E2, but in fact CALLCAR[2] and CALLCAR[3] have been set to 1 by men nos. 7 and 9 in the elevator. (If we envision the same situation with all floor numbers increased by 1, the fact that STATE = NEUTRAL or STATE = GOINGUP when the doors open would affect whether the elevator would perhaps continue to go downward or would unconditionally go upward.)

4. If a dozen or more people were getting out at the same floor, STATE might be NEUTRAL all during this time, and when E9 calls the DECISION subroutine this may set a new state before anyone has gotten in on the current floor. It happens very rarely indeed (and it certainly was the most puzzling phenomenon observed by the author during his elevator experiments).

5. The state from the time the doors start to open at time 1063 until man 7 gets in at time 1183 would have been NEUTRAL, since there would have been no calls to floor 0. Then man 7 would set CALLCAR[2] ← 1 and the state would correspondingly change to GOINGUP.

6. Add the condition "if OUT < IN then STATE ≠ GOINGUP; if OUT > IN then STATE ≠ GOINGDOWN" to the condition "FLOOR = IN" in steps M2 and M4. In step E4, accept men from QUEUE[FLOOR] only if they are headed in the elevator's direction, unless STATE = NEUTRAL (when we accept all comers); men from QUEUE[FLOOR] who have not been accepted should also push CALLUP or CALLDOWN again (since the state can change in step M5).

7. In line 227 this man is assumed to be in the WAIT list. Jumping to M4A makes sure he stays there. It is assumed that GIVEUPTIME is not zero, and indeed that it is probably 100 or more.

8. Comments are left to the reader.

```
277     E8   DEC4   1
278          ENTA   61
279          JMP    HOLDC
280          LDA    CALL,4(3:5)
281          JAP    1F
282          ENT1   -2,4
283          J1Z    2F
284          LDA    CALL,4(1:1)
285          JAZ    E8
286     2H   LDA    CALL-1,4
287          ADD    CALL-2,4
288          ADD    CALL-3,4
289          ADD    CALL-4,4
290          JANZ   E8
291     1H   ENTA   23
292          JMP    E2A
```

9.

```
01   DECISION  STJ    9F                    Store exit location.
02             J5NZ   9F                    D1. Decision necessary?
03             LDX    ELEV1+2(NEXTINST)
04             DECX   E1                    D2. Should door open?
05             JXNZ   1F                    Jump if elevator not at E1.
06             LDA    CALL+2
07             ENT3   E3                    Prepare to schedule E3,
08             JANZ   8F                      if there is a call on floor 2.
09   1H        ENT1   -4                    D3. Any calls?
10             LDA    CALL+4,1              Search for a nonzero call variable.
11             JANZ   2F
12   1H        INC1   1                     rI1 ≡ j − 4
13             J1NP   *-3
14             LDA    9F(0:2)               All CALL[j], j ≠ FLOOR, are zero
15             DECA   E6B                   Is exit location = line 250?
16             JANZ   9F
17             ENT1   -2                    Set j ← 2.
18   2H        ENT5   4,1                   D4. Set STATE.
19             DEC5   0,4                   STATE ← j − FLOOR.
20             J5NZ   *+2
21             JANZ   1B                    j = FLOOR not allowed in general.
22             JXNZ   9F                    D5. Elevator dormant?
23             J5Z    9F                    Jump if not at E1 or if j ≠ 2.
24             ENT3   E6                    Otherwise schedule E6.
25   8H        ENTA   20                    Wait 20 units of time.
26             ST6    TEMPD                 Save rI6.
27             ENT6   ELEV1
28             ST3    2,6(NEXTINST)         Set NEXTINST to E3 or E6.
29             JMP    HOLD                  Schedule the activity.
30             LD6    TEMPD                 Restore rI6.
31   9H        JMP    *                     Exit from subroutine.
```

11. Initially let LINK[k] $= 0$, $1 \le k \le n$, and HEAD $= 0$. During a simulation step that changes V[k], give an error indication if LINK[k] $\neq 0$; otherwise set LINK[k] $\leftarrow$ HEAD, HEAD $\leftarrow k$ and set NEWV[k] to the new value of V[k]. After each simulation step, set $k \leftarrow$ HEAD, HEAD $\leftarrow 0$, and do the following operation repeatedly zero or more times until $k = 0$: set V[k] $\leftarrow$ NEWV[k], $t \leftarrow$ LINK[k], LINK[k] $\leftarrow 0$, $k \leftarrow t$.

Clearly this method is readily adapted to the case of scattered variables, if we include a NEWV and LINK field in each node associated with a variable field V.

12. The WAIT list has deletions from the left to the right, but insertions are sorted in from the right to the left (since the search is likely to be shorter from that side). Also we delete nodes from all three lists in several places when we do not know the predecessor or successor of the node being deleted. Only the ELEVATOR list could be converted to a one-way list, without much loss of efficiency.

Note: It may be preferable to use a non-linear list as the WAIT list in a discrete simulator, to reduce the time for "sorting in". See the "heap sort" algorithms of Chapter 5, for a method that maintains a "smallest in first out" table using only $O(\log n)$ operations when n elements are present. (There is no need for such a fancy method when n is known to be small.) This idea is due to R. W. Floyd.

SECTION 2.2.6

1. (Note that the indices run from 1 to n, not from 0 to n as in Eq. (5).) LOC(A[0, 0]) + $2nJ + 2K$ = LOC(A[J, K]), where A[0, 0] is an assumed node that is actually non-existent. If we set J = K = 1, we get LOC(A[0, 0]) + $2n + 2$ = LOC(A[1, 1]), so the answer can be expressed in several ways. LOC(A[0, 0]) might be negative.

2. LOC(A[$I_1, \ldots, I_k$]) = LOC(A[0, . . . , 0]) + $\sum_{1\le r\le k} a_r I_r$

$$= \text{LOC}(A[l_1, \ldots, l_k]) - \sum_{1\le r\le k} a_r l_r + \sum_{1\le r\le k} a_r I_r,$$

where $a_r = \prod_{r<s\le k} (u_r - l_r + 1)$.

Note: For a generalization to the structures occurring in the COBOL and PL/I languages, and a simple algorithm to compute the relevant constants, see P. Deuel, *CACM* 9 (1966), 344–347.

3. $1 \le k \le j \le n$ if and only if $0 \le k - 1 \le j - 1 \le n - 1$; so replace k, j, n respectively by $k - 1$, $j - 1$, $n - 1$ in all formulas derived for lower bound zero.

4. LOC(A[J, K]) = LOC(A[0, 0]) + $(n + 1)$J − J(J − 1)/2 + K.

5. Let A0 = LOC(A[0, 0]). There are at least two solutions, assuming J is in rI1 and K is in rI2: (1) "LDA TA2,1:7", where location TA2+J is "NOP J+1*J/2+A0,2"; (2) "LDA C1,7:2", where location C1 contains "NOP TA,1:7" and location TA+J says "NOP J+1*J/2+A0". The latter takes one more cycle but doesn't tie the table down to index register 2.

6. (a) LOC(A[I, J, K]) = LOC(A[0, 0, 0]) + $\binom{I+2}{3} + \binom{J+1}{2} + \binom{K}{1}$.

(b) LOC(B[I, J, K]) = LOC(B[0, 0, 0])

$$+\binom{n+3}{3} - \binom{n+3-I}{3} + \binom{n+2-I}{2} - \binom{n+2-I-J}{2} + K;$$

the expansion of this polynomial contains a term in I · J; hence it cannot be expressed in the stated form. A tetrahedral array like B is *not* best stored in lexicographic order.

7. $\mathtt{LOC(A[I_1,\ldots,I_k])} = \mathtt{LOC(A[0,\ldots,0])} + \sum_{1\le r\le k}\binom{\mathtt{I}_r+k-r}{1+k-r}$. See Ex. 1.2.6–56.

8. Let X[I, J, K] be defined for $0 \le \mathtt{I} \le n+1$, $0 \le \mathtt{J} \le n+1$, $0 \le \mathtt{K} \le n+2$. We can let A[I, J, K] = X[I, J, K], B[I, J, K] = X[J, I + 1, K], C[I, J, K] = X[I + 1, K, J + 1], D[I, J, K] = X[K, I + 1, J + 1], E[I, J, K] = X[J, K, I + 1], F[I, J, K] = X[K, J + 1, I + 2]. Clearly each of these definitions fits within the confines of the X array; we must prove there is no overlap, and that is easily done if we rename the indices of X as i, j, k in each of the six definitions; then we find respectively $i \ge j \ge k$, $j > i \ge k$, $i \ge k > j$, $j \ge k > i$, $k > i \ge j$, $k > j > i$ and these conditions prove there is no overlap. However, the above method does not leave the X array filled: the elements X[$n+1$, J, K] and X[J, K, $n+2$] are unused for $0 \le \mathtt{K} \le \mathtt{J} \le n+1$. When $n \ge 2$, it appears to be impossible to pack any tighter than this and still retain linear addressing in each index, although the author has not proved this. Some indices might run negatively. The problem is trivial and of no interest when $n = 0$ or 1.

The above method uses locations X[0, 0, 0] through X[$n+1$, n, $n+1$] minus a few holes in between. The arrays, which contain $(n+1)(n+2)(n+3)$ elements in all, have been packed into $(n+2)(n+2)(n+3) - (n+4)$ consecutive cells. Is there a better solution?

9. **G1.** Set pointer variables P1, P2, P3, P4, P5, P6 to the first locations of the lists FEMALE, A21, A22, A23, BLOND, BLUE, respectively. Assume in what follows that the end of each list is given by link Λ, and Λ is smaller than any other link. If P6 = Λ, stop (the list, unfortunately, is empty).

G2. (Many possible orderings of the following actions could be done; we have chosen to examine EYES first, then HAIR, then AGE, then SEX.) Set P5 ← HAIR(P5) zero or more times until P5 ≤ P6. If now P5 < P6, go to step G5.

G3. Set P4 ← AGE(P4) repeatedly if necessary until P4 ≤ P6. Similarly do the same to P3 and P2 until P3 ≤ P6 and P2 ≤ P6. If now P4, P3, P2 are all smaller than P6, go to G5.

G4. Set P1 ← SEX(P1) until P1 ≤ P6. If P1 = P6, we have found one of the desired girls, so output her address, P6. (Her age can be determined from the settings of P2, P3, and P4.)

G5. Set P6 ← EYES(P6). Now stop if P6 = Λ; otherwise return to G2. ▮

This algorithm is interesting but not the best way to organize a list for such a search.

10. After trying out many different seemingly efficient schemes and analyzing their efficiency, the author feels there seems to be no better way than to divide all people into n approximately equal groups, where n is as large as possible based on the amount of space available, in such a way that a person's characteristics determine the group he is in; then search every person in the appropriate group for the desired characteristics. (See also Chapter 6.)

11. At most $200 + 200 + 3 \cdot 4 \cdot 200 = 2800$ words.

12. VAL(Q0) = c, VAL(P0) = b/a, VAL(P1) = d.

13. It is convenient to have at the end of each list a sentinel which "compares low" in some field on which the list is ordered. A straight one-way list *could* have been used, for example by retaining just the LEFT links in BASEROW[i] and the UP links in

BASECOL[j], by modifying Algorithm S thus: S2, test if P0 = Λ before setting J ← COL(P), and if so set P0 ← LOC(BASEROW[I0]) and go to S3. S3, test if Q0 = Λ and if so, terminate. S4, analogous to changes in S2. S5, test if P1 = Λ and if so treat this as if COL(P1) < 0. S6, test if UP(PTR[J]) = Λ and if so treat as if its ROW field were negative.

These modifications make the algorithm more complicated and save no storage space except a ROW or COL field in the list heads (which in the case of MIX is no saving at all).

14. One could first link together those columns which have a nonzero element in the pivot row, so that all other columns could be skipped as we pivot on each row. Rows in which the pivot column is zero are skipped over immediately.

15. Let rI1 ≡ PIVOT, J; rI2 ≡ P0; rI3 ≡ Q0; rI4 ≡ P; rI5 ≡ P1, X; LOC(BASEROW[i]) ≡ BROW + i; LOC(BASECOL[j]) ≡ BCOL + j; PTR[j] ≡ BCOL + j(1:3).

```
01   ROW        EQU   0:3
02   UP         EQU   4:5
03   COL        EQU   0:3
04   LEFT       EQU   4:5
05   PTR        EQU   1:3
06   PIVOTSTEP  STJ   9F          Subroutine entrance, rI1 = PIVOT
07   S1         LD2   0,1(ROW)
08              ST2   I0          I0 ← ROW(PIVOT).
09              LD3   1,1(COL)
10              ST3   J0          J0 ← COL(PIVOT).
11              LDA   =1.0=       Floating point constant 1
12              FDIV  2,1
13              STA   ALPHA       ALPHA ← 1/VAL(PIVOT).
14              LDA   =1.0=
15              STA   2,1         VAL(PIVOT) ← 1.
16              ENT2  BROW,2      P0 ← LOC(BASEROW([I0]).
17              ENT3  BCOL,3      Q0 ← LOC(BASECOL([J0]).
18              JMP   S2
19   2H         ENTA  BCOL,1
20              STA   BCOL,1(PTR) PTR[J] ← LOC(BASECOL([J]).
21              LDA   2,2
22              FMUL  ALPHA
23              STA   2,2         VAL(P0) ← ALPHA × VAL(P0).
24   S2         LD2   1,2(LEFT)   P0 ← LEFT(P0).
25              LD1   1,2(COL)    J ← COL(P0).
26              J1NN  2B          If J < 0, process J.
27   S3         LD3   0,3(UP)     Q0 ← UP(Q0).
28              LD4   0,3(ROW)    rI4 ← ROW(Q0).
29   9H         J4N   *           If rI4 < 0, exit.
30              CMP4  I0
31              JE    S3          If rI4 = I0, repeat.
32              ST4   I(ROW)      I ← rI4.
33              ENT4  BROW,4      P ← LOC(BASEROW[I]).
34   S4A        LD5   1,4(LEFT)   P1 ← LEFT(P).
35   S4         LD2   1,2(LEFT)   P0 ← LEFT(P0).
```

```
36              LD1    1,2(COL)       J ← COL(P0).
37              CMP1   J0
38              JE     S4             Repeat if J = J0.
39              ENTA   0,1
40              SLA    2              rA(0:3) ← J.
41              J1NN   S5
42              LDAN   2,3            If J < 0,
43              FMUL   ALPHA          set VAL(Q0) ← —ALPHA × VAL(Q0).
44              STA    2,3
45              JMP    S3
46      1H      ENT4   0,5            P ← P1.
47              LD5    1,4(LEFT)      P1 ← LEFT(P).
48      S5      CMPA   1,5(COL)
49              JL     1B             Loop until COL(P1) ≤ J.
50              JE     S7             If =, go right to S7.
51      S6      LD5    BCOL,1(PTR)    rI5 ← PTR[J].
52              LDA    I              rA(0:3) ← I.
53      2H      ENT6   0,5            rI6 ← rI5.
54              LD5    0,6(UP)        rI5 ← UP(rI6).
55              CMPA   1,5(COL)
56              JL     2B             Jump if COL(rI5) ← I.
57              LD5    AVAIL          X ⇐ AVAIL.
58              J5Z    OVERFLOW
59              LDA    0,5(UP)
60              STA    AVAIL
61              LDA    0,6(UP)        UP(PTR[J])
62              STA    0,5(UP)        → UP(X).
63              LDA    1,4(LEFT)      LEFT(P)
64              STA    1,5(LEFT)      → LEFT(X).
65              ST1    1,5(COL)       COL(X) ← J.
66              LDA    I(ROW)
67              STA    0,5(ROW)       ROW(X) ← I.
68              STZ    2,5            VAL(X) ← 0.
69              ST5    1,4(LEFT)      LEFT(P) ← X.
70              ST5    0,6(UP)        UP(PTR[J]) ← X.
71              ST5    BCOL,1(PTR)    PTR[J] ← X ≡ P1.
72      S7      LDAN   2,3            —VAL(Q0)
73              FMUL   2,2              × VAL(P0)
74              FADD   2,5                + VAL(P1)
75              JAZ    S8             If significance lost, to S8.
76              STA    2,5            Otherwise store in VAL(P1).
77              ST5    BCOL,1(PTR)    PTR[J] ← P1.
78              ENT4   0,5            P ← P1.
79              JMP    S4A            P1 ← LEFT(P), to S4.
80      S8      LD6    BCOL,1(PTR)    rI6 ← PTR[J].
81              JMP    *+2
82              LD6    0,6(UP)        rI6 ← UP(rI6).
83              LDA    0,6(UP)
84              DECA   0,5            Is UP(rI6) = P1?
```

```
85      JANZ   *-3          Loop until equal.
86      LDA    0,5(UP)
87      STA    0,6(UP)      UP(rI6) ← UP(P1).
88      LDA    1,5(LEFT)
89      STA    1,4(LEFT)    LEFT(P) ← LEFT(P1).
90      LDA    AVAIL        AVAIL ⇐ P1.
91      STA    0,5(UP)
92      ST5    AVAIL
93      JMP    M4A          P1 ← LEFT(P), to S4. ▮
```

Note: Using the conventions of Chapter 4, lines 72–75 would actually be coded

`LDA 2,3; FMUL 2,2; FCMP 2,5; JE S8; STA TEMP; LDA 2,5; FSUB TEMP;`

with a suitable parameter `EPSILON` in location zero.

18. $k = 1$, pivot column 3, we obtain

$$\begin{pmatrix} \frac{1}{3} & \frac{2}{3} & \frac{1}{3} \\ -\frac{2}{3} & -\frac{1}{3} & -\frac{2}{3} \\ -\frac{1}{3} & -\frac{2}{3} & -\frac{1}{3} \end{pmatrix};$$

$k = 2$, pivot column 1, we obtain

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ -\frac{3}{2} & \frac{1}{2} & 1 \\ -\frac{1}{2} & -\frac{1}{2} & 0 \end{pmatrix};$$

$k = 3$, pivot column 2, we obtain

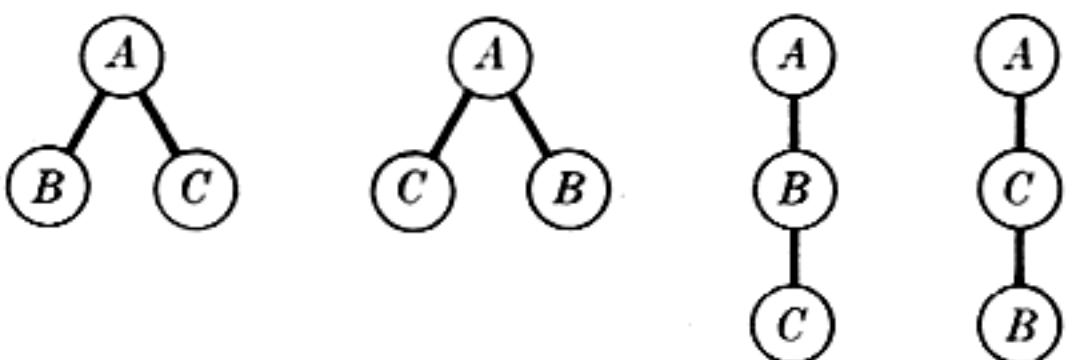
$$\begin{pmatrix} 0 & 1 & 0 \\ -2 & 1 & 1 \\ 1 & -2 & 0 \end{pmatrix};$$

after the final permutations, we have the answer

$$\begin{pmatrix} 1 & -2 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}.$$

SECTION 2.3

1. There are three ways to choose the root. Once the root has been chosen, say A, there are three ways to partition the remaining nodes into subtrees: $\{B\}, \{C\}$; $\{C\}, \{B\}$; $\{B, C\}$. In the latter case there are two ways to make $\{B, C\}$ into a tree, depending on which is the root. Hence get four trees when A is the root:



and 12 in all. This problem is solved for any number of nodes in Section 2.3.4.4.

2. The first two trees in the answer to Ex. 1 are the same, as oriented trees, so we get only 9 different possibilities in this case. For the general solution, see Section 2.3.4.4.

3. Part 1: To show there is *at least one* such sequence. Let the tree have n nodes. The result is clear when $n = 1$, since X must be the root. If $n > 1$, the definition implies there is a root X_1 and subtrees $T_1, T_2, \ldots, T_m$; either $X = X_1$ or X is a member of a unique T_j. In the latter case, there is by induction a path $X_2, \ldots, X$ where X_2 is the root of T_j, and since X_1 is the father of X_2 we have a path $X_1, X_2, \ldots, X$.

Part 2: To show there is *at most one* such sequence. We will prove by induction that if X is not the root of the tree, X has a unique father (so that X_k determines X_{k-1} determines X_{k-2}, etc.). If the tree has one node, there is nothing to prove; otherwise X is in a unique T_j. Either X is the root of T_j, in which case X has a unique father by definition; or X is not the root of T_j, in which case X has a unique father in T_j by induction, and no node outside of T_j can be X's father.

4. True (unfortunately). A similar situation is the popular expression "She is a star of the nth magnitude." According to the way astronomers give magnitudes to stars, a much better compliment would be "She is a star of the *zero*th magnitude."

5. 4.

6. Let $\text{father}^0(X)$ denote X, $\text{father}^1(X)$ denote X's father, $\text{father}^2(X) = \text{father}(\text{father}(X)) = X$'s grandfather, $\text{father}^k(X) = \text{father}(\text{father}^{k-1}(X)) = X$'s "(great)$^{k-2}$-grandfather." The cousinship condition is that $\text{father}^{m+1}(X) = \text{father}^{m+n+1}(Y)$ but $\text{father}^m(X) \neq \text{father}^{m+n}(Y)$; or, if $n > 0$, possibly the same condition with X, Y interchanged.

7. We go to an unsymmetric relation between X and Y; the condition of Ex. 6 is used, with the convention that $\text{father}^j(X) \neq \text{father}^k(Y)$ if either j or k (or both) is -1. To show that this relation is always valid for some unique m and n, consider the Dewey decimal notation for X and Y, namely $1 . a_1 . \cdots . a_p . b_1 . \cdots . b_q$ and $1 . a_1 . \cdots . a_p . c_1 . \cdots . c_r$, where $p \geq 0$, $q \geq 0$, $r \geq 0$ and (if $qr \neq 0$) $b_1 \neq c_1$. The numbers of any pair of nodes can be written in this form, and clearly we must take $m = q - 1$ and $n + m = r - 1$.

8. The empty binary tree (i.e. the binary tree having no nodes). A "tree" always has at least one node, its root, while a "binary tree" may be vacuous.

9. A is the root, since we conventionally put the root at the top.

10. Any *finite* collection of nested sets corresponds to a forest as defined in the text, as follows: Let $A_1, \ldots, A_n$ be the sets of the collection that are contained in no other. For fixed j, the sub-collection of all sets contained in A_j is nested, and therefore we may assume this sub-collection corresponds to a tree (unordered) with A_j as the root.

11. In a nested collection $\mathcal{C}$ let $X \equiv Y$ if there is some $Z \in \mathcal{C}$ such that $X \cup Y \subseteq Z$. This relation is obviously reflexive and symmetric, and it is in fact an equivalence relation since $W \equiv X$, $X \equiv Y$ implies there are Z_1, Z_2 in $\mathcal{C}$ with $W \subseteq Z_1$, $X \subseteq Z_1 \cap Z_2$, $Y \subseteq Z_2$. Since $Z_1 \cap Z_2 \neq \emptyset$, either $Z_1 \subseteq Z_2$ or $Z_2 \subseteq Z_1$, hence $W \cup Y \subseteq Z_1 \cup Z_2 \in \mathcal{C}$. Now if $\mathcal{C}$ is a nested collection, define an oriented forest corresponding to $\mathcal{C}$ by the rule "X is an ancestor of Y, and Y is a descendant of X, if and only if $X \supset Y$." Each equivalence class of $\mathcal{C}$ corresponds to an oriented tree, which is an oriented forest with

$X \equiv Y$ for all X, Y. (We thereby have generalized the definitions of forest and tree which were given for finite collections.) In these terms, we may define the *level* of X as the cardinal number of $(\{X\} \cup \text{ancestors}(X))$. The *degree* of X is the cardinal number of equivalence classes in the nested collection descendants(X). We say X is the *father* of Y, and Y is a *son* of X, if X is an ancestor of Y but there is no Z such that $X \supset Z \supset Y$. To get *ordered* trees and forests, order the equivalence classes mentioned above in some *ad hoc* manner, for example by embedding the relation $\subseteq$ into linear order.

Example (a): Let $S_{\alpha k} = \{x \mid x = .d_1 d_2 d_3 \ldots$ in decimal notation, where $\alpha = .e_1 e_2 e_3 \ldots$ in decimal notation, and $d_j = e_j$ if $j \bmod 2^k \neq 0\}$. The collection $\mathcal{C} = \{S_{\alpha k} \mid k \geq 0, 0 < \alpha < 1\}$ is nested, and gives a tree with infinitely many levels and uncountable degree for each node.

Example (b), (c): It is convenient to define this set in the plane, instead of in terms of real numbers, and this is sufficient since there is a one-to-one correspondence between the plane and the real numbers. Let $S_{\alpha mn} = \{(\alpha, \beta) \mid m/2^n \leq \beta < (m+1)/2^n\}$, and let $T_\alpha = \{(\gamma, \beta) \mid \gamma \leq \alpha\}$. The collection $\mathcal{C} = \{S_{\alpha mn} \mid 0 < \alpha < 1, n \geq 0, 0 \leq m < 2^n\} \cup \{T_\alpha \mid 0 < \alpha < 1\}$ is easily seen to be nested. The sons of $S_{\alpha mn}$ are $S_{\alpha(2m)(n+1)}$ and $S_{\alpha(2m+1)(n+1)}$, and T_α has the son $S_{\alpha 00}$ plus the subtree $\{S_{\gamma mn} \mid \gamma < \alpha\} \cup \{T_\gamma \mid \gamma < \alpha\}$. So each node has degree 2, and each node has uncountably many ancestors of the form T. This construction is due to R. Bigelow.

Note: If we take a suitable well-ordering of the real numbers, and if we define $T_\alpha = \{(\gamma, \beta) \mid \gamma \succ \alpha\}$, we can improve this construction slightly, obtaining a nested collection where each node has degree 2, uncountable level, *and* 2 sons.

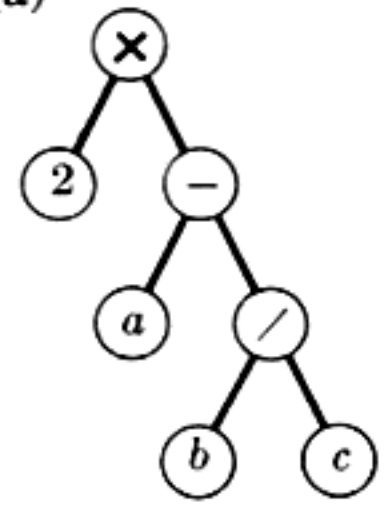
12. We impose an additional condition on the partial ordering (analogous to that of "nested sets") to ensure that it corresponds to a forest: If $x \preceq y$ and $x \preceq z$ then either $y \preceq z$ or $z \preceq y$. To make a tree, also assert the existence of a node r such that $x \succeq r$ for all x. A proof that this gives an unordered tree as defined in the text, when the number of nodes is finite, runs like the proof for nested sets in Ex. 10.

13. $a_1,\ a_1.a_2,\ \cdots,\ a_1.a_2.\ \cdots\ .a_k.$

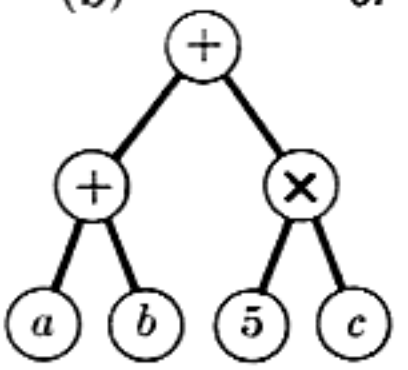
14. Since S is nonempty, it contains an element "$1.a_1.\ \cdots\ .a_k$" where k is as small as possible; if $k > 0$ we also take a_k as small as possible in S, and we immediately see that k must be 0, i.e. S contains the element "1". Let this be the root. All other elements have $k > 0$, and so the remaining elements of S can be partitioned into sets $S_j = \{1.j.a_2.\ \cdots\ .a_k\}$, $1 \leq j \leq m$, for some $m \geq 0$. If $m \neq 0$, and S_m is non-empty, by reasoning as above we find "$1.j$" is in S_j for each S_j, so that each S_j is non-empty. Then it is easy to see that the sets $S_j' = \{1.a_2.\ \cdots\ .a_k \mid 1.j.a_2.\ \cdots\ .a_k \text{ is in } S_j\}$ satisfy the same condition as S did, so by induction each of the S_j forms a tree also.

15. Let the root be "1" and let the root of the left subtree of α be $\alpha.0$; the root of the right subtree of α may be named $\alpha.1$. For example in Fig. 19(a), King Christian IX appears in two positions, 1.0.0.0.0 and 1.1.0.0.1.0. For brevity we may drop the decimal points, and write merely 10000 and 110010. *Note:* This notation is due to Francis Galton; see *Natural Inheritance* (Macmillan, 1889), 249. For "pedigrees", it is more mnemonic to use F and M in place of 0 and 1; e.g., Christian IX is Charles's $MFFMF$, i.e. Charles's mother's father's father's mother's father. The 0 and 1 convention is interesting for another reason as it gives us an important correspondence between nodes in a binary tree and positive integers expressed in the binary system (namely, memory addresses in a computer).

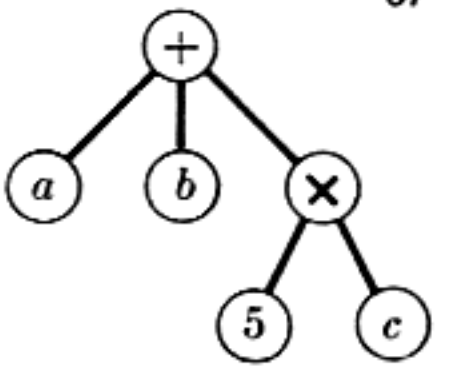
16. (a)



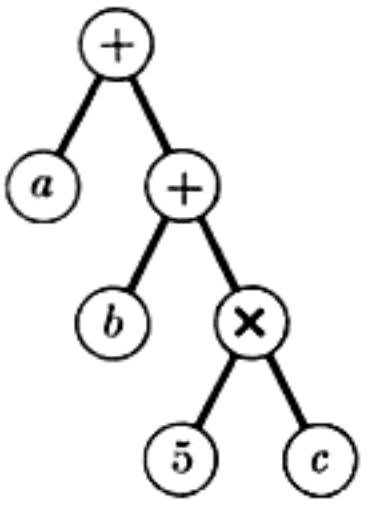
(b)



or



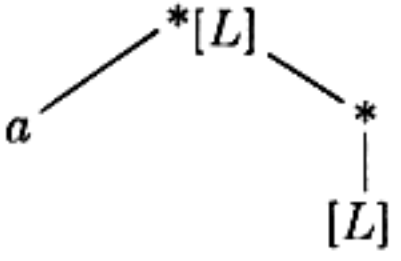
or



17. root(T) = A; root($T[2]$) = C; root($T[2, 2]$) = E.

18. $L[5, 1, 1]$ = "(2)". $L[3, 1]$ is nonsense, since $L[3]$ is an empty List.

19.



$L[2]$ = "(L)"; $L[2, 1, 1]$ = "a".

20. (Intuitively, the correspondence between b-trees and binary trees is obtained by removing all terminal nodes of the b-tree; see the constructions in Sections 2.3.4.4 and 2.3.4.5.) Let a b-tree with one node correspond to the empty binary tree; and let a b-tree with more than one node, consisting therefore of a root r and b-trees T_1 and T_2, correspond to the binary tree with root r, left subtree T'_1, and right subtree T'_2, where T_1 and T_2 correspond respectively to T'_1 and T'_2.

21. $1 + 0 \,.\, n_1 + 1 \,.\, n_2 + \cdots + (m - 1) \,.\, n_m$. For the number of nodes in the tree is $n_0 + n_1 + n_2 + \cdots + n_m$, and this also equals $1 +$ (number of sons in the tree) $= 1 + 0 \,.\, n_0 + 1 \,.\, n_1 + 2 \,.\, n_2 + \cdots + m \,.\, n_m$.

SECTION 2.3.1

1. INFO(T) = A, INFO(RLINK(T)) = C, etc.; the answer is H.

2. Preorder: 1245367; symmetric order: 4251637; endorder: 4526731.

3. The statement is true (notice for example that nodes 4, 5, 6, 7 always appear in this order in Ex. 2). The result is immediately proved by induction on the size of the binary tree.

4. It is the reverse of endorder. (This is easily proved by induction.)

5. For example in the tree of Ex. 2, preorder is (using binary notation which is in this case equivalent to the Dewey system) 1, 10, 100, 101, 11, 110, 111. This is recognizable as sorting from left to right, as in a dictionary.

In general, the nodes will be listed in preorder if they are sorted lexicographically from left to right, with "blanks" treated as less than 0 or 1. The nodes will be listed in postorder if they are sorted lexicographically with 0 < "blank" < 1. For endorder, use 0 < 1 < "blank".

6. The fact that $p_1\, p_2 \ldots p_n$ is obtainable with a stack is readily proved by induction on n, or in fact we may observe that Algorithm T does precisely what is required in its stack actions. (The corresponding sequence of S's and X's as in Ex. 2.2.1–3 is the same as the sequence of 1's and 2's as subscripts in the dual order, see Ex. 18.)

Conversely, if $p_1 p_2 \ldots p_n$ is obtainable with a stack and if $p_k = 1$, then $p_1 \ldots p_{k-1}$ is a permutation of $\{2, \ldots, k\}$ and $p_{k+1} \ldots p_n$ is a permutation of $\{k+1, \ldots, n\}$ each of which are obtainable by stack, and which are the permutations corresponding to the left and right subtrees. The proof now proceeds by induction.

7. From the preorder, the root is known; then from the postorder, we know the left subtree and the right subtree; and in fact we know the preorder and postorder of the nodes in the latter subtrees. Hence the tree is readily constructed (and indeed it is quite amusing to construct a simple algorithm which links the tree together in the normal fashion, starting with the nodes linked together in preorder in LLINK and in postorder in RLINK). Similarly, postorder and endorder together characterize the structure. But preorder and endorder do not; there are two binary trees having "AB" as preorder and "BA" as endorder. If all nonterminal nodes of a binary tree have *both* branches nonempty, its structure *is* characterized by preorder and endorder.

8. (a) Trees with all LLINKs null. (b) Trees with zero or one nodes. (c) Trees with all RLINKs null.

9. T1 once, T2 $(2n+1)$ times, T3 n times, T4 $(n+1)$ times, T5 n times. This is derived by induction or by Kirchhoff's law.

10. A binary tree with all RLINKs null will cause all n node addresses to be put in the stack before any are removed.

11. Let a_{nk} be the number of binary trees with n nodes for which the stack in Algorithm T never contains more than k items. If $g_k(z) = \sum_n a_{nk} z^n$, we find $g_1(z) = 1/(1-z)$, $g_2(z) = 1/(1 - z/(1-z)) = (1-z)/(1-2z)$, . . . , $g_k(z) = 1/(1 - zg_{k-1}(z)) = q_{k-1}(z)/q_k(z)$ where $q_{-1}(z) = q_0(z) = 1$, $q_{k+1}(z) = q_k(z) - zq_{k-1}(z)$. $g_\infty(z)$ is, of course, the generating function for the number of binary trees, see Section 2.3.4. These formulas make it possible to calculate the value a_{nk} readily, but seem to give no obvious way to find the asymptotic value of $na_{nn} - \sum_{0 \le k < n} a_{nk}$ which is required for this problem. Presumably the mean is asymptotic to $\alpha\sqrt{n}$ for some α. (See the references in the answer to Ex. 2.3.4.5–5, which deal with the case of oriented trees.) Empirical calculations show good agreement with the formula $\sqrt{\pi n} - 1.6$.

12. Visit NODE(P) between step T2 and T3, instead of between step T4 and T2. For the proof, show that "Starting at step T2 with . . . original value A[1] . . . A[m]." essentially as in the text.

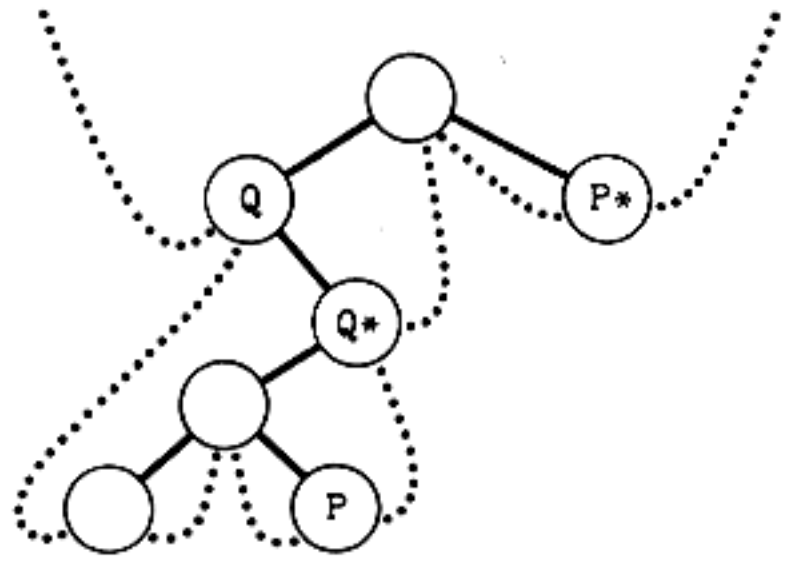
13. Let steps T1, T2 be unchanged. In step T3, put (P, 0) on top of the stack instead of just P. In step T4, when the stack is not empty, set (P, d) $\Leftarrow$ A; and if $d = 0$ set A $\Leftarrow$ (P, 1), P $\leftarrow$ RLINK(P), and return to T2. Finally, step T5 becomes "Visit NODE(P), and go to T4."

14. By induction, there are always exactly $n+1$ A links (counting T when it is null). There are n non-null links, counting T, so the remark in the text about the majority of null links is justified.

15. There is a thread LLINK pointing to a node if and only if it has a nonempty right subtree; there is a thread RLINK pointing to a node if and only if its left subtree is nonempty. (See Fig. 24.)

16. If LTAG(Q) = "+", Q* = LLINK(Q), i.e. one step down to the left. Otherwise Q* is obtained by going upwards in the tree (if necessary) repeatedly until the first time it is possible to go down to the right without retracing steps; typical examples

are the trips from P to P* and from Q to Q* in the following tree:



17. If LTAG(P) = "+", set Q ← LLINK(P) and terminate; otherwise set Q ← P and now set Q ← RLINK(Q) zero or more times until finding RTAG(Q) = "+", and finally set Q ← RLINK(Q) one further time.

18. Modify Algorithm T by inserting a step T2a, "Visit NODE(P) the first time"; and in step T5, we are visiting NODE(P) the second time.

Given a threaded tree the traversal is extremely simple:

$$(P, 1)^{\Delta} = (\mathrm{LLINK}(P), 1) \text{ if } \mathrm{LTAG}(P) = \text{"+"}, \text{ otherwise } (P, 2);$$

$$(P, 2)^{\Delta} = (\mathrm{RLINK}(P), 1) \text{ if } \mathrm{RTAG}(P) = \text{"+"}, \text{ otherwise } (\mathrm{RLINK}(P), 2).$$

In each case, we move at most one step in the tree; so in practice the dual order and the values of d and e are embedded in a program and not explicitly mentioned.

Suppressing all the "first visits" gives us precisely algorithms T and S; suppressing all the "second visits" gives us the solutions to Exs. 12 and 17.

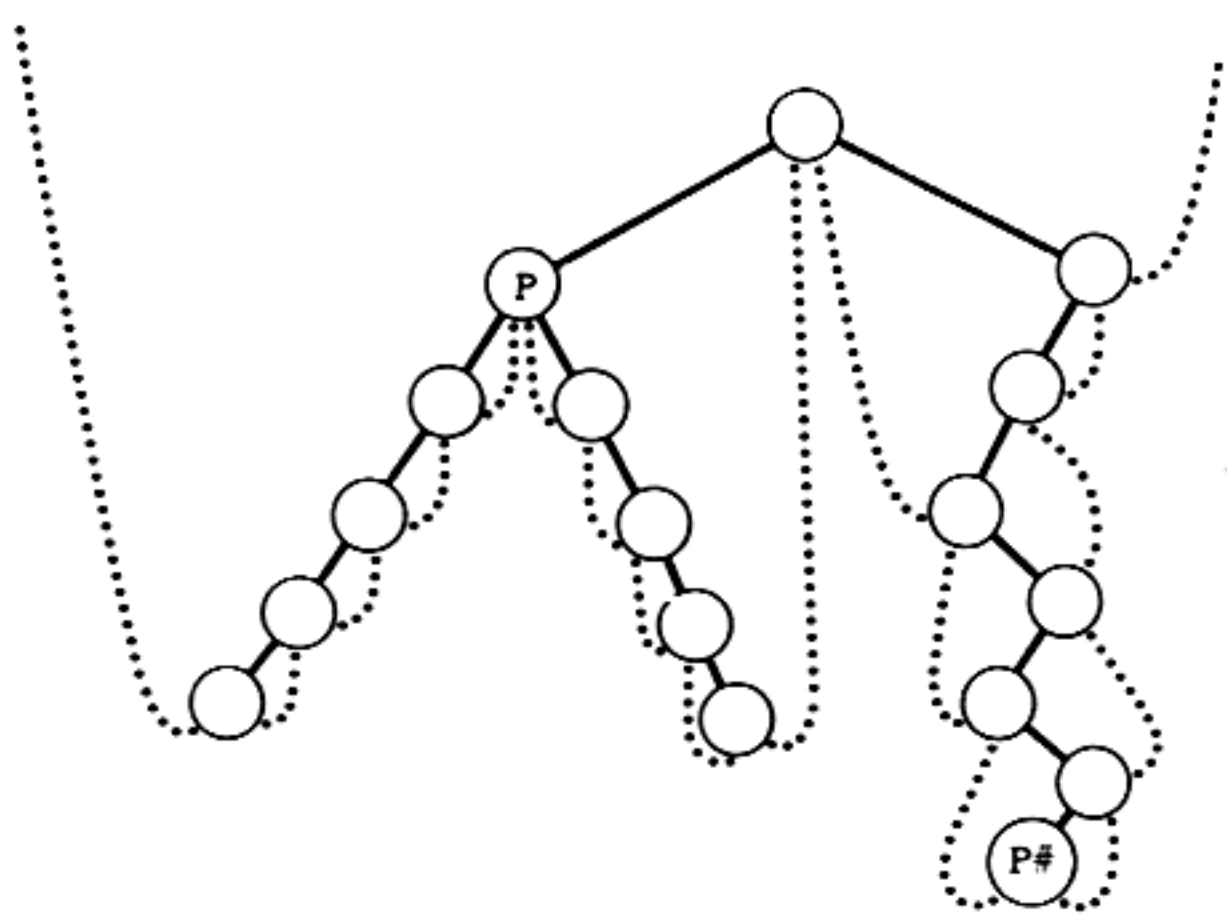
19. **E1.** Set Q ← P. If Q = HEAD, go to E5.

E2. If RTAG(Q) = "+", set Q ← RLINK(Q) and repeat this step.

E3. Set Q ← RLINK(Q). If now LLINK(Q) = P, go to E5; otherwise set Q ← LLINK(Q).

E4. If RLINK(Q) ≠ P, set Q ← RLINK(Q) and repeat this step; otherwise terminate the algorithm.

E5. If RTAG(Q) = "+", set Q ← Q$ using Algorithm S and repeat this step. ∎



Note: There seems to be no more efficient algorithm than this (consider for example P and P# in the tree shown at the bottom of the previous page), and it is inferior to an algorithm using a stack (like Ex. 13) for traversing an entire tree in endorder.

20. Replace lines 06–09 by:

```
T3  ENT4  0,6
    LD6   AVAIL
    J6Z   OVERFLOW
    LDX   0,6(LINK)
    STX   AVAIL
    ST5   0,6(INFO)
    ST4   0,6(LINK)
```

Replace lines 12–13 by:

```
LD4   0,6(LINK)
LD5   0,6(INFO)
LDX   AVAIL
STX   0,6(LINK)
ST6   AVAIL
ENT6  0,4
```

If two more lines of code are added at line 06

```
T3  LD4   0,5(LLINK)
    J4Z   T5
```

the running time goes down from $30n + a + 3$ to $27a + 7n - 23$ units. (This same device would reduce the running time of program T to $12a + 7n - 7$, which is a slight improvement, if we set $a = (n + 1)/2$.)

21. The following algorithm may in fact be used for traversal in any of the three orders:

V1. [Initialize.] Set P ← LOC(T), Q ← T. If Q = Λ, terminate the algorithm.

V2. [Preorder visit.] If traversing in preorder, visit NODE(Q).

V3. [Go to left.] Set R ← LLINK(Q). If R ≠ Λ, set LLINK(Q) ← P, P ← Q, Q ← R, and go back to V2. (It is assumed that RTAG(P) is initially "+".)

V4. [Postorder visit.] If traversing in postorder, visit NODE(Q).

V5. [Go to right.] Set R ← RLINK(Q). If R ≠ Λ, set RTAG(Q) ← "−", RLINK(Q) ← P, P ← Q, Q ← R, go to V2.

V6. [Endorder visit.] If traversing in endorder, visit NODE(Q).

V7. [Go up.] If P = LOC(T), terminate the algorithm. Otherwise if RTAG(P) = "+", set R ← LLINK(P), LLINK(P) ← Q, Q ← P, P ← R, and go to V4. Otherwise set R ← RLINK(P), RTAG(P) ← "+", RLINK(P) ← Q, Q ← P, P ← R, and go to V6. ▮

Algorithms related to this one are discussed further in Section 2.3.5. It is apparently impossible to solve the stated problem without using the additional one-bit field RTAG in each node, but this has not been proved. The above algorithm works even if the tree involves "shared" subtrees, and it has the possible further advantage that NODE(P) is the father of the node being visited. But the fact that it radically affects the structure while it is doing the processing makes it unsuitable for use except in special situations, such as in connection with the copying routine, Algorithm C.

22. Let rI4 ≡ R, rI5 ≡ Q, rI6 ≡ P, and use other conventions of Programs T and S.

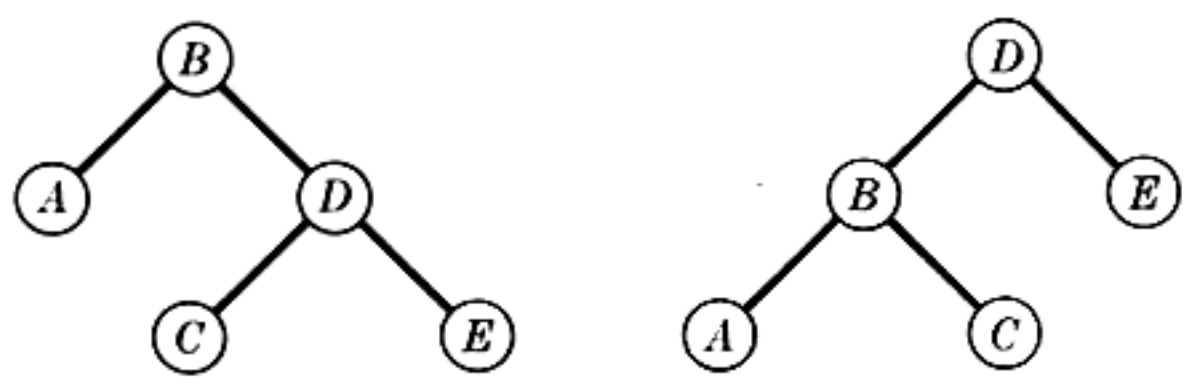
```
V1  ENT6  T            1    VI. Initialize. P ← LOC(T).
    LD5   T(LLINK)     1    Q ← T.
```

```
     J5NZ  V3              1
     JMP   DONE            0      Special exit for empty tree
1H   ST6   0,5(LLINK)     a − 1   LLINK(Q) ← P.
2H   ENT6  0,5            n − 1   P ← Q.
     ENT5  0,4            n − 1   Q ← R.
V3   LD4   0,5(LLINK)       n     V3. Go to left. R ← LLINK(Q).
     J4NZ  1B               n     Repeat if R ≠ Λ.
V4   JMP   VISIT            n     V4. Post-order visit.
V5   LD4   1,5(RLINK)       n     V5. Go to right. R ← RLINK(Q).
     J4Z   V7               n     To V7 if R = Λ.
     ENNA  0,6            n − a
     STA   1,5(RLINKT)    n − a   RLINK(Q) ← P, RTAG(Q) ← "−".
     JMP   2B             n − a   P ← Q, Q ← R, go to V3.
V7   ENT4  -T,6             n     V7. Go up.
     J4Z   DONE             n     Terminate if P = LOC (T).
     LD4   1,6(RLINKT)    n − 1   R ← RLINKT(P).
     J4N   1F             n − 1   Jump if RTAG(P) = "−".
     LD4   0,6(LLINK)     a − 1   R ← LLINK(P).
     ST5   0,6(LLINK)     a − 1   LLINK(P) ← Q.
     ENT5  0,6            a − 1   Q ← P.
     ENT6  0,4            a − 1   P ← R.
     JMP   V4             a − 1
1H   ST5   1,6(RLINKT)    n − a   RLINK(P) ← Q, RTAG(P) ← "+".
     ENT5  0,6            n − a   Q ← P.
     ENN6  0,4            n − a   P ← −R.
     JMP   V7             n − a                                  ▮
```

The running time is $23n - 10$ (curiously independent of a), for $n \neq 0$. So the execution time is competitive with Ex. 20.

23. Insertion to the right: RLINKT(Q) ← RLINKT(P), RLINKT(P) ← +Q, LLINK(Q) ← Λ. Insertion to the left, assuming LLINK(P) = Λ: Set LLINK(P) ← Q, LLINK(Q) ← Λ, RLINKT(Q) ← −P. Insertion to the left, between P and LLINK(P) ≠ Λ: Set R ← LLINK(P), LLINK(Q) ← R, and then if RTAG(R) ≠ "−" set R ← RLINK(R) repeatedly until RTAG(R) = "−"; finally, set RLINK(R) ← Q, LLINK(P) ← Q, RLINKT(Q) ← −P. (A more efficient algorithm for the last case can be used if we know a node F such that P = LLINK(F) or P = RLINK(F); assuming the latter, for example, we could set INFO(P) ↔ INFO(Q), RLINK(F) ← Q, LLINK(Q) ← P, RLINKT(P) ← −Q; this takes a fixed amount of time, but since it switches nodes around in memory it is generally not recommended.)

24. No:



25. We first prove (b), by induction on the number of nodes in T, and similarly (c). Now (a) breaks into special cases; write $T \preceq_1 T'$ if (1) holds, $T \preceq_2 T'$ if (2) holds, etc. Then $T \preceq_1 T'$ and $T' \preceq T''$ implies $T \preceq_1 T''$; $T \preceq_2 T'$ and $T' \preceq T''$ implies $T \preceq_2 T''$; and the remaining two cases are treated by proving (a) by induction on the number of nodes in T.

26. If the dual order of T is $(u_1, d_1), (u_2, d_2), \ldots, (u_{2n}, d_{2n})$ where the u's are nodes and the d's are 1 or 2, form the "trace" of the tree $(v_1, s_1), (v_2, s_2), \ldots, (v_{2n}, s_{2n})$, where $v_j = \text{info}\,(u_j)$, and $s_j = l(u_j)$ or $r(u_j)$ according as $d_j = 1$ or 2. Now $T \preceq T'$ if and only if the trace of T (as defined here) *lexicographically* precedes or equals the trace of T'. Formally, this means either $n \leq n'$ and $(v_j, s_j) = (v'_j, s'_j)$ for $1 \leq j \leq n$, or else there is a k for which $(v_j, s_j) = (v'_j, s'_j)$ for $1 \leq j < k$ and either $v_k \prec v'_k$ or $v_k = v'_k$ and $s_k < s'_k$.

27. **R1.** [Initialize.] Set P ← HEAD, P′ ← HEAD′, i.e. the respective list heads of the given right-threaded binary trees. Go to R3.

R2. [Check INFO.] If INFO(P) ≺ INFO(P′), terminate ($T \prec T'$); if INFO(P) ≻ INFO(P′), terminate ($T \succ T'$).

R3. [Go to left.] If LLINK(P) = Λ = LLINK(P′), go to R4; if LLINK(P) = Λ ≠ LLINK(P′), terminate ($T \prec T'$); if LLINK(P) ≠ Λ = LLINK(P′), terminate ($T \succ T'$); otherwise set P ← LLINK(P), P′ ← LLINK(P′) and go to R2.

R4. [End of tree?] If P = HEAD (or, equivalently, if P′ = HEAD′), terminate (T equivalent to T').

R5. [Go to right.] If RTAG(P) = "—" = RTAG(P′), set P ← RLINK(P), P′ ← RLINK(P′), and go to R4. If RTAG(P) = "—" ≠ RTAG(P′), terminate ($T \prec T'$). If RTAG(P) ≠ "—" = RTAG(P′), terminate ($T \succ T'$). Otherwise, set P ← RLINK(P), P′ ← RLINK(P′), and go to R2. ▮

To prove the validity of this algorithm (and therefore to understand how it works), one may show by induction on the size of the tree T_0 that the following statement is valid: "Starting at step R2 with P and P′ pointing to the roots of two nonempty right-threaded binary trees T_0 and T'_0, the algorithm will terminate if T_0 and T'_0 are not equivalent, indicating whether $T_0 \prec T'_0$ or $T_0 \succ T'_0$; the algorithm will reach step R4 if T_0 and T'_0 *are* equivalent, with P and P′ then pointing respectively to the successor nodes of T_0 and T'_0 in postorder."

28. Equivalent *and* similar.

29. Prove by induction on the size of T that the following statement is valid: "Starting at step C2 with P pointing to the root of a nonempty binary tree T and with Q pointing to a node that has empty left and right subtrees, the procedure will ultimately arrive at step C6 after attaching copies of the left and right subtrees of NODE(P) to NODE(Q), and with P and Q pointing respectively to the successor nodes of T and NODE(Q) in preorder."

30. Assume the pointer T in (2) is LLINK(HEAD) in (9).

L1. [Initialize.] Set Q ← HEAD, RLINK(Q) ← Q.

L2. [Advance.] Set P ← Q$. (See below.)

L3. [Thread.] If RLINK(Q) = Λ, set RLINK(Q) ← P, RTAG(Q) ← "−", else set RTAG(Q) ← "+". If LLINK(P) = Λ, set LLINK(P) ← Q, LTAG(P) ← "−", else set LTAG(P) ← "+".

L4. [Done?] If P ≠ HEAD, set Q ← P and return to L2. ▮

Step L2 of this algorithm implies the activation of a postorder traversal coroutine like Algorithm T, with the additional proviso that Algorithm T "visits" HEAD after it has fully traversed the tree. This notation is a convenient simplification in the description of tree algorithms, since we need not repeat the stack mechanisms of Algorithm T over and over again. Of course Algorithm S cannot be used during step L2, since the tree hasn't been threaded yet. But the algorithm of Ex. 21 *can* be used in step L2, and this provides us with a very pretty method that threads a tree *without using any auxiliary stack!*

31. (a) Set P ← HEAD; (b) set Q ← P$ (e.g. using Algorithm S, modified for a right-threaded tree); (c) if P ≠ HEAD, AVAIL ⇐ P; (d) if Q ≠ HEAD, set P ← Q and return to (b). [Other solutions which decrease the length of the inner loop are clearly possible, although the order of the basic steps is somewhat critical. The above procedure works since we never return a node to available storage until after Algorithm S has looked at both its LLINK and its RLINK; as observed in the text, each of these links is used precisely once during a complete tree traversal.]

32. RLINK(Q) ← RLINK(P), SUC(Q) ← SUC(P), SUC(P) ← RLINK(P) ← Q, PRED(Q) ← P, PRED(SUC(Q)) ← Q.

33. Inserting NODE(Q) just to the left and below NODE(P) is quite simple: Set LLINKT(Q) ← LLINKT(P), LLINKT(P) ← +Q, RLINK(Q) ← Λ. Insertion to the right is considerably harder, since it essentially requires finding *Q, which is of comparable difficulty to finding Q# (see exercise 19); the node-moving technique discussed in exercise 23 could perhaps be used. So general insertions are more difficult with this type of threading. But the insertions required by Algorithm C are not as difficult as insertions are in general, and in fact the copying process is slightly faster for this kind of threading:

C1. Set P ← HEAD, Q ← U, go to C4. (The assumptions and philosophy of Algorithm C in the text are being used throughout.)

C2. If RLINK(P) ≠ Λ, set R ⇐ AVAIL, LLINK(R) ← LLINK(Q), LTAG(R) ← "−", RLINK(R) ← Λ, RLINK(Q) ← LLINK(Q) ← R.

C3. Set INFO(Q) ← INFO(P).

C4. If LTAG(P) = "+", set R ⇐ AVAIL, LLINK(R) ← LLINK(Q), LTAG(R) ← "−", RLINK(R) ← Λ, LLINK(Q) ← R, LTAG(Q) ← "+".

C5. Set P ← LLINK(P), Q ← LLINK(Q).

C6. If P ≠ HEAD, go to C2. ▮

The algorithm now seems almost too simple to be correct!

Algorithm C for threaded or right-threaded binary trees is slightly longer due to the extra time to calculate P*, Q* in step C5.

It would be possible to thread RLINKs in the usual way or to put #P in RLINK(P), in conjunction with the above copying method, by appropriately setting RLINK(R) and RLINKT(Q) in steps C2 and C4.

34. A1. Set Q ← P, and then repeatedly set Q ← RLINK(Q) zero or more times until RTAG(Q) = "—".

A2. Set R ← RLINK(Q). If LLINK(R) = P, set LLINK(R) ← Λ; otherwise set R ← LLINK(R), and repeatedly set R ← RLINK(R) zero or more times until RLINK(R) = P, then finally set RLINKT(R) ← RLINKT(Q). (This step has removed NODE(P) and its subtrees from the original tree.)

A3. Set RLINK(Q) ← HEAD, LLINK(HEAD) ← P. ▮

(The key to inventing and/or understanding this algorithm is the construction of good "before and after" diagrams.)

36. No; cf. the answer to Ex. 1.2.1–15(e).

37. If

$$\text{LLINK(P)} = \text{RLINK(P)} = \Lambda$$

in the representation (2), let

$$\text{LINK(P)} = \Lambda;$$

otherwise let LINK(P) = Q where NODE(Q) corresponds to NODE(LLINK(P)) and NODE(Q+1) to NODE(RLINK(P)). The condition LLINK(P) or RLINK(P) = Λ is represented by a sentinel in NODE(Q) or NODE(Q+1) respectively. This representation uses between n and $2n - 1$ memory positions; under the stated assumptions, (2) would require 18 words of memory, compared to 11 in the present scheme. Insertion and deletion operations are approximately of equal efficiency in either representation.

SECTION 2.3.2

1. If B is empty, $F(B)$ is an empty forest. Otherwise, $F(B)$ consist of a tree T plus the forest $F(\text{rightsubtree}(B))$, where $\text{root}(T) = \text{root}(B)$ and $\text{subtrees}(T) = F(\text{leftsubtree}(B))$.

2. The number of zeroes in the binary notation is the number of decimal points in the decimal notation, and the exact formula for the correspondence is

$$a_1 \,.\, a_2 \,.\, \cdots \,.\, a_k \leftrightarrow 1^{a_1}01^{a_2-1}0 \cdots 01^{a_k-1},$$

where 1^a denotes a ones in a row.

3. Sort the Dewey decimal notations for the nodes lexicographically (from left to right, as in a dictionary), placing a shorter sequence $a_1 \,.\, \cdots \,.\, a_k$ in front of its extensions $a_1 \,.\, \cdots \,.\, a_k \,.\, \cdots \,.\, a_r$ for preorder, and behind its extensions for postorder. (Thus, if we were sorting words instead of sequences of numbers, we would place the words *cat*, *cataract* in usual dictionary order, for preorder, but with the order of initial subwords reversed "*cataract, cat*", for postorder.) These rules are readily proved by induction on the size of the tree.

4. It is false. The terminal nodes do occur in the same relative position in preorder and postorder, as induction readily shows; but for example in a tree with exactly two levels we find that endorder is precisely the opposite. The corresponding statement *is* true for binary trees (Ex. 2.3.1–3).

5. First list the roots of all the trees; then list the second-level nodes of the rightmost tree; then the third-level nodes of the rightmost second-level node listed; and so on, always giving the nodes of a complete family together, using the farthest right, highest level family that has not already been listed, whose parent has been listed.

6. We have preorder(T) = preorder(T'), and endorder(T) = postorder(T'), even if T has nodes with only one son; the remaining two orders are not in any simple relation (for example, the root of T comes at the end in one case and about in the middle in the other).

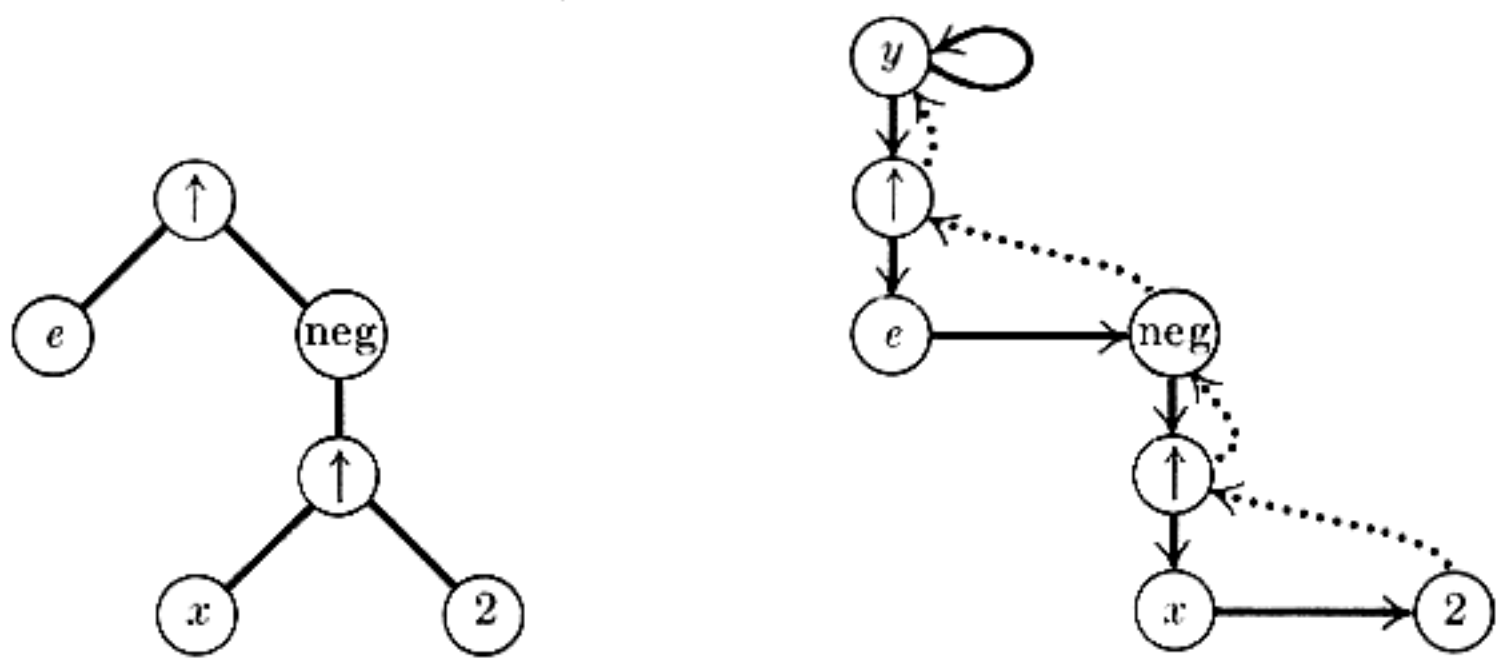
7. (a) yes; (b) no; (c) no; (d) no; (e) yes; (f) yes.

8. $T \preceq T'$ means info(root(T)) $\prec$ info(root(T')), or these info's are equal and either (a): the subtrees of root(T) are $T_1, \ldots, T_n$ and the subtrees of root(T') are $T'_1, \ldots, T'_{n'}$, where there is a k such that T_j is equivalent to T'_j for $1 \le j < k$ but T_k is $\preceq$ and not equivalent to T'_k; or (b): with the notation of (a), T_j is equivalent to T'_j for $1 \le j \le n$, and $n \le n'$.

9. The number of nonterminal nodes is one less than the number of right links that are Λ, in a nonempty forest, because the null right links correspond to the rightmost son of each nonterminal node, and also to the root of the rightmost tree in the forest. (This fact gives another proof of Ex. 2.3.1–14, since obviously the number of null left links is equal to the number of *terminal* nodes.)

10. The forests are similar if and only if $n = n'$, and $s(u_j) = s(u'_j)$, $1 \le j \le n$; they are equivalent if and only if in addition info(u_j) = info(u'_j), $1 \le j \le n$. The proof is similar to the previous proof, by generalizing Lemma 2.3.1P (take $f(u) = s(u) - 1$).

11.



12. If INFO(Q1) $\neq 0$, do the following operations: set R ← COPY(P1); then if TYPE(P2) = 0 and INFO(P2) $\neq 2$, set R ← TREE("↑",R,INFO(P0) − 1); if TYPE(P2) $\neq 0$, set R ← TREE("↑",R,TREE("−",COPY(P2),"1")); then set Q1 ← MULT(Q1,MULT(COPY(P2),R)).

If INFO(Q) $\neq 0$, set Q ← TREE("×",MULT(TREE("ln",COPY(P1)),Q),TREE("↑", COPY(P1),COPY(P2)). Finally go to DIFF[4].

13. The following program implements Algorithm C with rI1 ≡ P, rI2 ≡ Q, rI3 ≡ R, and with "list head" changed to "root of tree":

```
62    ST3   6F(0:2)    Save contents of rI3, rI2.
63    ST2   7F(0:2)    C1. Initialize.
64    ENT2  8F         Start by creating NODE(U) with
```

```
65          JMP   1F              RLINK(U) = Λ.
66     2H   ENN2  0,2             Q ← −Q.
67     C2   LDA   0,1             C2. Anything to right?
68          JAN   C3              Jump if RTAG(P) = "−".
69     1H   LD3   AVAIL           R ⇐ AVAIL.
70          J3Z   OVERFLOW
71          LDA   0,3(LLINK)
72          STA   AVAIL
73          LDA   0,2
74          STA   0,3             Set LLINKRLINKT(R) ← LLINKRLINKT(Q).
75          ST3   0,2(RLINKT)     RLINKT(Q) ← +R.
76     C3   LDA   1,1             C3. Copy INFO.
77          STA   1,2             INFO field copied.
78          LDA   0,1(TYPE)
79          STA   0,2(TYPE)       TYPE field copied.
80     C4   STZ   0,2(LLINK)      C4. Anything to left?
81          LDA   0,1(LLINK)
82          JAZ   C5              Jump if LLINK(P) = Λ.
83          LD3   AVAIL           R ⇐ AVAIL.
84          J3Z   OVERFLOW
85          ST3   0,2(LLINK)      LLINK(Q) ← R.
86          LDA   0,3(LLINK)
87          STA   AVAIL
88          ENNA  0,2
89          STA   0,3(RLINKT)     RLINKT(R) ← −Q.
90          ENT2  0,3             C5. Advance. Set Q ← R.
91          LD1   0,1(LLINK)      Set P ← LLINK(P) = P*.
92          JMP   C2
93     8H   CON   0               Zero constant for initialization
94     C5   LD2N  0,2(RLINKT)     Q ← −RLINKT(Q).
95          LD1   0,1(RLINK)      P ← RLINK(P).
96          J2N   2B              Jump if RTAG(Q) = "+".
97     C6   J2NZ  C5              C6. Test if complete.
98          LD1   8B(RLINK)       rI1 ← location of first node created.
99          STZ   8B              Reset constant to zero for next COPY.
100    6H   ENT3  *               Restore index registers.
101    7H   ENT2  *                                          ▮
```

14. Let a be the number of nonterminal (operator) nodes copied. The number of executions of the various lines in the previous program is as follows: 62–65, 1; 66, $n - 1 - a$; 67–68, $n - 1$; 69–75, $n - a$; 76–83, n; 84–92, a; 94–96, n; 97, $a + 1$; 98–101, 1. The total time comes to $(35n + a + 9)u$; about $\frac{1}{4}$ of this is to get available nodes, $\frac{1}{4}$ to traverse, and $\frac{1}{2}$ to copy the INFO and LINK information.

15. Comments are left to the reader.

```
216    DIV  LDA   1,6
217         JAZ   1F
218         JMP   COPYP2
219         ENTA  SLASH
220         ENTX  0,6
```

```
221          JMP   TREE2
222          ENT6  0,1
223     1H   LDA   1,5
224          JAZ   SUB
225          JMP   COPYP2
226          ST1   1F(0:2)
227          ENTA  CON2
228          JMP   TREE0
229          ENTA  UPARROW
230     1H   ENTX  *
231          JMP   TREE2
232          ST1   1F(0:2)
233          JMP   COPYP1
234          ENTA  0,1
235          ENT1  0,5
236          JMP   MULT
237          ENTX  0,1
238     1H   ENT1  *
239          ENTA  SLASH
240          JMP   TREE2
241          ENT5  0,1
242          JMP   SUB
```

16. Comments are left to the reader.

```
243     PWR  LDA   1,6
244          JAZ   4F
245          JMP   COPYP1
246          ST1   R(0:2)
247          LDA   0,3(TYPE)
248          JANZ  2F
249          LDA   1,3
250          DECA  2
251          JAZ   3F
252          INCA  1
253          STA   CON0+1
254          JMP   TREE0
255          STZ   CON0+1
256          JMP   5F
257     2H   JMP   COPYP2
258          ST1   1F(0:2)
259          ENTA  CON1
260          JMP   TREE0
261     1H   ENTX  *
262          ENTA  MINUS
263          JMP   TREE2
264     5H   LDX   R(0:2)
265          ENTA  UPARROW
266          JMP   TREE2
267          ST1   R(0:2)
```

```
268     3H    JMP    COPYP2
269           ENTA   0,1
270     R     ENT1   *
271           JMP    MULT
272           ENTA   0,6
273           JMP    MULT
274           ENT6   0,1
275     4H    LDA    1,5
276           JAZ    ADD
277           JMP    COPYP1
278           ENTA   LOG
279           JMP    TREE1
280           ENTA   0,1
281           ENT1   0,5
282           JMP    MULT
283           ST1    1F(0:2)
284           JMP    COPYP1
285           ST1    2F(0:2)
286           JMP    COPYP2
287     2H    ENTX   *
288           ENTA   UPARROW
289           JMP    TREE2
290     1H    ENTX   *
291           ENTA   TIMES
292           JMP    TREE2
293           ENT5   0,1
294           JMP    ADD    ▮
```

20. More generally, let u and v be any nodes of a forest. If u is an ancestor of v, it is immediate by induction that u precedes v in preorder and follows v in postorder. Conversely, suppose u precedes v in preorder and follows v in postorder; we must show that u is an ancestor of v. This is clear if u is the root of the first tree. If u is another node of the first tree, v must be also, since u follows v in postorder; so induction applies. If u is not in the first tree, v must not be either, since u precedes v in preorder; so again induction applies.

21. If `NODE(P)` is a binary operator, pointers to its two operands are `P1 = LLINK(P)` and `P2 = RLINK(P1) = $P`. Algorithm D makes use of the fact that `P2$ = P`, so that `RLINK(P1)` may be changed to `Q1`, a pointer to the derivative of `NODE(P1)`, then later `RLINK(P1)` is reset in step D3. For ternary operations, it is difficult to generalize this trick; we would have, say, `P1 = LLINK(P)`, `P2 = RLINK(P1)`, `P3 = RLINK(P2) = $P`. Now after computing the derivative `Q1`, we could set `RLINK(P1)` ← `Q1` temporarily, and then after computing the next derivative `Q2` we could set `RLINK(Q2)` ← `Q1` and `RLINK(P2)` ← `Q2` and reset `RLINK(P1)` ← `P2`. But this is certainly inelegant, and it becomes progressively more so as the degree of the operator becomes higher. Therefore the device of temporarily changing `RLINK(P1)` in Algorithm D is definitely a *trick*, not a *technique;* a more aesthetic way to control a differentiation process, because it generalizes to operators of higher degree and does not rely on isolated tricks, may be based on Algorithm 2.3.3F, and this is discussed in detail in Ex. 2.3.3–3.

22. From the definition it follows immediately that the relation is transitive, i.e. if $T \subseteq T'$ and $T' \subseteq T''$ then $T \subseteq T''$. (In fact the relation is easily seen to be a partial ordering.) Clearly if we let f be the function taking nodes into themselves, $l(T) \subseteq T$ and $r(T) \subseteq T$. Therefore if $T \subseteq l(T')$ or $T \subseteq r(T')$ we must have $T \subseteq T'$.

Suppose f_l and f_r are functions that respectively show $l(T) \subseteq l(T')$ and $r(T) \subseteq r(T')$. Let $f(u) = f_l(u)$ if u is in $l(T)$, $f(u) = \text{root}(T')$ if u is $\text{root}(T)$, otherwise $f(u) = f_r(u)$. Now it follows easily that f shows that $T \subseteq T'$; for example, if we let $r'(T)$ denote $r(T) - \text{root}(T)$ we have $\text{preorder}(T) = \text{root}(T)\ \text{preorder}(l(T))\ \text{preorder}(r'(T))$; $\text{preorder}(T') = f(\text{root}(T))\ \text{preorder}(l(T'))\ \text{preorder}(r'(T'))$.

The converse does not hold, e.g. consider the subtrees with roots b and b' in Fig. 25.

SECTION 2.3.3

1. The family-order representation becomes the preorder representation if LLINK and RLINK are consistently interchanged in the binary tree. Therefore anything that is true for preorder representation remains true for family-order under this correspondence.

2. Make the following changes in Algorithm F: Step F1, change to "last node of the forest in preorder." Step F2, change "$f(x_d), \ldots, f(x_1)$" to "$f(x_1), \ldots, f(x_d)$" in two places. Step F4, "If P is the first node in preorder, terminate the algorithm. (Then the stack contains $f(\text{root}(T_1)), \ldots, f(\text{root}(T_m))$, from top to bottom, where $T_1, \ldots, T_m$ are the trees of the given forest, from left to right.) Otherwise set P ← *P (P ← P − 1 in the given representation), and return to F2."

3. In step D1, also set S ← Λ. (S is a link variable that links to the top of the stack.) Step D2 becomes, for example, "If NODE(P) denotes a unary operator, set Q ← S, S ← RLINK(Q), P1 ← LLINK(P); if it denotes a binary operator, set Q ← S, Q1 ← RLINK(Q), S ← RLINK(Q1), P1 ← LLINK(P), P2 ← RLINK(P1). Then perform DIFF[TYPE(P)]." Step D3 becomes "Set RLINK(Q) ← S, S ← Q." Step D4 becomes "Set P ← PS." The operation LLINK(DY) ← Q may be avoided in step D5 if we assume S ≡ LLINK(DY). This technique clearly generalizes to ternary and higher-order operators.

4. A representation like (10) takes $n - m$ LLINKs and $n + (n - m)$ RLINKs. The difference in total number of links is $n - 2m$ between the two forms of representation. Arrangement (10) is superior when the LLINK and INFO fields require about the same amount of space in a node and when m is rather large, i.e. when the degree of nonterminal nodes is high.

5. It would certainly be silly to include threaded RLINKs, since an RLINK thread just points to FATHER anyway. Threaded LLINKs as in 2.3.2–(4) would be useful if it is necessary to move leftward in the tree, for example if we wanted to traverse a tree in reverse postorder or in family-order; but these operations are not significantly harder without threaded LLINKs unless the nodes tend to have very high degrees.

6. **L1.** Set P ← FIRST, FIRST ← Λ.

L2. If P = Λ, terminate. Otherwise set Q ← RLINK(P).

L3. If FATHER(P) = Λ, set RLINK(P) ← FIRST, FIRST ← P; otherwise set RLINK(P) ← LSON(FATHER(P)), LSON(FATHER(P)) ← P.

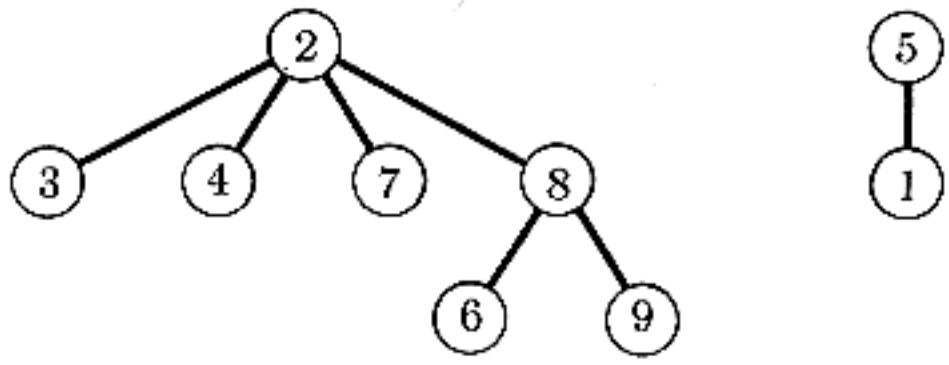
L4. Set P ← Q and return to L2. ▮

7. $\{1, 5\}\ \{2, 3, 4, 7\}\ \{6, 8, 9\}$.

8. Perform step E3 of Algorithm E, then test if $j = k$.

9.

FATHER[k]:	5	0	2	2	0	8	2	2	8
k:	1	2	3	4	5	6	7	8	9



10. (It would be of interest to solve the similar problem for the following "speeded up" version of the equivalence algorithm when n is large, as suggested by Alan Tritter: In step E4, set FATHER[x] $\leftarrow k$ for all values $x \neq k$ which were encountered in step E3. This means an extra pass is made up the trees, but it collapses the trees further so that future searches are faster.)

11. It suffices to define the transformation which is done for each input (P, j, Q, k):

T1. If FATHER(P) $\neq \Lambda$, set $j \leftarrow j +$ DELTA(P), P $\leftarrow$ FATHER(P), and repeat this step.

T2. If FATHER(Q) $\neq \Lambda$, set $k \leftarrow k +$ DELTA(Q), Q $\leftarrow$ FATHER(Q), and repeat this step.

T3. If P = Q, check that $j = k$ (otherwise an error has been made in the input, the equivalences are contradictory). If P $\neq$ Q, set DELTA(Q) $\leftarrow j - k$, FATHER(Q) $\leftarrow$ P, LBD(P) $\leftarrow$ min (LBD(P), LBD(Q) + DELTA(Q)), UBD(P) $\leftarrow$ max (UBD(P), UBD(Q) + DELTA(Q)).

Note: It is possible to allow the "ARRAY X[l:u]" declarations to occur intermixed with equivalences, or to allow assignment of certain addresses of variables before others are equivalenced to them, etc., under suitable conditions which are not difficult to understand. For further development of this algorithm, see *CACM* 7 (1964), 301–303. Similar algorithms apply to other programming languages, as in COBOL's SAME AREA clauses.

12. (a) Yes. (If this condition is not required, it would be possible to avoid the loops on S which appear in steps A2 and A9.) (b) P returns, but Q does not. To make Q return to the root, one could set Q $\leftarrow$ UP(Q) zero or more times until UP(Q) = Λ.

13. The crucial fact is that the UP chain leading upward from P always mentions the same variables and the same exponents for these variables as the UP chain leading upward from Q, except that the latter chain may include additional steps for variables with exponent zero. (This condition holds throughout most of the algorithm, except during the execution of steps A9 and A10.) Now we get to step A8 either from A3 or from A10, and in each case it was verified that EXP(Q) $\neq 0$. Therefore EXP(P) $\neq 0$ and in particular it follows that P $\neq \Lambda$, Q $\neq \Lambda$, UP(P) $\neq \Lambda$, UP(Q) $\neq \Lambda$, and the result stated in the exercise follows. So the proof depends on showing that the UP chain condition stated above is preserved by the actions of the algorithm.

16. The INFO1, RLINK tables together with the suggestion for computing LTAG in the text gives us the equivalent of a binary tree represented in the usual manner. The

idea is to traverse this tree in postorder, counting degrees as we go:

P1. Let R, D, and I be stacks which are initially empty; then set R $\Leftarrow n+1$, D $\Leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$.

P2. If top(R) $> j+1$, (i.e. if LTAG[j] = "+", if that field were present), go to P5.

P3. If I is empty, terminate the algorithm; otherwise set $i \Leftarrow$ I, $k \leftarrow k+1$, INFO2[k] $\leftarrow$ INFO1[i], DEGREE[k] $\Leftarrow$ D.

P4. If RLINK[i] = 0, go to P3; otherwise delete the top of R (which will equal RLINK[i]).

P5. Set top(D) $\leftarrow$ top(D) $+ 1$, $j \leftarrow j+1$, I $\Leftarrow j$, D $\Leftarrow 0$, and if RLINK[j] $\neq 0$ set R $\Leftarrow$ RLINK[j]. Go to P2. ▮

17. We prove (by induction on the number of nodes in a *single tree* T) that if P is a pointer to T, and if the stack is initially empty, steps F2 through F4 will end with the single value $f(\text{root}(T))$ on the stack. This is true for $n = 1$. If $n > 1$, there are $0 < d =$ DEGREE(root(T)) subtrees $T_1, \ldots, T_d$; by induction and the nature of a stack, and since postorder consists of $T_1, \ldots, T_d$ followed by root(T), the algorithm computes $f(T_1), \ldots, f(T_d)$, and then $f(\text{root}(T))$, as desired. The validity of Algorithm F for forests follows.

18. G1. Set the stack empty, and let P point to the root of the tree (the last node in postorder). Evaluate f(NODE(P)).

G2. Push DEGREE(P) copies of f(NODE(P)) down onto the stack.

G3. If P is the first node in postorder, terminate the algorithm. Otherwise set P $\leftarrow$ \$P (this would be simply P $\leftarrow$ P $-$ 1 in (9)).

G4. Evaluate f(NODE(P)) using the value at the top of the stack (which is f(NODE(FATHER(P))). Pop this value off the stack, and return to G2. ▮

Note: An algorithm analogous to this one can be based on preorder instead of postorder as in exercise 2.

SECTION 2.3.4.1

1. (B, A, C, D, B), (B, A, C, D, E, B), (B, D, C, A, B), (B, D, E, B), (B, E, D, B), (B, E, D, C, A, B).

2. Let $(V_0, V_1, \ldots, V_n)$ be a path of smallest possible length from V to V'. If now $V_j = V_k$ for some $j < k$, $(V_0, \ldots, V_j, V_{k+1}, \ldots, V_n)$ is a shorter path.

3. (The fundamental path traverses e_3 and e_4 once, but cycle C_2 traverses them "-1" times, giving a net total of zero.) Traverse the following edges: e_1, e_2, e_6, e_7, e_9, e_{10}, e_{11}, e_{12}, e_{14}.

4. If not, let G'' be the subgraph of G' obtained by deleting each edge e_j for which $E_j = 0$. Then G'' is a finite graph which has no cycles and at least one edge, so by the proof of Theorem A there is at least one vertex, V, which is adjacent to exactly one other vertex, V'. Let e_j be the edge joining V to V'; then Kirchhoff's equation (1) at vertex V is $E_j = 0$, contradicting the definition of G''.

5. $A = 1 + E_8$, $B = 1 + E_8 - E_2$, $C = 1 + E_8$, $D = 1 + E_8 - E_5$, $E = 1 + E_{17} - E_{21}$, $F = 1 + E_{13} + E_{17} - E_{21}$, $G = 1 + E_{13}$, $H = E_{17} - E_{21}$,

$J = E_{17}$, $K = E_{19} + E_{20}$, $L = E_{17} + E_{19} + E_{20} - E_{21}$, $P = E_{17} + E_{20} - E_{21}$, $Q = E_{20}$, $R = E_{17} - E_{21}$, $S = E_{25}$. *Note:* In this case it is also possible to solve for $E_2, E_5, \ldots, E_{25}$ in terms of $A, B, \ldots, S$; hence there are nine independent solutions, explaining why we eliminated six variables in Eq. 1.3.3–(8).

6. Fundamental cycles: $C_0 = e_0 + e_1 + e_4 + e_9$ (fundamental path is $e_1 + e_4 + e_9$); $C_5 = e_5 + e_3 + e_2$; $C_6 = e_6 - e_2 + e_4$; $C_7 = e_7 - e_4 - e_3$; $C_8 = e_8 - e_9 - e_4 - e_3$. Therefore we find $E_1 = 1$, $E_2 = E_5 - E_6$, $E_3 = E_5 - E_7 - E_8$, $E_4 = 1 + E_6 - E_7 - E_8$, $E_9 = 1 - E_8$.

7. In the case considered in Ex. 5, the number of independent variables among $A, B, \ldots, S$ is the same as the number of independent variables among $E_1, \ldots, E_{27}$, namely the number of fundamental cycles. However in Ex. 6 there are four fundamental cycles and four variables A, B, C, D; in this case we have $A = 1 + E_5 - E_7 - E_8$, $B = E_5$, $C = E_5$, $D = 1 + E_6 - E_8$ so there are only *three* quantities which are actually independent in spite of the four independent variables E_5, E_6, E_7, E_8. The reason is that an increase in E_6 and E_8 together with an equivalent decrease in E_7 makes no change in A, B, C, D; stated another way, there are two ways to go from "start" to "stop" in which the *vertices* are touched the same number of times but the *edges* are not (e.g. Start–A–D–B–C–Stop, Start–A–B–C–D–Stop). Such pairs of paths are impossible in Fig. 3.

If we consider the case where the flowchart has many, many edges (e.g. nearly $\binom{n}{2}$ edges when there are n vertices), it is clear that there are many more fundamental cycles than vertices, and in fact Kirchhoff's law may be of *no help whatever* in detecting redundancies between the number of times each *vertex* is encountered. For example, if we add an edge from B to "stop" in the graph of Ex. 6, the quantities A, B, C, D are essentially independent of each other.

Unfortunately it appears to be difficult to tell in advance just how much help Kirchhoff's law will be if we are concerned about vertices instead of edges. Our beautiful theory works only for the edges! Indeed, it should be stressed that Kirchhoff's law is essentially directed only to edges; there is no easy analog to Eq. (1) in which only vertex variables, not edge variables, appear. In general when we set up equations like the "deductions" made between (7) and (8) in Section 1.3.3, these equations are being set up from the understood edge variables, not the vertex variables.

The moral of this story is that *Kirchhoff's law should not be regarded as a set of equations relating the vertex variables* (although the rather simple examples in Section 1.3.3 and elsewhere in this book may have given this impression); it is properly regarded as a set of relations between the edge variables. When the number of edges does not greatly exceed the number of vertices, Kirchhoff's law will yield important redundancies between the vertex variables, but the precise number is unclear.

8. Several approaches may be used; perhaps the simplest is the following: Each edge from a vertex to itself becomes a "fundamental cycle" all by itself; if there are k edges $e, e', \ldots, e^{(k)}$ between vertices V and V', make $k - 1$ fundamental cycles $e' \pm e, \ldots, e^{(k)} \pm e$ (choosing $+$ or $-$ according as the edges go in the opposite or the same direction), and then proceed as if only edge e were present.

Actually this situation would be much simpler conceptually if we had defined a graph in such a way that multiple edges are allowed between vertices, and edges are allowed from a vertex to itself; paths and cycles would be defined in terms of edges instead of vertices. This type of definition is, in fact, made in the following section for directed graphs.

9. (The following solution is based on the idea that we may print out each edge that does not make a cycle with the preceding edges.) Use Algorithm 2.3.3E, with each pair (a_i, b_i) representing $a_i \equiv b_i$ in the notation of that algorithm. The only change is to print (a_i, b_i) if $j \neq k$ in step E4.

To show this algorithm is valid, we must prove that (a) the algorithm prints out no edges that form a cycle, and (b) if G contains at least one free subtree, the algorithm prints out $n - 1$ edges. Define $j \equiv k$ if there exists a path from V_j to V_k or if $j = k$. This is clearly an equivalence relation, and moreover $j \equiv k$ if and only if this relation can be deduced from the equivalences $a_1 \equiv b_1, \ldots, a_m \equiv b_m$. Now (a) is valid since the algorithm prints out no edges that form a cycle with previously printed edges; (b) is true because if all vertices of G are equivalent, FATHER$[k] = 0$ for precisely one value of k.

10. If the terminals have all been connected together, the corresponding graph must be connected in the technical sense. A minimum number of wires clearly will involve no cycles, so we must have a free tree. By Theorem A, a free tree contains $n - 1$ wires, and a graph with n vertices and $n - 1$ edges is a free tree if and only if it is connected.

11. It is sufficient to prove that when $n > 1$ and $c(n - 1, n)$ is the minimum of the $c(i, n)$, there exists at least one minimum cost tree in which T_{n-1} is wired to T_n. (For, any minimum cost tree with $n > 1$ terminals and with T_{n-1} wired to T_n must also be a minimum cost tree with $n - 1$ terminals if we regard T_{n-1} and T_n as "common", using the stated convention in the algorithm.)

To prove the above statement, suppose we have a minimum cost tree in which T_{n-1} is not wired to T_n. If we add the wire $T_{n-1}T_n$ we obtain a cycle, and any of the other wires in that cycle may be removed; removing the other wire touching T_n gives us another tree, whose total cost is not greater than the original, in which $T_{n-1}T_n$ appears.

12. Keep two auxiliary tables, $a(i)$ and $b(i)$, for $1 \leq i < n$, representing the fact that the cheapest connection from T_i to a chosen terminal is to $T_{b(i)}$, and its cost is $a(i)$; initially $a(i) = c(i, n)$ and $b(i) = n$. Then do the following operation $n - 1$ times: Find i such that $a(i) = \min_{1 \leq j < n} a(j)$; connect T_i to $T_{b(i)}$; for $1 \leq j < n$ if $c(i, j) < a(i)$ set $a(i) \leftarrow c(i, j)$ and $b(i) \leftarrow j$; and set $a(i) \leftarrow \infty$.

(It is somewhat more efficient to avoid the use of ∞, keeping instead a one-way linked list of those j which have not yet been chosen. This straightforward improvement is left to the reader. With or without the improvement, the algorithm takes $O(n^2)$ operations.)

13. We must prove G is connected. If $V \neq V'$ and VV' is not an edge of G, add the edge VV' to G; this introduces a cycle which must involve the new edge, so it may be written $(V, V', V_2, \ldots, V)$; hence there is a path in G from V' to V.

14. If there is no path from V_i to V_j, for some $i \neq j$, then no product of the transpositions will move i to j. So if all permutations are generated, the graph must be connected. Conversely if it is connected, remove edges if necessary until we have a tree. Then renumber the vertices so that V_n is adjacent to only one other vertex, namely V_{n-1}. (See the proof of Theorem A.) Now the transpositions other than $(n - 1, n)$ form a tree with $n - 1$ vertices; so by induction if π is any permutation on $\{1, 2, \ldots, n\}$ which leaves n fixed, π can be written as a product of the transpositions other than $(n - 1, n)$. If π moves n to j then $\pi(j, n - 1)(n - 1, n) = \rho$ fixes n; hence $\pi = \rho(n - 1, n)(j, n - 1)$ can be written as a product of the given transpositions.

1. Let $(e_1, \ldots, e_n)$ be an oriented path of smallest possible length from V to V'. If now $\text{init}(e_j) = \text{init}(e_k)$ for $j < k$, $(e_1, \ldots, e_{j-1}, e_k, \ldots, e_n)$ would be a shorter path; a similar argument applies if $\text{fin}(e_j) = \text{fin}(e_k)$ for $j < k$. Hence $(e_1, \ldots, e_n)$ is simple.

2. Those cycles in which all signs are the same: C_0, C_8, C''_{13}, C_{17}, C''_{19}, C_{20}.

3. For example, use three vertices A, B, C, with arcs from B to A and to C.

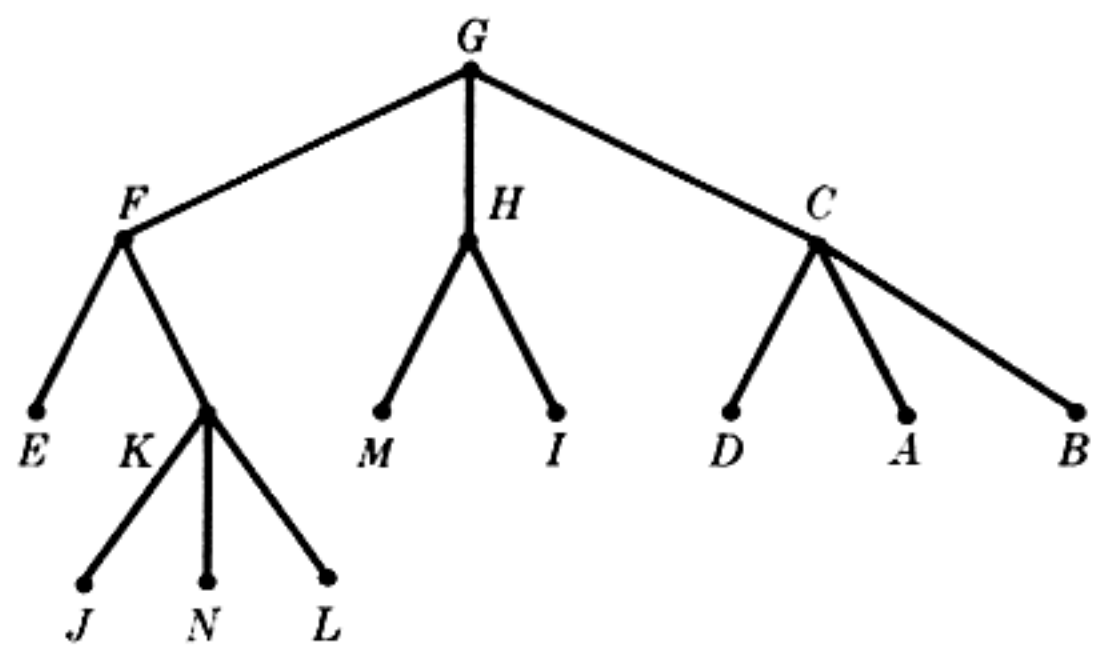
4. If there are no oriented cycles, Algorithm 2.2.3T topologically sorts G. If there is an oriented cycle, topological sorting is clearly impossible. (Depending on how this exercise is interpreted, oriented cycles of length 1 could be excluded from the above discussion.)

5. Let k be the smallest integer such that $\text{fin}(e_k) = \text{init}(e_j)$ for some $j \leq k$. Then $(e_j, \ldots, e_k)$ is an oriented cycle.

6. False (on a technicality), just because there may be several different arcs from one vertex to another.

7. True for finite directed graphs: For if we disregard the direction of the arcs we have a graph with n vertices, $n - 1$ edges, and no cycles. (Any cycle that is not an oriented cycle includes a vertex that is the initial vertex of two arcs, and this would contradict (a).) For infinite graphs the result is obviously false since we might have vertices R, V_1, V_2, V_3, ... and arcs from V_j to V_{j+1} for $j \geq 1$.

9. All arcs point upward.



10. **G1.** Set $k \leftarrow F[j]$, $F[j] \leftarrow 0$.

G2. If $k = 0$, stop; otherwise set $m \leftarrow F[k]$, $F[k] \leftarrow j$, $j \leftarrow k$, $k \leftarrow m$, and repeat step G2. ▮

11. This algorithm combines Algorithm 2.3.3E with the method of the preceding exercise, so that all oriented trees have arcs which correspond to actual arcs in the directed graph; $S[j]$ is an auxiliary table which tells whether an arc goes from j to $F[j]$ ($S[j] = +1$) or from $F[j]$ to j ($S[j] = -1$). Initially $F[1] = \cdots = F[n] = 0$. The following steps may be used to process each arc (a, b):

C1. Set $j \leftarrow a$, $k \leftarrow F[j]$, $F[j] \leftarrow 0$, $s \leftarrow S[j]$.

C2. If $k = 0$, go to C3; otherwise set $m \leftarrow F[k]$, $t \leftarrow S[k]$, $F[k] \leftarrow j$, $S[k] \leftarrow -s$, $s \leftarrow t$, $j \leftarrow k$, $k \leftarrow m$, and repeat step C2.

C3. (Now a appears as the root of its tree.) Set $j \leftarrow b$, and then if $F[j] \neq 0$ repeatedly set $j \leftarrow F[j]$ until $F[j] = 0$.

C4. If $j = a$, go to C5; otherwise set $F[a] \leftarrow b$, $S[a] \leftarrow +1$, print (a, b) as an arc belonging to the free subtree, and terminate.

C5. Print "CYCLE" followed by "(a, b)".

C6. If $F[b] = 0$, terminate. Otherwise if $S[b] = +1$, print "$+(b, F[b])$", else print "$-(F[b], b)$"; set $b \leftarrow F[b]$ and repeat step C6. ▮

12. It equals the in-degree; the out-degree of each vertex can be only 0 or 1.

13. Define a sequence of oriented subtrees of G as follows: G_0 is the vertex R alone. G_{k+1} is G_k, plus any vertex V of G that is not in G_k but for which there is an arc from V to V' where V' *is* in G_k, plus one such arc $e[V]$ for each such vertex. It is immediate by induction that G_k is an oriented tree for all $k \geq 0$, and that if there is an oriented path of length k from V to R in G then V is in G_k. Therefore G_∞, the set of all V and $e[V]$ in any of the G_k, is the desired oriented subtree of G.

14. $(e_{12}, e_{20}, e_{00}, e'_{01}, e_{10}, e_{01}, e'_{12}, e_{22}, e_{21})$, $(e_{12}, e_{20}, e_{00}, e'_{01}, e'_{12}, e_{22}, e_{21}, e_{10}, e_{01})$,
$(e_{12}, e_{20}, e'_{01}, e_{10}, e_{00}, e_{01}, e'_{12}, e_{22}, e_{21})$, $(e_{12}, e_{20}, e'_{01}, e'_{12}, e_{22}, e_{21}, e_{10}, e_{00}, e_{01})$,
$(e_{12}, e_{22}, e_{20}, e_{00}, e'_{01}, e_{10}, e_{01}, e'_{12}, e_{21})$, $(e_{12}, e_{22}, e_{20}, e_{00}, e'_{01}, e'_{12}, e_{21}, e_{10}, e_{01})$,
$(e_{12}, e_{22}, e_{20}, e'_{01}, e_{10}, e_{00}, e_{01}, e'_{12}, e_{21})$, $(e_{12}, e_{22}, e_{20}, e'_{01}, e'_{12}, e_{21}, e_{10}, e_{00}, e_{01})$,

in "lexicographic order"; the eight possibilities come from the independent choices of which of e_{00} or e'_{01}, e_{10} or e'_{12}, e_{21} or e_{22}, should precede the other.

15. If it is connected and balanced, it either has only one vertex or there is an Eulerian circuit which touches all the vertices; twice around that circuit will touch any given vertex V the first time and any other given vertex V' the second time.

16. Consider the directed graph G with vertices $V_1, \ldots, V_{13}$ and with an arc from V_j to V_k for each k in pile j. Winning the game is equivalent to the existence of an Eulerian circuit in this directed graph (for if the game is won the final card turned up must come from the center; this graph is balanced). Now if the game is won, we have an oriented subsubtree by Lemma E. Conversely if the stated configuration is an oriented subtree, the game is won by Theorem D.

17. $\frac{1}{13}$. This answer can be obtained, as the author first obtained it, by laborious enumeration of oriented trees of special types and the application of generating functions, etc., based on the methods of Section 2.3.4.4; it also follows easily from the following simple, direct proof: Define an order for turning up *all* cards of the deck, as follows: Obey the rules of the game until getting stuck, then "cheat" by turning up the first available card (find the first pile that is not empty, going clockwise from pile 1) and continue as before, until eventually all cards have been turned up. The cards *in the order of turning up* are in completely random order (since the value of a card need not be specified until after it is turned up). So the problem is just to calculate the probability that in a randomly shuffled deck the last card is a king. More generally the probability that k cards are still face down when the game is over is the probability that the last king in a random shuffle is followed by k cards, namely $4!\binom{51-k}{3}\frac{48!}{52!}$. Hence a man playing this game without cheating will turn up an average of exactly 42.4 cards per game. *Note:* Similarly, it can be shown that the probability

that the player will have to "cheat" k times in the process described above is exactly given by the Stirling number $\left[{13 \atop k+1}\right]/13!$. (See Section 1.2.10, Eq. (9) and Ex. 7; the case of a more general deck is considered in Ex. 18.)

18. (a) If there is a cycle $(V_0, V_1, \ldots, V_k)$, where necessarily $3 \le k \le n$, the sum of the k rows of A corresponding to the k edges of this cycle is a row of zeroes; so if G is not a free tree the determinant of A_0 is zero.

Now if G is a free tree we may regard it as an ordered tree with root V_0, and we can rearrange the rows and columns of A_0 so that columns are in preorder and so that the kth row corresponds to the edge from the kth vertex (column) to its father; then the matrix is triangular with ± 1 in each diagonal position, so the determinant is ± 1.

(b) By the Binet-Cauchy formula (Ex. 1.2.3–46) we have

$$\det A_0^T A_0 = \sum_{1 \le i_1 < \cdots < i_n \le m} (\det A_{i_1 \ldots i_n})^2$$

where $A_{i_1 \ldots i_n}$ represents a matrix consisting of rows $i_1, \ldots, i_n$ of A_0 (thus corresponding to a choice of n edges of G). The result now follows from (a).

19. (a) The conditions $a_{00} = 0$ and $a_{jj} = 1$ are just conditions (a), (b) of the definition of oriented tree. If G is not an oriented tree there is an oriented cycle (by Ex. 7), and this means the rows of A_0 corresponding to the vertices in this oriented cycle sum to a row of zeroes; hence $\det A_0 = 0$. If G is an oriented tree, assign an arbitrary order to the sons of each family and regard G as an ordered tree. Now permute rows and columns of A_0 until they correspond to preorder of the vertices. Since the same permutation has been applied to the rows as to the columns, the determinant is unchanged; and the resulting matrix is triangular with $+1$ in every diagonal position.

(b) We may assume $a_{0j} = 0$ for all j, since no arc emanating from V_0 can participate in an oriented subtree. We may also assume $a_{jj} > 0$ for all $j \ge 1$ since otherwise the whole jth row is zero and there obviously are no oriented subtrees. Now use induction on the number of arcs: If $a_{jj} > 1$ let e be some arc leading from V_j; let B_0 be a matrix like A_0 but with arc e deleted, and let C_0 be the matrix like A_0 but with all arcs *except* e that lead from V_j deleted.
Example: $A_0 = \left(\begin{smallmatrix} 3 & -2 \\ -1 & 2 \end{smallmatrix}\right)$, $j = 1$, $e =$ arc from V_1 to V_0; then $B_0 = \left(\begin{smallmatrix} 2 & -2 \\ -1 & 2 \end{smallmatrix}\right)$, $C_0 = \left(\begin{smallmatrix} 1 & 0 \\ -1 & 2 \end{smallmatrix}\right)$. Then $\det A_0 = \det B_0 + \det C_0$, since the matrices agree in all rows except row j and in that row A_0 is the sum of B_0, C_0. Moreover, the number of oriented subtrees of G is the number of subtrees which do *not* use e (namely, $\det B_0$, by induction) plus the number which *do* use e (namely, $\det C_0$).

20. Using Ex. 18 we find $B = A_0^T A_0$. Or, using Ex. 19, B is the matrix A_0 for the directed graph G' with two arcs (one in each direction) in place of each edge of G; and each free subtree of G corresponds uniquely to an oriented subtree of G' with root V_0 (the directions of the arcs are determined by the choice of root).

21. (This result may be derived from interesting but considerably more complicated arguments used in the paper of van Aardenne-Ehrenfest and de Bruijn quoted in the text. The following derivation is not only simpler, it also may be generalized to determine the number of oriented subtrees of G^* when G is an *arbitrary* directed graph; see D. E. Knuth, *Journal of Combinatorial Theory* **3** (1967), 309–314.)

Construct the matrices A and A^* as in Ex. 19. For the example graphs G, G^* in Figs. 3 and 4,

$$A = \begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & -2 \\ -1 & -1 & 2 \end{pmatrix}$$

$$A^* = \begin{array}{c} \\ [00] \\ [10] \\ [10] \\ [01] \\ [01] \\ [21] \\ [12] \\ [12] \\ [22] \end{array} \begin{array}{c} \begin{array}{ccccccccc} [00] & [10] & [10] & [01] & [01] & [21] & [12] & [12] & [22] \end{array} \\ \left(\begin{array}{ccc|ccc|ccc} 2 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 3 & -1 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & -1 & 0 & 3 & 0 & 0 & -1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 3 & 0 & -1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 3 & -1 & -1 & 0 \\ \hline 0 & 0 & -1 & 0 & 0 & -1 & 3 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 2 \end{array}\right) \end{array}$$

Add the indeterminate λ to the upper left corner element of A and A^* (in the example this gives $2 + \lambda$ in place of 2). If $t(G)$, $t(G^*)$ are the numbers of oriented subtrees of G and G^* we have $t(G) = (1/\lambda)(n+1) \det A$, $t(G^*) = (1/\lambda)m(n+1) \det A^*$. (The number of oriented subtrees of a balanced graph is the same for any given root, e.g. by Ex. 22.)

If we group vertices V_{jk} for equal k the matrix A^* can be partitioned as shown above. Let $B_{kk'}$ be the submatrix of A^* consisting of the rows for V_{jk} and the columns for $V_{j'k'}$, for all j and j' such that V_{jk} and $V_{j'k'}$ are in G^*. By adding the 2nd, . . . , mth columns of each submatrix to the first column and then subtracting the first row of each submatrix from the 2nd, . . . , mth rows, the matrix A^* is transformed so that

$$B_{kk'} = \begin{pmatrix} a_{kk'} & * & \dots & * \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \text{ for } k \neq k', \qquad B_{kk} = \begin{pmatrix} a_{kk} + \lambda\delta_{k0} & * & \dots & * \\ -\lambda\delta_{k0} & m & 0 & 0 \\ \vdots & & \ddots & \vdots \\ -\lambda\delta_{k0} & 0 & 0 \dots & m \end{pmatrix}.$$

Here "*" indicates values which are more or less irrelevant. It follows that det A^* is $m^{m(n-1)}$ times the determinant of

$$\begin{pmatrix} \lambda + a_{00} & * & * & \dots & * & a_{01} & \dots & a_{0m} \\ -\lambda & m & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & & & & & & \\ -\lambda & 0 & 0 & \dots & m & 0 & \dots & 0 \\ a_{10} & * & * & \dots & * & a_{11} & \dots & a_{1n} \\ \vdots & & & & & & & \\ a_{n0} & * & * & \dots & * & a_{n1} & \dots & a_{nn} \end{pmatrix}$$

The asterisks left are all zero except for precisely one -1 in each column. Add the last n rows to the top row, and expand the determinant by the first row, to get $m^{n(m-1)+m-1} \det A - (m-1)m^{n(m-1)+m-2} \det A$.

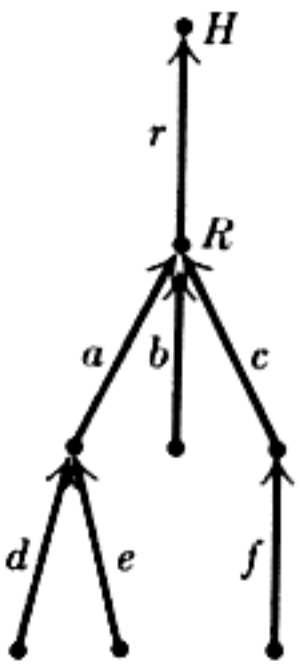
22. The total number is $(\sigma_1 + \cdots + \sigma_n)$ times the number of Eulerian circuits starting with a given edge e_1, where init $(e_1) = V_1$. Each such circuit determines an oriented subtree with root V_1 by Lemma E, and for each of the T oriented subtrees there are $\prod_{1 \le j \le n} (\sigma_j - 1)!$ paths satisfying the three conditions of Theorem D, corresponding to the different order in which the arcs $\{e \mid \text{init}(e) = V_j,\ e \ne e[V_j],\ e \ne e_1\}$ are entered into P. (Cf. Ex. 14.)

23. Construct the directed graph G_k with m^{k-1} vertices as in the hint, and let $[x_1, \ldots, x_k]$ denote the arc mentioned there. For each function that has maximum period length, we can define a unique corresponding Eulerian circuit, by letting $f(x_1, \ldots, x_k) = x_{k+1}$ if arc $[x_1, \ldots, x_k]$ is followed by $[x_2, \ldots, x_{k+1}]$. (We regard Eulerian circuits as being the same if one is just a cyclic permutation of the other.) Now $G_k = G_{k-1}^*$ in the sense of Ex. 21, so G_k has $m^{m^{k-1} - m^{k-2}}$ times as many oriented subtrees as G_{k-1}; by induction G_k has $m^{m^{k-1}-1}$ oriented subtrees, and $m^{m^{k-1}-k}$ with a given root. Therefore by Ex. 22 the number of functions with maximum period, i.e. the number of Eulerian circuits of G_k starting with a given arc, is $m^{-k}(m!)^{m^{k-1}}$.

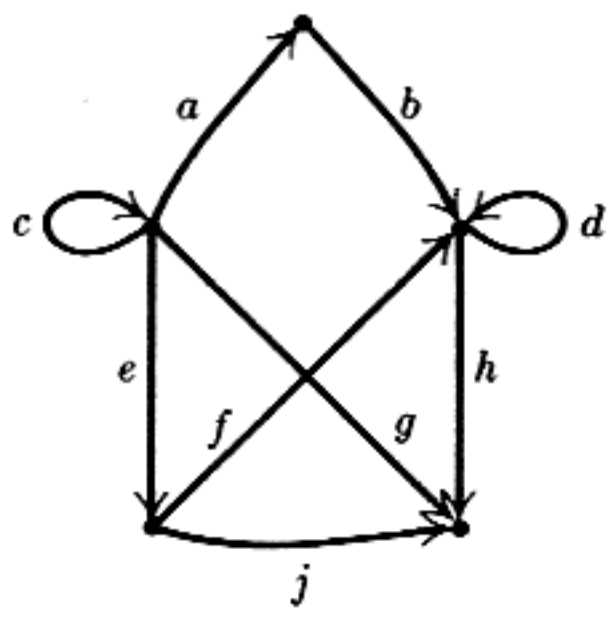
24. Define a new directed graph having E_j copies of e_j, for $0 \le j \le m$. This graph is balanced and so we know there is an Eulerian circuit $(e_0, \ldots)$ by Theorem G. The desired oriented path comes by deleting the edge e_0 from this Eulerian circuit.

25. Assign an order to the sets of arcs having common initial vertices, and assign an order to the sets of arcs having common final vertices. Now for each arc e, let the fields in the node representing e be the following: If e' is the next arc (in the assumed ordering) for which init (e') = init (e), let ALINK point to e' and let ATAG = "+"; if e is the last arc (in the assumed ordering) with this initial vertex, however, let ATAG = "—" and let ALINK be a pointer to the first arc e' for which init (e) = fin (e'); if no such e' exists, let ALINK = Λ. Define BLINK and BTAG by the same rules, reversing the roles of *init* and *fin*.

Examples:



arc	ALINK	ATAG	BLINK	BTAG
a	d	—	b	+
b	Λ	—	c	+
c	f	—	r	—
d	Λ	—	e	+
e	Λ	—	a	—
f	Λ	—	c	—
r	a	—	Λ	—



arc	ALINK	ATAG	BLINK	BTAG
a	c	+	b	—
b	a	—	d	+
c	e	+	a	—
d	h	+	f	+
e	g	+	f	—
f	j	+	h	—
g	c	—	h	+
h	b	—	j	+
j	e	—	Λ	—

Note: If in the oriented tree representation we add another arc from H to itself, we get an interesting situation; either we get the standard conventions 2.3.1–(7) with LLINK, LTAG, RLINK, RTAG *interchanged* in the list head, or (if the new arc is placed last in the ordering) we get the standard conventions except RTAG = "+" in the node associated with the root of the tree!

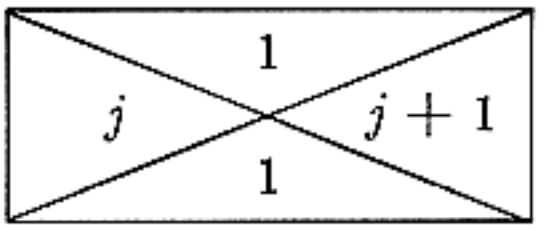
This exercise is based on an idea communicated to the author by W. C. Lynch. It would be interesting to explore further properties of this representation, e.g., to compare tree-traversal algorithms with the Eulerian circuit constructions of this section.

SECTION 2.3.4.3

1. The root is the empty sequence; arcs go from $(x_1, \ldots, x_n)$ to $(x_1, \ldots, x_{n-1})$.

2. Take one domino type and rotate it 180° to get another domino type; these two types give an obvious way to tile the plane (without further rotations) by replication of a 2×2 pattern.

3. Consider the set of domino types



for all positive integers j. Then the upper half plane may be tiled in uncountably many ways; but whatever square is placed in the center of the plane puts a finite limit on the distance it can be continued to the left.

4. Systematically enumerate all possible ways to tile an $n \times n$ block, for $n = 1, 2, \ldots$, looking for toroidal solutions within these blocks. If there is no way to tile the plane, the infinity lemma tells us there is an integer n with no $n \times n$ solutions. If there *is* a way to tile the plane, the assumption tells us there is an n with an $n \times n$ solution containing a rectangle that yields a toroidal solution. Hence in either case the algorithm will terminate. (But the stated assumption is false, as shown in the next exercise, and in fact there is no algorithm which will determine in a finite number of steps whether or not there exists a way to tile the plane with a given set of types.)

5. Start by noticing that we need classes $\begin{smallmatrix}\alpha & \beta \\ \gamma & \delta\end{smallmatrix}$ replicated in 2×2 groups in any solution. Then, step 1: Consider just the α squares; we show that the pattern $\begin{smallmatrix}a & b \\ c & d\end{smallmatrix}$ must be replicated in 2×2 groups of α squares. Step $n > 1$: We determine a pattern that must appear in a cross-shaped region of height and width $2^n - 1$. The middle of the crosses has the pattern $\begin{smallmatrix}Na & Nb \\ Nc & Nd\end{smallmatrix}$ replicated throughout the plane.

For example, after step 3 we will know the contents of 7×7 blocks throughout the plane, separated by unit length strips, every eight units. The 7×7 blocks which are of class Na in the center have the form

αa	βKQ	αb	βQP	αa	βBK	αb
γPJ	δNa	γRB	δQK	γLJ	δNb	γPB
αc	βDS	αd	βQTY	αc	βBS	αd
γPQ	δPJ	γPXB	δNa	γRQ	δRB	γRB
αa	βUK	αb	βDP	αa	βBK	αb
γTJ	δNc	γSB	δDS	γST	δNd	γTB
αc	βQS	αd	βDT	αc	βBS	αd

The middle column and the middle row is the "cross" just filled in during step 3; the other four 3×3 squares were filled in after step 2; the squares just to the right and below this 7×7 square are part of a 15×15 cross to be filled in at step 4.

6. Let k and m be fixed. Consider an oriented tree whose vertices each represent, for some n, one of the partitions of $\{1, \ldots, n\}$ into k parts, containing no arithmetic progression of length m. A node that partitions $\{1, \ldots, n+1\}$ is a son of one for $\{1, \ldots, n\}$ if the two partitions agree on $\{1, \ldots, n\}$. If there were an infinite path from the root we would have a way to divide *all* integers into k sets with no arithmetic progression of length m. Hence, by the infinity lemma and van der Waerden's theorem, this tree is finite. (If $k = 2$, $m = 3$, the tree can be rapidly calculated by hand, and the least value of N is 9.)

7. There exist two sets S_0, S_1 which partition the integers such that neither contains any infinite *computable* sequence (cf. Ex. 3.5–32). So in particular there is no infinite arithmetic progression. Theorem K does not apply because there is no way to put partial solutions into a tree with finite degrees at each vertex.

8. Let a "counterexample sequence" be an infinite sequence of trees that violates Kruskal's theorem, if such sequences exist. Assume the theorem is false; then let T_1 be a tree with the smallest possible number of nodes such that T_1 can be the first tree in a counterexample sequence; if $T_1, \ldots, T_j$ have been chosen, let T_{j+1} be a tree with the smallest possible number of nodes such that $T_1, \ldots, T_j, T_{j+1}$ is the beginning of a counterexample sequence. This process defines a counterexample sequence $T_1, T_2, \ldots$. None of these T's is just a root. Now, we look at this sequence very carefully:

(a) Suppose there is a subsequence $T_{n_1}, T_{n_2}, \ldots$ for which $l(T_{n_1}), l(T_{n_2}), \ldots$ is a counterexample sequence. This is impossible, otherwise $T_1, \ldots, T_{n_1-1}, l(T_{n_1}), l(T_{n_2}), \ldots$ would be a counterexample sequence, contradicting the definition of T_{n_1}.

(b) Because of (a), there are only finitely many j for which $l(T_j)$ cannot be embedded in $l(T_k)$ for any $k > j$. Therefore by taking n_1 larger than any such j we may find a subsequence for which $l(T_{n_1}) \subseteq l(T_{n_2}) \subseteq l(T_{n_3}) \subseteq \cdots$.

(c) Now by the result of Ex. 2.3.2–22, $r(T_{n_j})$ cannot be embedded in $r(T_{n_k})$ for any $k > j$, else $T_{n_j} \subseteq T_{n_k}$. Therefore $T_1, \ldots, T_{n_1-1}, r(T_{n_1}), r(T_{n_2}), \ldots$ is a counterexample sequence. But this contradicts the definition of T_{n_1}. This proves Kruskal's theorem.

Note: Kruskal's theorem does not seem to follow simply from the infinity lemma, although they seem to be related in a vague way; there are in general infinitely many trees T such that $T_1 \not\subseteq T$, $T_2 \not\subseteq T$, $\ldots$, $T_n \not\subseteq T$ when $T_1, T_2, \ldots, T_n$ are given.

SECTION 2.3.4.4

1. $\ln A(z) = \ln z + \sum_{k\geq 1} a_k \ln\left(\frac{1}{1-z^k}\right) = \ln z + \sum_{k,t\geq 1} \frac{a_k z^{kt}}{t} = \ln z + \sum_{t\geq 1} \frac{A(z^t)}{t}.$

2. By differentiation, and equating the coefficients of z^n, we obtain the identity

$$na_{n+1} = \sum_{k\geq 1} \sum_{d\backslash k} d a_d a_{n+1-k}.$$

Now interchange the order of summation.

4. (a) $A(z)$ certainly converges at least for $|z| < \frac{1}{4}$, since a_n is less than the number of *ordered* trees b_{n-1}. Since $A(1)$ is infinite and all coefficients are positive, there is a positive number $\alpha \leq 1$ such that $A(z)$ converges for $|z| < \alpha$, and there is a singularity at $z = \alpha$. Let $\psi(z) = (1/z)A(z)$; since $\psi(z) > e^{z\psi(z)}$, we see $\psi(z) = m$ implies $z < \ln m/m$, so $\psi(z)$ is bounded and $\lim_{z\to\alpha-}\psi(z)$ exists. Thus $\alpha < 1$, and by Abel's limit theorem $a = \alpha \cdot \exp\left(a + \frac{1}{2}A(\alpha^2) + \frac{1}{3}A(\alpha^3) + \cdots\right)$.

(b) $A(z^2), A(z^3), \ldots$ are analytic for $|z| < \sqrt{\alpha}$, and $\frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \cdots$ converges uniformly in a slightly smaller disk.

(c) If $\partial F/\partial w = a - 1 \neq 0$, the implicit function implies there is an analytic function $f(z)$ in a neighborhood of $(\alpha, a/\alpha)$ such that $F(z, f(z)) = 0$. But this implies $f(z) = (1/z)A(z)$, contradicting the fact that $A(z)$ is singular at α.

(d) Obvious.

(e) $\partial F/\partial w = A(z) - 1$ and $|A(z)| < A(\alpha) = 1$ since the coefficients of $A(z)$ are all positive. Hence as in (c), $A(z)$ is regular at all such points.

(f) Near $(\alpha, 1/\alpha)$ we have the identity $0 = \beta(z - \alpha) + (\alpha/2)(w - 1/\alpha)^2 +$ higher order terms, where $w = (1/z)A(z)$; so w is an analytic function of $\sqrt{z - \alpha}$ here by the implicit function theorem. Consequently there is a region $|z| < \alpha_1$ minus a cut $[\alpha, \alpha_1]$ in which $A(z)$ has the stated form. (The minus sign is chosen since a plus sign would make the coefficients ultimately negative.)

(g) Any function of the stated form has coefficients asymptotically

$$\frac{\sqrt{2\beta}}{\alpha^n}\binom{1/2}{n}. \quad \left(\text{Note that } \binom{3/2}{n} = O\left(\frac{1}{n}\binom{1/2}{n}\right).\right)$$

For further details, and asymptotic values of the number of free trees, see the cited article.

5.
$$c_n = \sum_{j_1+2j_2+\cdots=n} \binom{c_1+j_1-1}{j_1}\cdots\binom{c_n+j_n-1}{j_n} - c_n, \qquad n > 1.$$

Therefore

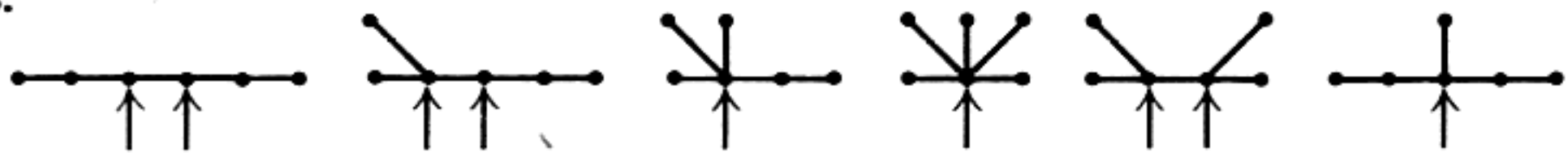
$$\begin{aligned} 2C(z) + 1 - z &= (1-z)^{-c_1}(1-z^2)^{-c_2}(1-z^3)^{-c_3}\cdots \\ &= \exp\left(C(z) + \tfrac{1}{2}C(z^2) + \cdots\right). \end{aligned}$$

We find $C(z) = z + z^2 + 2z^3 + 5z^4 + 12z^5 + 33z^6 + 90z^7 + \cdots$. There is no obvious connection with $A(z)$, although it is plausible some relation does exist.

6. $zG(z)^2 = 2G(z) - 2 - zG(z^2)$; $G(z) = 1 + z + z^2 + 2z^3 + 3z^4 + 6z^5 + 11z^6 + 23z^7 + \cdots$. See *AMM* **56** (1949), 697–699 for references.

7. $g_n \sim ca^n n^{-3/2}$, where $c = .791603$, $a = 2.48325$.

8.



9. If there are two centroids, by considering a path from one to the other we find there can't be intermediate points, so any two centroids are adjacent. It is impossible for a tree to contain three mutually adjacent vertices, so there are at most two.

10. If X, Y are adjacent, let $s(X, Y)$ be the number of vertices in the Y-subtree of X. Then $s(X, Y) + s(Y, X) = n$. The argument in the text shows that if Y is a centroid, $\text{height}(X) = s(X, Y)$. Therefore if both X and Y are centroids, $\text{height}(X) = \text{height}(Y) = n/2$.

In terms of this notation, the argument in the text goes on to show that if $s(X, Y) \geq s(Y, X)$, there is a centroid in the Y subtree of X. So if two free trees with m vertices are joined by an edge between X and Y, we obtain a free tree in which $s(X, Y) = m = s(Y, X)$, and there must be two centroids (namely X and Y).

11. $zT(z)^t = T(z) - 1$; i.e., $z + T(z)^{-t} = T(z)^{1-t}$. By Eq. 1.2.9–21, $T(z) = \sum_n A_n(1, -t)z^n$, so the number of t-ary trees is

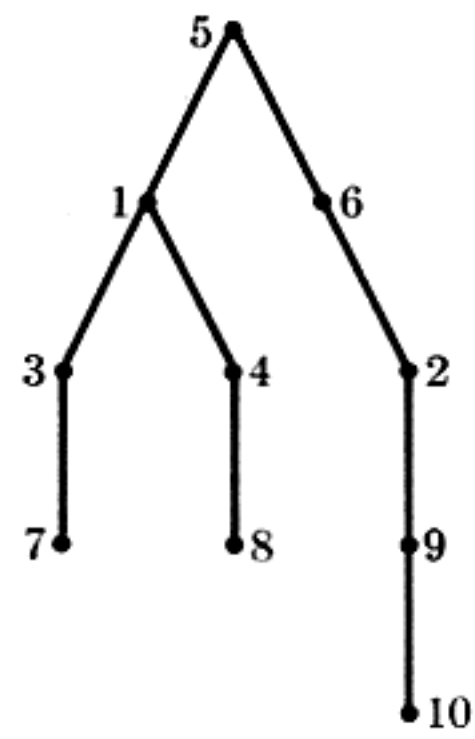
$$\binom{1+tn}{n}\frac{1}{1+tn} = \binom{tn}{n}\frac{1}{(t-1)n+1}.$$

12. Consider the directed graph which has one arc from V_i to V_j for all $i \neq j$. The matrix A_0 of Ex. 2.3.4.2–19 is a combinatorial $(n-1) \times (n-1)$ matrix with $n-1$ on the diagonal and -1 off the diagonal. So its determinant is

$$\big(n + (n-1)(-1)\big)n^{n-2} = n^{n-2},$$

the number of oriented trees with a given root. (Ex. 2.3.4.2–20 could also be used.)

13.

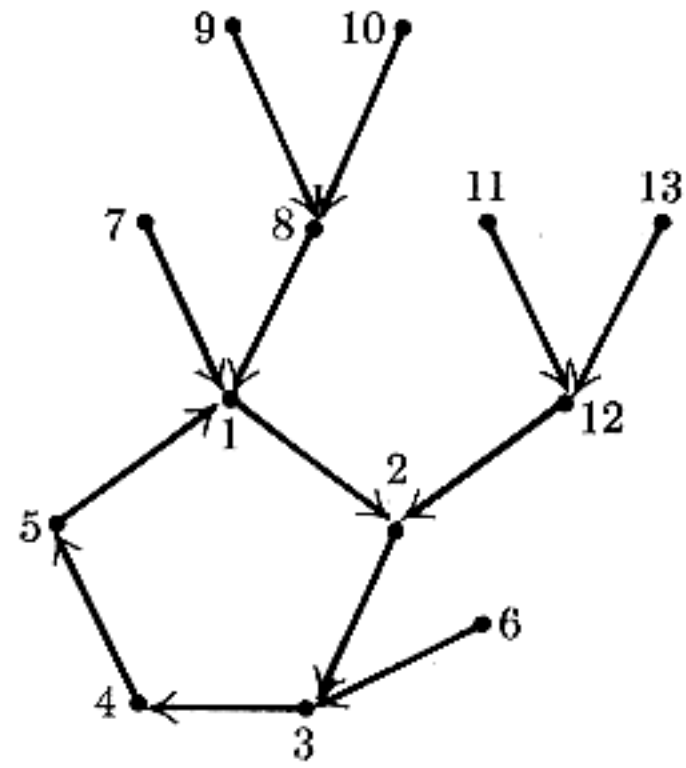


14. True, since the root will not become a leaf until all other branches have been removed.

15. In the canonical representation, $V_1, V_2, \ldots, V_{n-1}, f(V_{n-1})$ is a topological sort of the oriented tree considered as a directed graph, but this order would not in general be output by Algorithm 2.2.3T. Algorithm 2.2.3T can be changed so that it determines the values of $V_1, V_2, \ldots, V_{n-1}$ if the "insert into queue" operation of step T6 is replaced by a procedure which adjusts links so that the entries of the list appear in ascending order from front to rear.

17. There must be exactly one cycle $x_1, x_2, \ldots, x_k$ where $f(x_j) = x_{j+1}$ and $f(x_k) = x_1$. We will enumerate all f having a cycle of length k such that the iterates of each x ultimately come into this cycle. Define the canonical representation $f(V_1), f(V_2), \ldots, f(V_{m-k})$ as in the text; now $f(V_{m-k})$ is in the cycle, so we continue to get a "canonical representation" by writing down the rest of the cycle $f\big(f(V_{m-k})\big), f\big(f(f(V_{m-k}))\big)$, etc.

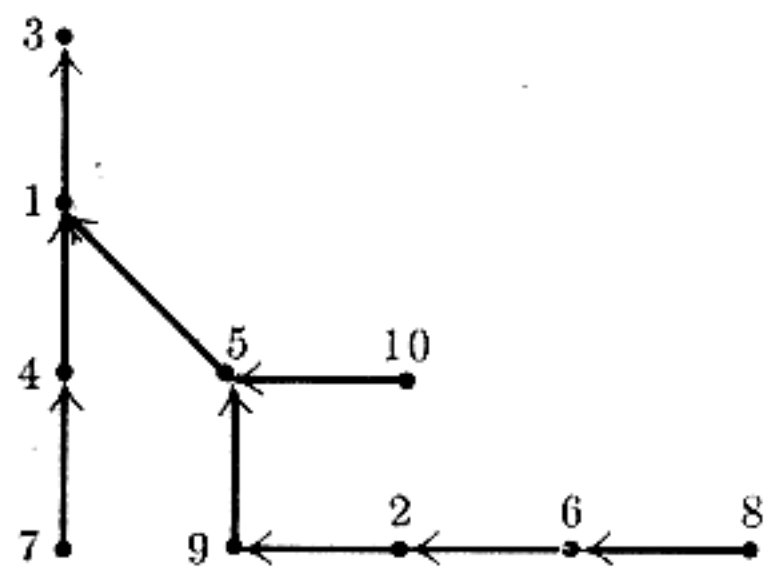
For example, the function with $m = 13$ whose graph is



leads to the representation 3, 1, 8, 8, 1, 12, 12, 2, 3, 4, 5, 1. We obtain a sequence of $m - 1$ numbers in which the last k are distinct. Conversely, from any such sequence we can reverse the construction (assuming k is known), hence there are precisely $m(m-1)\cdots(m-k+1)m^{m-k-1}$ such functions having a k-cycle.

18. To reconstruct the tree from a sequence $s_1, s_2, \ldots, s_{n-1}$, begin with s_1 as the root and successively attach arcs to the tree which point to $s_1, s_2, \ldots$; if vertex s_k has appeared earlier, leave the initial vertex of the arc leading to s_{k-1} nameless, otherwise give this vertex the name s_k. After all $n - 1$ arcs have been placed, give names to all vertices which remain nameless by using the numbers that have not yet appeared, and assigning names in increasing order to nameless vertices in the order of their creation.

For example from 3, 1, 4, 1, 5, 9, 2, 6, 5 we would construct



There is no simple connection between this method and the one in the text. Several more representations are possible; see the article by E. H. Neville, *Proc. Cambridge Phil. Soc.* 49 (1953), 381–385.

19. The canonical representation will have precisely $n - k$ different values, so we enumerate the sequences of $n - 1$ numbers with this property. The answer is $n(n-1)(n-2)\cdots(k+1)\left\{{n-1 \atop n-k}\right\}$. (See Section 3.3.2D.)

20. Consider the canonical representation of such trees. We are asking how many terms of $(x_1 + \cdots + x_n)^{n-1}$ have k_0 exponents zero, k_1 exponents one, etc. This is plainly the coefficient of such a term times the number of such terms, namely $(n-1)!/(0!)^{k_0}(1!)^{k_1}\ldots(n!)^{k_n}$ times $n!/k_0!k_1!\ldots k_n!$.

21. There are none with an even number of vertices; if there are $n = 2m + 1$ vertices, the answer is obtained from Ex. 20 with $k_0 = m + 1$, $k_2 = m$: $\binom{2m+1}{m}(2m)!/2^m$.

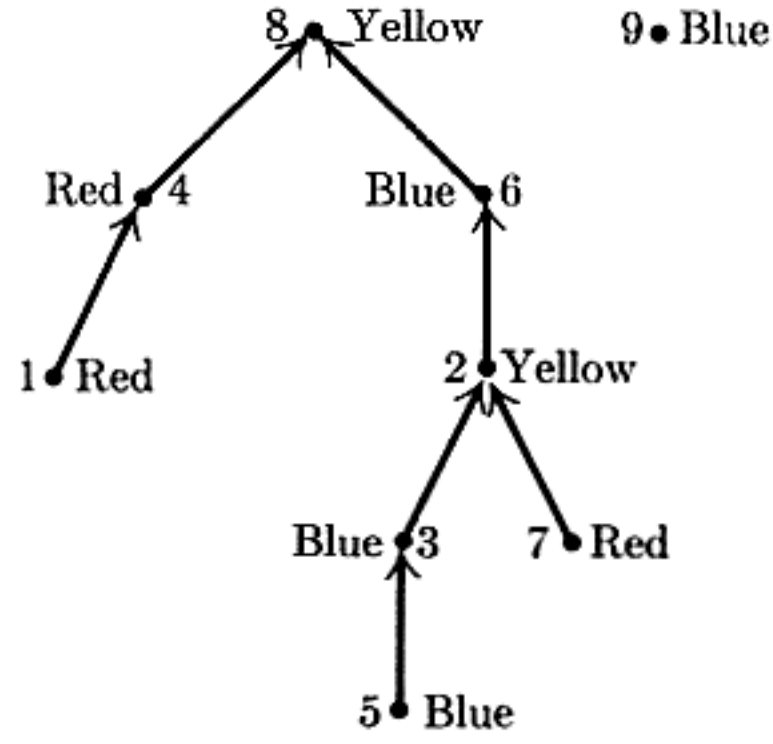
22. Exactly n^{n-2}; for if X is a particular vertex, the free trees are in one-to-one correspondence with oriented trees having root X.

23. It is possible to put the labels on every unlabeled, ordered tree in $n!$ ways, and each of these labeled, ordered trees is distinct. So the total number is $n!b_{n-1} = (2n - 2)!/(n - 1)!$.

24. There are as many with one given root as with another, so the answer in general is $1/n$ times the answer in Ex. 23; and in this particular case the answer is 30.

25. For $0 \leq q < n$, $r(n, q) = (n - q)n^{q-1}$. (Special case $s = 1$ in Eq. (22).)

26.



27. Given a function g from $\{1, 2, \ldots, r\}$ to $\{1, 2, \ldots, q\}$ such that adding arcs from V_k to $U_{g(k)}$ introduces no oriented cycles, construct a sequence $a_1, \ldots, a_r$ as follows: Call vertex V_k "free" if there is no oriented path from V_j to V_k for any $j \neq k$. Since there are no oriented cycles, there must be at least one free vertex. Let b_1 be the smallest integer for which V_{b_1} is free; and assuming $b_1, \ldots, b_t$ have been chosen, let b_{t+1} be the smallest integer different from $b_1, \ldots, b_t$ for which $V_{b_{t+1}}$ is free in the graph obtained by deleting the arcs from V_{b_k} to $U_{g(b_k)}$ for $1 \leq k \leq t$. This rule defines a permutation $b_1, \ldots, b_r$ of the integers $\{1, 2, \ldots, r\}$. Let $a_k = g(b_k)$ for $1 \leq k \leq r$; this defines a sequence such that $1 \leq a_k \leq q$ for $1 \leq k < r$, and $1 \leq a_r \leq p$.

Conversely if such a sequence $a_1, \ldots, a_r$ is given, call a vertex V_k "free" if there is no j for which $a_j > p$ and $f(a_j) = k$. Since $a_r \leq p$ there are at most $r - 1$ non-free vertices. Let b_1 be the smallest integer for which V_{b_1} is free; and assuming $b_1, \ldots, b_t$ have been chosen let b_{t+1} be the smallest integer different from $b_1, \ldots, b_t$ for which $V_{b_{t+1}}$ is free with respect to the sequence $a_{t+1}, \ldots, a_r$. This rule defines a permutation $b_1, \ldots, b_r$ of the integers $\{1, 2, \ldots, r\}$. Let $g(b_k) = a_k$ for $1 \leq k \leq r$; this defines a function such that adding arcs from V_k to $U_{g(k)}$ introduces no oriented cycles.

28. Let f be any of the n^{m-1} functions from $\{2, \ldots, m\}$ to $\{1, 2, \ldots, n\}$, and consider the directed graph with vertices $U_1, \ldots, U_m, V_1, \ldots, V_n$ and arcs from U_k to $V_{f(k)}$ for $1 < k \leq m$. Apply Ex. 27 with $p = 1$, $q = m$, $r = n$, to show there are m^{n-1} ways to add further arcs from the V's to the U's to obtain an oriented tree with root U_1. Since there is a one-to-one correspondence between the desired set of

free trees and the set of oriented trees with root U_1, the answer is $n^{m-1}m^{n-1}$. [*Note:* This construction can be extensively generalized; see D. Knuth, *Canadian J. Math.* **20** (1968), 1077–1086.]

29. If $y = x^t$, then $(tz)y = \ln y$, and we see it is sufficient to prove the identity for $t = 1$. Now if $zx = \ln x$ we know by Ex. 25 that $x^m = \sum_k E_k(m, 1)z^k$ for nonnegative integers m. Now

$$x^r = e^{zxr} = \sum_k \frac{(zxr)^k}{k!} = \sum_{j,k} \frac{r^k z^{k+j} E_j(k, 1)}{k!j!} = \sum_{j,k} \frac{z^k}{k!}\binom{k}{j} E_j(k-j, 1)r^{k-j}$$

$$= \sum_k \frac{z^k}{k!} \sum_j \binom{k-1}{j} k^j r^{k-j} = \sum_k z^k E_k(r, 1).$$

30. Each graph described defines a set $C_x \subseteq \{1, \ldots, n\}$, where j is in C_x if and only if there is a path from t_j to r_i for some $i \le x$. For a given C_x each graph described is composed of two independent parts: one of the $x(x + \epsilon_1 z_1 + \cdots + \epsilon_n z_n)^{\epsilon_1 + \cdots + \epsilon_n - 1}$ graphs on the vertices r_i, s_{jk}, t_j for $i \le x$ and $j \in C_x$, where $\epsilon_j = 1$ iff $j \in C_x$, plus one of the $y(y + (1 - \epsilon_1)z_1 + \cdots + (1 - \epsilon_n)z_n)^{(1-\epsilon_1)+\cdots+(1-\epsilon_n)-1}$ graphs on the remaining vertices.

31. $G(z) = z + G(z)^2 + G(z)^3 + G(z)^4 + \cdots = z + G(z)^2/(1 - G(z))$. Hence $G(z) = \frac{1}{4}(1 + z - \sqrt{1 - 6z + z^2})$. [*Note:* It is perhaps remarkable that these numbers for $n > 1$ are just half the values obtained in Ex. 2.2.1–11, i.e. half the number of permutations on n elements obtainable with a restricted deque. Another problem equivalent to this one was posed and solved by Schröder, *Zeitschrift für Mathematik* **15** (1870), 361–376, who determined the number of ways to insert nonoverlapping diagonals in a convex $(n + 1)$-gon. Can the reader find any connection between this problem and Ex. 2.2.1–11, to explain why the answers are essentially the same?]

32. Zero if $n_0 \ne 1 + n_2 + 2n_3 + 3n_4 + \cdots$ (cf. Ex. 2.3–21), otherwise

$$(n_0 + n_1 + \cdots + n_m - 1)!/n_0!n_1! \ldots n_m!.$$

To prove this result we recall that an unlabeled tree with $n = n_0 + n_1 + \cdots + n_m$ nodes is characterized by the sequence $d_1\, d_2 \ldots d_n$ of the degrees of the nodes in postorder (Section 2.3.3). Furthermore such a sequence of degrees corresponds to a tree if and only if $\sum_{1 \le j \le k} (1 - d_j) > 0$ for $0 < k \le n$. (This important property of Polish notations is readily proved by induction; cf. Algorithm 2.3.3F with f a function that creates a tree, like the TREE function of Section 2.3.2.) In particular, d_1 must be 0. The answer to our problem is therefore the number of sequences $d_2 \ldots d_n$ with n_j occurrences of j for $j > 0$, namely the multinomial coefficient

$$\binom{n-1}{n_0 - 1, n_1, \ldots, n_m},$$

minus the number of such sequences $d_2 \ldots d_n$ for which $\sum_{2 \le j \le k} (1 - d_j) < 0$ for some $k \ge 2$.

We may enumerate the latter sequences as follows: Let t be minimal such that $\sum_{2 \le j \le t} (1 - d_j) < 0$; then $\sum_{2 \le j \le t} (1 - d_j) = -s$ where $1 \le s < d_t$, and we may form the subsequence $d_2' \ldots d_n' = d_{t-1} \ldots d_2 0 d_{t+1} \ldots d_n$ which has n_j occurrences

of j for $j \neq d_t$, $n_j - 1$ occurrences of j for $j = d_t$. Now $\sum_{2 \le j \le k} (1 - d'_j)$ is equal to d_t when $k = n$, equal to $d_t - s$ when $k = t$, and less than $d_t - s$ when $k < t$. (To prove the latter statement, note that

$$\sum_{2 \le j \le k} (1 - d'_j) = \sum_{2 \le j < t} (1 - d_j) - \sum_{2 \le j \le t-k} (1 - d_j) \le \sum_{2 \le j < t} (1 - d_j)$$
$$= d_t - s - 1.)$$

It follows that, given s and any sequence $d'_2 \ldots d'_n$, the construction can be reversed; hence the number of sequences $d_2 \ldots d_n$ which have a given value of d_t and s is the multinomial coefficient

$$\binom{n-1}{n_0, \ldots, n_{d_t} - 1, \ldots, n_m}.$$

The number of sequences $d_2 \ldots d_n$ which correspond to trees is therefore obtained by summing over the possible values of d_t and s:

$$\sum_{0 \le j \le m} (1 - j)\binom{n-1}{n_0, \ldots, n_j - 1, \ldots, n_m} = \frac{(n-1)!}{n_0! n_1! \ldots n_m!} \sum_{0 \le j \le m} (1 - j) n_j$$

and the latter sum is 1.

An even simpler proof of this result has been given by G. N. Raney (*Transactions of the American Math. Society* **94** (1960), 441–451). If $d_1 d_2 \ldots d_n$ is any sequence with n_j appearances of j, there is precisely one cyclic rearrangement $d_k \ldots d_n d_1 \ldots d_{k-1}$ that corresponds to a tree, namely the rearrangement where k is maximal such that $\sum_{1 \le j \le k} (1 - d_j)$ is minimal.

Either of the methods above can be generalized to show that the number of (ordered, unlabelled) forests having f trees and n_j nodes of degree j is $(n-1)!f/n_0! n_1! \ldots! n_m!$, provided the necessary condition $n_0 = f + n_2 + 2n_3 + \cdots$ is satisfied. This result was independently proved by L. Takács, whose paper is cited in the answer to Ex. 2.2.1–4.

33. Consider the number of trees with n_1 nodes labelled 1, n_2 nodes labelled 2, . . . , and such that each node labelled j has degree e_j. Let this number be $c(n_1, n_2, \ldots)$, with the specified degrees $e_1, e_2, \ldots$ regarded as fixed. The generating function $G(z_1, z_2, \ldots) = \sum c(n_1, n_2, \ldots) z_1^{n_1} z_2^{n_2} \ldots$ satisfies the identity $G = z_1 G^{e_1} + \cdots + z_r G^{e_r}$, since $z_j G^{e_j}$ enumerates those trees whose root is labelled j. And by the result of the previous exercise,

$$c(n_1, n_2, \ldots) = \begin{cases} \dfrac{(n_1 + n_2 + \cdots - 1)!}{n_1! n_2! \ldots}, & \text{if } (1 - e_1)n_1 + (1 - e_2)n_2 + \cdots = 1; \\ 0, & \text{otherwise.} \end{cases}$$

More generally, since G^f enumerates the number of ordered forests having such labels, we have for integer $f > 0$

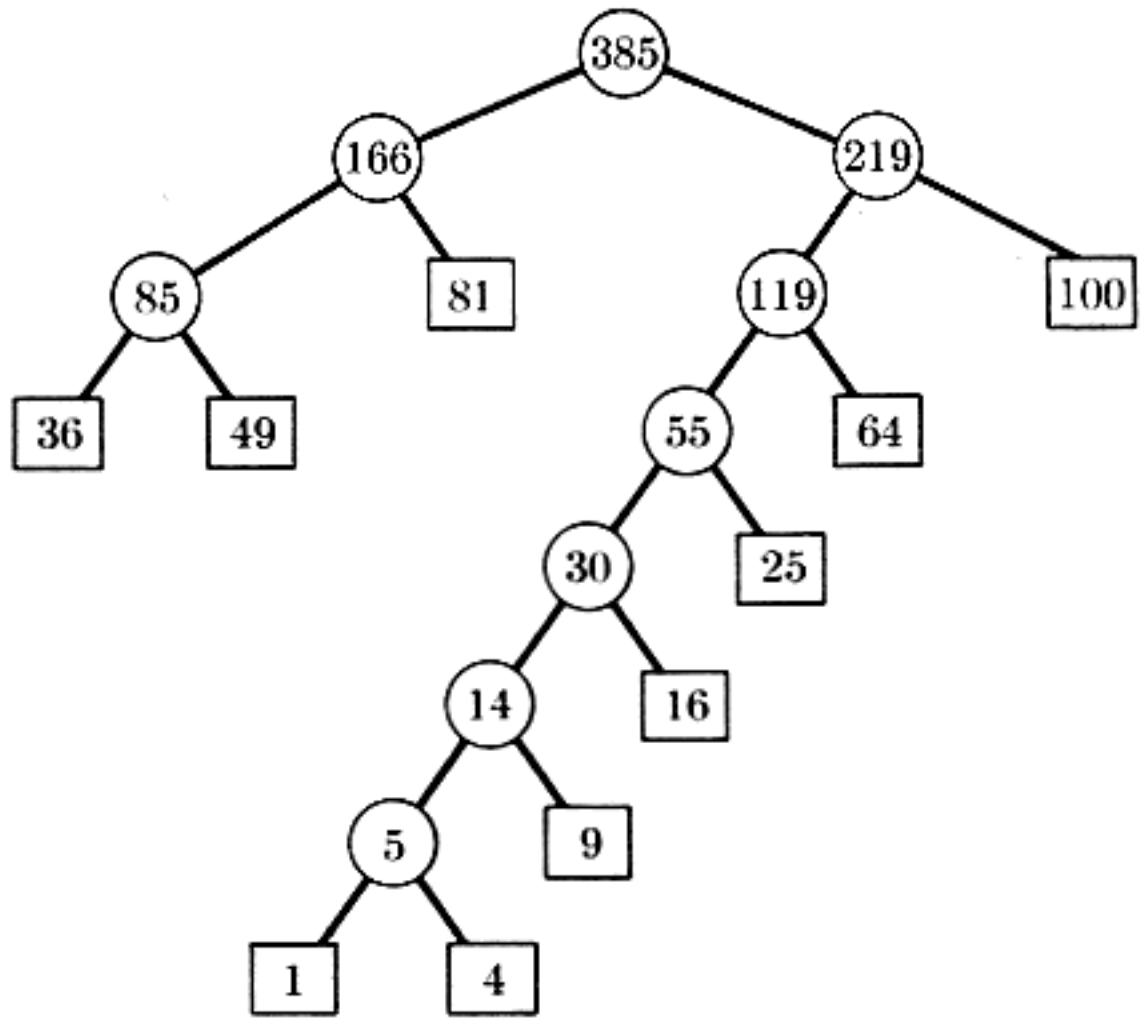
$$w^f = \sum_{f = (1-e_1)n_1 + (1-e_2)n_2 + \ldots} \frac{(n_1 + n_2 + \cdots - 1)! f}{n_1! n_2!} z_1^{n_1} z_2^{n_2} \ldots$$

These formulas are meaningful when $r = \infty$, and they are essentially equivalent to "Lagrange's inversion formula."

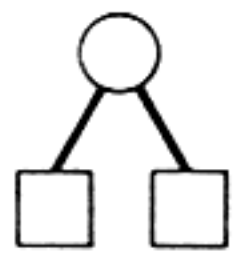
SECTION 2.3.4.5

1. Yes, there are $\binom{8}{5}$ in all, since the nodes numbered 8, 9, 10, 11, 12 may be attached in any of eight positions below 4, 5, 6, and 7.

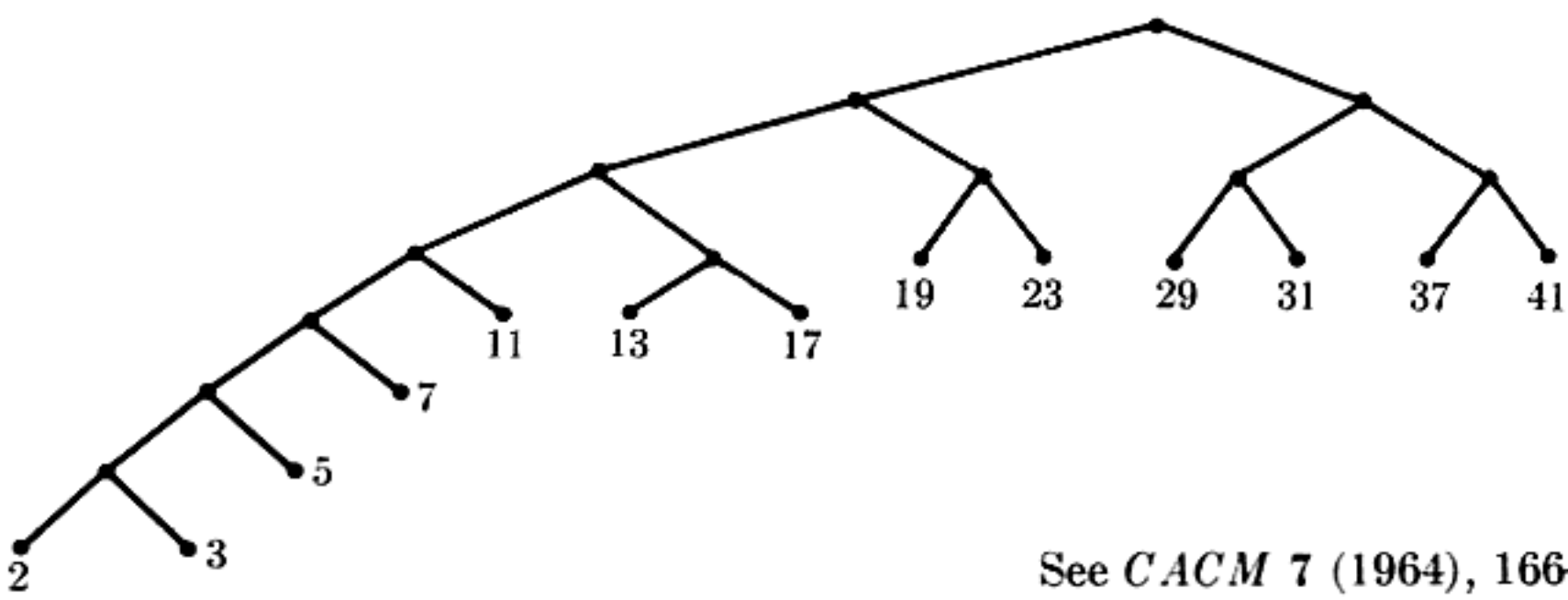
2.



3. By induction on m, the condition is necessary. Conversely if $\sum_{1\le j\le m} 2^{-l_j} = 1$, we want to construct an extended binary tree with these path lengths. If $m = 1$, then $l_1 = 0$ and the construction is trivial. Otherwise we may assume the l's are ordered so that $l_1 = l_2 = \cdots = l_q > l_{q+1} \ge l_{q+2} \ge \cdots \ge l_m > 0$ for some q with $1 \le q \le m$. Now $2^{l_1-1} = \sum_{1\le j\le m} 2^{l_1-l_j-1} = \frac{1}{2}q +$ integer, hence q is even. By induction on m there is a tree with path lengths $l_1 - 1, l_3, l_4, \ldots, l_m$; take such a tree and replace one of the terminals at distance $l_1 - 1$ from the root by



4. First, find a tree by Huffman's method. If $w_j < w_{j+1}$, then $l_j > l_{j+1}$, or else the tree would not be optimal. The construction in the answer to Ex. 3 now gives us another tree with these same path lengths and with the weights in the proper sequence. For example, the tree (11) becomes



See *CACM* **7** (1964), 166–169.

5. (a)
$$b_{np} = \sum_{\substack{k+l=n-1 \\ r+s+n-1=p}} b_{kr} b_{ls}.$$

Hence $zB(w, wz)^2 = B(w, z) - 1$. (b) Take the partial derivative with respect to w:

$$2zB(w, wz)\big(B_w(w, wz) + zB_z(w, wz)\big) = B_w(w, z).$$

Therefore if $H(z) = B_w(1, z) = \sum_n h_n z^n$, we find $H(z) = 2zB(z)\big(H(z) + zB'(z)\big)$; and the known formula for $B(z)$ implies

$$H(z) = \frac{1}{1-4z} - \frac{1}{z}\left(\frac{1-z}{\sqrt{1-4z}} - 1\right),$$

so

$$h_n = 4^n - \frac{3n+1}{n+1}\binom{2n}{n}.$$

The average value is h_n/b_n. (c) Asymptotically, this comes to $n\sqrt{\pi n} - 3n + O(\sqrt{n})$.

For the solution to similar problems, see John Riordan, *IBM J. Res. and Devel.* **4** (1960), 473–478; and A. Rényi and G. Szekeres, "On the height of trees," *J. Australian Math. Soc.* **7** (1967), 497–507.

6. $n + s - 1 = tn$.

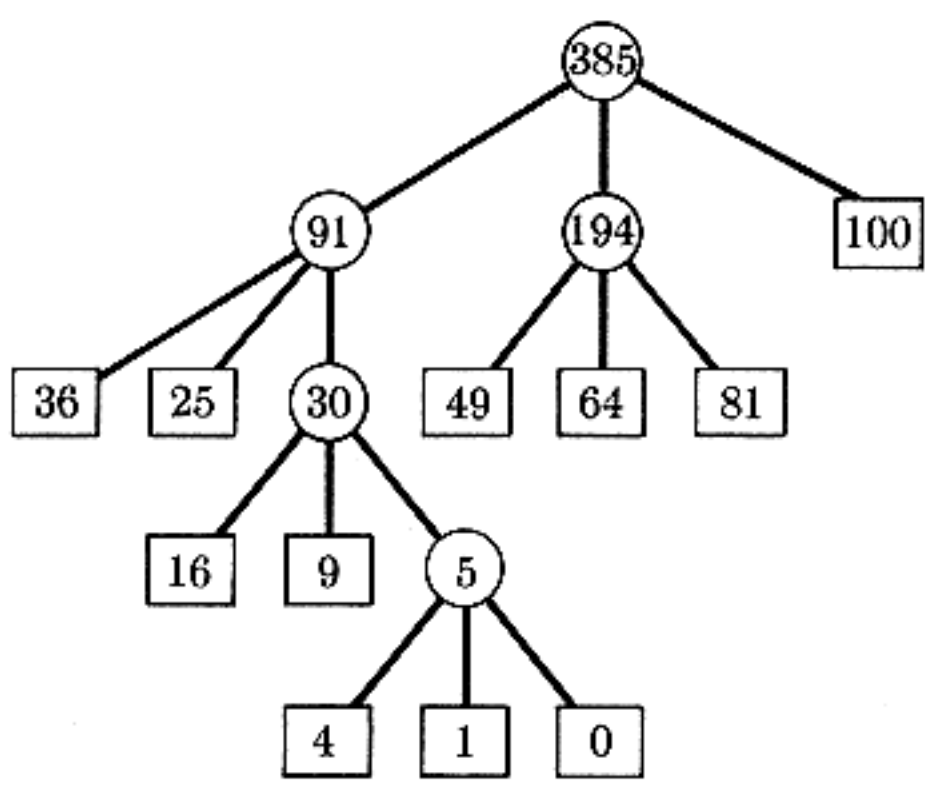
7. $E = (t-1)I + tn$.

8. $\displaystyle\sum_{1 \le k \le n} \lfloor \log_t ((t-1)k) \rfloor = \text{(summation by parts) } nq - \sum_{\substack{0 \le k \le n \\ \exists j, (t-1)k+1 = t^j}} k.$

The latter sum may be rewritten $\sum_{1 \le j \le q} (t^j - 1)/(t-1)$.

9. Induction on the size of the tree.

10. By adding extra *zero* weights, if necessary, we may assume that $m \bmod (t-1) = 1$. To obtain a t-ary tree with minimum weighted path length, combine the smallest t values at each step and replace them by their sum. The proof is essentially the same as the binary case. The desired ternary tree is



11. The "Dewey" notation is the binary representation of the node number.

SECTION 2.3.5

1. A List structure is a directed graph in which the arcs leaving each vertex are ordered, and some of the vertices which have no arcs leaving (out-degree 0) are designated "atoms." Furthermore there is a vertex S such that there is an oriented path from S to V for all vertices $V \neq S$.

2. Not in the same way, since thread links in the usual representation lead back to "FATHER" which is not unique for sub-Lists. However, the representation discussed in Ex. 2.3.4.2–25 can perhaps be used (although this idea has not yet been exploited at the time of writing).

3. If only PO is to be marked, the algorithm certainly operates correctly. If $n > 1$ nodes are to be marked, then note that ATOM(PO) = 0. Step E4 then sets ALINK(PO) ← Λ and executes the algorithm with PO replaced by ALINK(PO) and T replaced by PO. By induction (note that since MARK(PO) is now 1, all links to PO are equivalent to Λ by steps E4 and E5), we see that ultimately we will mark all nodes on paths that start with ALINK(PO) and do not pass through PO; and we will then get to step E6 with T = PO and P = ALINK(PO). Now since ATOM(T) = 1, step E6 restores ALINK(PO) and ATOM(PO) and we reach step E5. Step E5 sets BLINK(PO) ← Λ, etc., and a similar argument shows that we will ultimately mark all nodes on paths that start with BLINK(PO) and do not pass through PO or nodes reachable from ALINK(PO). Then we will get to E6 with T = PO, P = BLINK(PO), and finally we get to E6 with T = Λ, P = PO.

4. The program which follows incorporates the suggested improvements in the speed of processing atoms which appear in the text after the statement of Algorithm E.

In steps E4 and E5 of the algorithm, we want to test if MARK(Q) = 0. If NODE(Q) = +0, this is an unusual case which can be properly handled by setting it to −0 and treating it as if it were originally −0, since it has ALINK and BLINK both Λ. This simplification is not reflected in the timing calculations below.

rI1 ≡ P, rI2 ≡ T, rI3 ≡ Q, rX ≡ −1 (for setting MARKs)

```
01    MARK   EQU   (0:0)
02    ATOM   EQU   (1:1)
03    ALINK  EQU   (2:3)
04    BLINK  EQU   (4:5)
05    E1     LD1   PO            1      E1. Initialize. P ← PO.
06           ENT2  0             1      T ← Λ.
07           ENTX  -1            1      rX ← −1.
08    E2     STX   0,1(MARK)     1      E2. Mark. MARK(P) ← "−".
09    E3     LDA   0,1(ATOM)     1      E3. Atom?
10           JAZ   E4            1      Jump if ATOM(P) = 0.
11    E6     J2Z   DONE          n      E6. Up.
12           ENT3  0,2           n − 1  Q ← T.
13           LDA   0,3(ATOM)     n − 1
14           JANZ  1F            n − 1  Jump if ATOM(T) = 1.
15           LD2   0,3(BLINK)    t2     T ← BLINK(Q).
16           ST1   0,3(BLINK)    t2     BLINK(Q) ← P.
17           ENT1  0,3           t2     P ← Q.
18           JMP   E6            t2
```

19	1H	STZ	0,2(ATOM)	$t1$	ATOM(T) ← 0.
20		LD2	0,3(ALINK)	$t1$	T ← ALINK(Q).
21		ST1,	0,3(ALINK)	$t1$	ALINK(Q) ← P.
22		ENT1	0,3	$t1$	P ← Q.
23	E5	LD3	0,1(BLINK)	n	*E5. Down* BLINK. Q ← BLINK(P).
24		J3Z	E6	n	Jump if Q = Λ.
25		LDA	0,3	$n - b2$	
26		STX	0,3(MARK)	$n - b2$	MARK(Q) ← "—".
27		JANP	E6	$n - b2$	Jump if NODE(Q) was already marked.
28		LDA	0,3(ATOM)	$t2 + a2$	
29		JANZ	E6	$t2 + a2$	Jump if ATOM(Q) = 1.
30		ST2	0,1(BLINK)	$t2$	BLINK(P) ← T.
31	E4A	ENT2	0,1	$n - 1$	T ← P.
32		ENT1	0,3	$n - 1$	P ← Q.
33	E4	LD3	0,1(ALINK)	n	*E4. Down* ALINK. Q ← ALINK(P).
34		J3Z	E5	n	Jump if Q = Λ.
35		LDA	0,3	$n - b1$	
36		STX	0,3(MARK)	$n - b1$	MARK(Q) ← "—".
37		JANP	E5	$n - b1$	Jump if NODE(Q) was already marked.
38		LDA	0,3(ATOM)	$t1 + a1$	
39		JANZ	E5	$t1 + a1$	Jump if ATOM(Q) = 1.
40		STX	0,1(ATOM)	$t1$	ATOM(P) ← 1.
41		ST2	0,1(ALINK)	$t1$	ALINK(P) ← T.
42		JMP	E4A	$t1$	T ← P, P ← Q, to E4. ▮

By Kirchhoff's law, $t1 + t2 + 1 = n$. The total time is $(34n + 4t1 + 3a - 5b - 8)u$, where n is the number of non-atomic nodes marked, a is the number of atoms marked, b is the number of Λ links encountered in marked non-atomic nodes, and $t1$ is the number of times we went down an ALINK ($0 \le t1 < n$).

5. (The following is the 'fastest known' marking algorithm.)

S1. Set MARK(PO) ← 1. If ATOM(PO) = 1, the algorithm terminates; otherwise set S ← 0, R ← PO, T ← Λ.

S2. Set P ← BLINK(R). If P = Λ or MARK(P) = 1, go to S3. Otherwise set MARK(P) ← 1. Now if ATOM(P) = 1, go to S3; otherwise if S < N set S ← S + 1, STACK[S] ← P, and go to S3; otherwise set F ← 0 and go to S5.

S3. Set P ← ALINK(R). If P = Λ or MARK(P) = 1, go to S4. Otherwise set MARK(P) ← 1. Now if ATOM(P) = 1, go to S4; otherwise if S < N set S ← S + 1, STACK[S] ← P, and go to S4; otherwise set F ← 1 and go to S5.

S4. If S = 0, terminate the algorithm; otherwise set R ← STACK[S], S ← S − 1, and go to S2.

S5. Set Q ← ALINK(P). If Q = Λ or MARK(Q) = 1, go to S6. Otherwise set MARK(Q) ← 1. Now if ATOM(Q) = 1, go to S6; otherwise set ATOM(P) ← 1, ALINK(P) ← T, T ← P, P ← Q, go to S5.

S6. Set Q ← BLINK(P). If Q = Λ or MARK(Q) = 1, go to S7. Otherwise set MARK(Q) ← 1. Now if ATOM(Q) = 1, go to S7; otherwise set BLINK(P) ← T, T ← P, P ← Q, go to S5.

S7. If T = Λ, go to S3 if F = 0, to S4 if F = 1. Otherwise set Q ← T. If ATOM(Q) = 1, set ATOM(Q) ← 0, T ← ALINK(Q), ALINK(Q) ← P, P ← Q, and return to S6. If ATOM(Q) = 0, set T ← BLINK(Q), BLINK(Q) ← P, P ← Q, and return to S7. ▮

Reference: *CACM* **10** (1967), 501–506.

6. From the second phase of garbage collection (or perhaps also the initial phase if all mark bits are set to zero at this time).

7. Delete steps E2 and E3, and delete "ATOM(P) ← 1" in E4. Set MARK(P) ← 1 in step E5 and use "MARK(Q) = 0", "MARK(Q) = 1" in step E6 in place of the present "ATOM(Q) = 1", "ATOM(Q) = 0" respectively. The idea is to set the MARK bit only after the left subtree has been marked. *This algorithm works even if the tree has overlapping (shared) subtrees,* but it does not work for all recursive List structures such as those with NODE(ALINK(Q)) an ancestor of NODE(Q). (Note that ALINK of a marked node is never changed.)

8. Solution 1: Analogous to Algorithm E, but simpler.

F1. Set T ← Λ, P ← PO.

F2. Set MARK(P) ← 1, and set P ← P + SIZE(P).

F3. If MARK(P) = 1, go to F5.

F4. Set Q ← LINK(P). If Q ≠ Λ and MARK(Q) = 0, set LINK(P) ← T, T ← P, P ← Q, and go to F2.

F5. If T = Λ, stop. Otherwise set Q ← T, T ← LINK(Q), LINK(Q) ← P, P ← Q − 1, and return to F3. ▮

Solution 2: Analogous to Algorithm D. For this solution, we assume the SIZE field is large enough to contain a link address. Such an assumption is probably not justified by the statement of the problem, but it lets us use a slightly faster method than the first solution when it is applicable.

G1. Set T ← Λ, MARK(PO) ← 1, P ← PO + SIZE(PO).

G2. If MARK(P) = 1, go to G5.

G3. Set Q ← LINK(P), P ← P − 1.

G4. If Q ≠ Λ and MARK(Q) = 0, set MARK(Q) ← 1, S ← SIZE(Q), SIZE(Q) ← T, T ← Q + S. Go back to G2.

G5. If T = Λ, stop. Otherwise set P ← T and find the first value of Q = P, P − 1, P − 2, . . . for which MARK(Q) = 1; set T ← SIZE(Q) and SIZE(Q) ← P − Q. Go back to G2. ▮

9. H1. Set L ← 0, K ← M + 1, MARK(0) ← 1, MARK(M + 1) ← 0.

H2. Increase L by one, and if MARK(L) = 1 repeat this step.

H3. Decrease K by one, and if MARK(K) = 0 repeat this step.

H4. If L > K, go to step H5; otherwise set NODE(L) ← NODE(K), ALINK(K) ← L, MARK(K) ← 0, and return to H2.

H5. For L = 1, 2, . . . , K do the following: Set MARK(L) ← 0.
If ATOM(L) = 0 and ALINK(L) > K, set ALINK(L) ← ALINK(ALINK(L)).
If ATOM(L) = 0 and BLINK(L) > K, set BLINK(L) ← ALINK(BLINK(L)). ▮

10. **Z1.** [Initialize.] Set F ← P0, R ⇐ AVAIL, NODE(R) ← NODE(F), REF(F) ← R. (Here F and R are pointers for a queue set up in the REF fields of all header nodes encountered.)

Z2. [Begin new List.] Set P ← F, Q ← REF(P).

Z3. [Advance to right.] Set P ← RLINK(P). If P = Λ, go to Z6.

Z4. [Copy one node.] Set Q1 ⇐ AVAIL, RLINK(Q) ← Q1, Q ← Q1, NODE(Q) ← NODE(P).

Z5. [Translate sub-List link.] If T(P) = 1, set P1 ← REF(P), and if REF(P1) = Λ set REF(R) ← P1, R ⇐ AVAIL, REF(P1) ← R, NODE(R) ← NODE(P1), REF(Q) ← R. If T(P) = 1 and REF(P1) ≠ Λ, set REF(Q) ← REF(P1). Go to Z3.

Z6. [Move to next List.] Set RLINK(Q) ← Λ. If REF(F) ≠ R, set F ← REF(REF(F)) and return to Z2. Otherwise set REF(R) ← Λ, P ← P0.

Z7. [Final cleanup.] Set Q ← REF(P). If Q ≠ Λ, set REF(P) ← Λ and P ← Q and repeat step Z7. ▮

11. Here is a pencil-and-paper method which can be written out more formally to answer the problem: First attach a unique name (e.g. a capital letter) to each List in the given set; in the example we would have for example $A = (a{:}C, b, a{:}F)$, $F = (b{:}D)$, $B = (a{:}F, b, a{:}E)$, $C = (b{:}G)$, $G = (a{:}C)$, $D = (a{:}F)$, $E = (b{:}G)$. Now make a list of pairs of List names that must be proved equal. Successively add pairs to this list until either a contradiction is found because we have a pair which disagree on the first level (then the originally given Lists are unequal), or until the list of pairs does not imply any further pairs (then the originally given Lists are equal). In the example, this list of pairs would originally contain only the given pair, AB; then it gets the further pairs CF, EF (by matching A and B), DG (from CF) and then we have a self-consistent set.

To prove the validity of this method, observe that (a) if it returns the answer "unequal", the given Lists are unequal; (b) if the given Lists are unequal, it returns the answer "unequal"; (c) it always terminates.

12. When the AVAIL list contains N nodes, where N is a specified constant to be chosen as discussed below, initiate another coroutine which shares computer time with the main routine and does the following: (a) Marks all N nodes on the AVAIL list; (b) Marks all other nodes which are accessible to the program; (c) links all unmarked nodes together to prepare a new AVAIL list for use when the current AVAIL list is empty, and (d) resets the mark bits in all nodes. One must choose N and the ratio of time sharing so there is a positive guarantee that operations (a), (b), (c), and (d) are complete before N nodes are taken from the AVAIL list, yet the main routine is running sufficiently fast. It is necessary to use some care in step (b) to make sure all nodes "accessible to the program" are included, as the program continues to run; details are omitted here. If the list formed in (c) has less than N nodes, it may be necessary to stop eventually because memory space might become exhausted.

SECTION 2.4

1. Preorder.
2. Essentially proportional to the number of Data Table entries created.

3. Change step A5 to: "A5′. [Remove top level.] Remove the top stack entry; and if the new level number at the top of the stack is $\geq L$, let (L1, P1) be the new entry at the top of the stack and repeat this step. Otherwise set BROTHER(P1) ← Q and then let (L1, P1) be the new entry at the top of the stack."

4. Rule (c) is violated if and only if there is a data item whose *complete qualification* A_0 OF . . . OF A_n is also a COBOL reference to some other data item. Therefore Algorithm A would be extended to check, as each new data item is added to the Data Table, whether its complete qualification is contained in some previous reference of the same name, or whether the complete qualification of some previous item of the same name is also a reference to this new data item. This means essentially that Algorithm B is appended to Algorithm A in the appropriate ways.

5. Make the following changes:

Step	*replace*	*by*
B1.	P ← LINK(P_0)	P ← LINK(INFO(T))
B2.	$k \leftarrow 0$	K ← T
B3.	$k < n$	RLINK(K) $\neq \Lambda$
B4.	$k \leftarrow k + 1$	K ← RLINK(K)
B6.	NAME(S) = P_k	NAME(S) = INFO(K)

6. A simple modification of Algorithm B makes it search only for complete references (if $k = n$ and FATHER(S) $\neq \Lambda$ in step B3, or if NAME(S) $\neq P_k$ in step B6, set P ← PREV(P) and go to B2). The idea is to run through this modified Algorithm B first; then, if Q is *still* Λ, perform the unmodified algorithm.

7. MOVE MONTH OF DATE OF SALES TO MONTH OF DATE OF PURCHASES. MOVE DAY OF DATE OF SALES TO DAY OF DATE OF PURCHASES. MOVE YEAR OF DATE OF SALES TO YEAR OF DATE OF PURCHASES. MOVE ITEM OF TRANSACTION OF SALES TO ITEM OF TRANSACTION OF PURCHASES. MOVE QUANTITY OF TRANSACTION OF SALES TO QUANTITY OF TRANSACTION OF PURCHASES. MOVE PRICE OF TRANSACTION OF SALES TO PRICE OF TRANSACTION OF PURCHASES. MOVE TAX OF TRANSACTION OF SALES TO TAX OF TRANSACTION OF PURCHASES.

8. If and only if α or β is an elementary item. (It may be of interest to note that the author failed to handle this case properly in his first draft of Algorithm C, and it actually made the algorithm more complicated.)

9. "MOVE CORRESPONDING α TO β", if neither α nor β is elementary, is equivalent to the set of statements "MOVE CORRESPONDING A OF α TO A OF β" taken over all names A common to groups α and β. (This is a more elegant way to state the definition than the more traditional and more cumbersome definition of "MOVE CORRESPONDING" given in the text.) We may verify that Algorithm C satisfies this definition, using an inductive proof that steps C2 through C5 will ultimately terminate with P = P0 and Q = Q0. Further details of the proof are filled in as we have done many times before in a "tree induction" (cf. the proof of Algorithm 2.3.1T).

10. (a) Set S1 ← LINK(P_k). Then repeatedly set S1 ← PREV(S1) zero or more times until either S1 = Λ (NAME(S) $\neq P_k$) or S1 = S (NAME(S) = P_k). (b) Set P1 ← P and then set P1 ← PREV(P1) zero or more times until PREV(P1) = Λ; do a similar operation with variables Q1, Q; and then test if P1 = Q1. Alternatively if the Data

Table entries are ordered so that PREV(P) < P for all P, a faster test can be made in an obvious way depending on whether P > Q or not and following the PREV links of the larger to see if the smaller is encountered.

11. A miniscule improvement in the speed of step C4 would be achieved by adding a new link field BROTHER1(P) ≡ SON(FATHER(P)). More significantly, we could modify the SON and BROTHER links so that NAME(BROTHER(P)) > NAME(P); this would speed up the search in step C3 considerably because it would require only one pass over each family to find the matching members. This would therefore remove the only "search" present in Algorithms B or C. Algorithms A and C are readily modified for this interpretation, and the reader may find this an interesting exercise. (However, if we consider the relative frequency of MOVE CORRESPONDING statements and the usual size of family groups, the resulting speedup will not be terribly significant in the translation of actual COBOL programs.)

12. Leave steps B1, B2, B3 unchanged; change the other steps thus:

B4. Set $k \leftarrow k + 1$, R ← LINK(P_k).

B5. If R = Λ, there is no match; set P ← PREV(P) and go to B2. If R ≤ S ≤ SCOPE(R), go to B3. Otherwise set R ← PREV(R) and repeat step B5. ▮

This algorithm does *not* adapt to the PL/I convention of Ex. 6.

13. Use the same algorithm, minus the operations that set NAME, FATHER, SON, and BROTHER. Whenever removing the top stack entry in step A5, set SCOPE(L1) ← Q − 1. When the input is exhausted in step A2, simply set L ← 0 and continue, then terminate the algorithm if L = 0 in step A7.

14. The following algorithm, using an auxiliary stack (cf. Chapter 8), has steps numbered to show a direct correspondence with the text's algorithm.

C1. Set P ← P0, Q ← Q0, and set the stack contents empty.

C2. If SCOPE(P) = P or SCOPE(Q) = Q, output (P, Q) as one of the desired pairs and go to C5. Otherwise put (P, Q) on the stack and set P ← P + 1, Q ← Q + 1.

C3. Determine if P and Q point to entries with the same name (cf. Ex. 10(b)). If so, go to C2. If not, let (P1, Q1) be the entry at the top of the stack; if SCOPE(Q) < SCOPE(Q1), set Q ← SCOPE(Q) + 1 and repeat step C3.

C4. Let (P1, Q1) be the entry at the top of the stack. If SCOPE(P) < SCOPE(P1), set P ← SCOPE(P) + 1, Q ← Q1 + 1, and go back to C3. If SCOPE(P) = SCOPE(P1), set P ← P1, Q ← Q1 and remove the top entry of the stack.

C5. If the stack is empty, the algorithm terminates. Otherwise go to C4. ▮

SECTION 2.5

1. In such fortuitous circumstances, a stack-like operation may be used as follows: Let the memory pool area be locations 0 through M − 1, and let AVAIL point to the lowest free location. To reserve N words, report failure if AVAIL + N ≥ M, otherwise set AVAIL ← AVAIL + N. To free these N words, just set AVAIL ← AVAIL − N.

2. The amount of storage space for an item of length l is $k\lceil l/(k-b)\rceil$, which has the average value $kL/(k-b) + (1-\alpha)k$, where α is assumed to be $\frac{1}{2}$, independent

of k. This expression is a minimum (for real values of k) when $k = b + \sqrt{2bL}$. So choose k to be the integer just above or just below this value, whichever gives the lowest value of $kL/(k - b) + \frac{1}{2}k$. For example if $b = 1$ and $L = 10$, $k \approx 1 + \sqrt{20} =$ 5 or 6; both are equally good. For much greater detail about this problem, see *JACM* **12** (1965), 53–70.

4. rI1 ≡ Q, rI2 ≡ P.

```
A1    LDA    N            rA ← N.
      ENT2   AVAIL        P ← LOC(AVAIL).
A2A   ENT1   0,2          Q ← P.
A2    LD2    0,1(LINK)    P ← LINK(Q).
      J2N    OVERFLOW     If P = Λ, no room.
A3    CMPA   0,2(SIZE)
      JG     A2A          Jump if N > SIZE(P).
A4    SUB    0,2(SIZE)    rA ← N − SIZE(P) ≡ K.
      JANZ   *+3          Jump if K ≠ 0.
      LDX    0,2(LINK)    LINK(P)
      STX    0,1(LINK)    → LINK(Q).
      STA    0,2(SIZE)    SIZE(P) ← K.
      LD1    0,2(SIZE)    Optional ending,
      INC1   0,2          sets rI1 ← P + K. ▮
```

5. Probably not. The unavailable storage area just before location P will subsequently become available, and its length will be increased by the amount K; an increase of 99 would not be negligible.

6. The idea is to try to search in different parts of the AVAIL list each time. We can use a "roving pointer," called ROVER for example, which is treated as follows: In step A1, set Q ← ROVER. After step A4, set ROVER ← LINK(Q). In step A2, when P = Λ the first time during a particular execution of Algorithm A, set Q ← LOC(AVAIL) and repeat step A2. When P = Λ the *second* time, the algorithm terminates unsuccessfully. In this way ROVER will tend to point to a random spot in the AVAIL list, and the sizes will be more balanced. At the beginning of the program, set ROVER ← Λ; it is *also* necessary to set ROVER to Λ everywhere else in the program where the block whose address equals the current setting of ROVER is taken out of the AVAIL list.

7. 2000, 1000 with requests of sizes 800, 1200.

8. In step A1, also set R ← Λ. In step A2, if P = Λ go to A6. In step A3, go to A5 not A4. Add new steps as follows:

A5. [Better fit?] If R = Λ or M > SIZE(P), set R ← Q and M ← SIZE(P). Then set Q ← P and return to A2.

A6. [Any found?] If R = Λ, the algorithm terminates unsuccessfully. Otherwise set Q ← R, P ← LINK(Q), and go to A4. ▮

9. Obviously if we are so lucky as to find SIZE(P) = N, we have a "best fit" and it is not necessary to search farther. (When there are only very few different block sizes, this occurs rather often.) If a "boundary tag" method like in Algorithm C is being used, it is possible to maintain the AVAIL list in sorted order, so the length of search could be cut down to $\frac{1}{2}$ the length of the list or less, on the average; more generally, we could in this case break the AVAIL list into several separate lists for

different size ranges, or even make it into a pseudo-balanced tree structure as described in Chapter 6.

10. Make the following changes:

Step B2, for "P > PO" read "P ≥ PO".

Step B3, insert "If PO + N > P (and P ≠ Λ), set P ← LINK(P) and repeat step B3."

Step B4, for "Q + SIZE(Q) = PO", read "Q + SIZE(Q) ≥ PO"; and for "SIZE (Q) ← SIZE(Q) + N" read "SIZE(Q) ← PO + N − Q".

11. If ROVER ≠ Λ and PO > ROVER, we can set Q ← ROVER instead of Q ← LOC(AVAIL) in step B1. If there are n entries in the AVAIL list, the average number of iterations of step B2 is $(2n+3)(n+2)/6(n+1) = \frac{1}{3}n + \frac{5}{6} + O(\frac{1}{n})$. For example if $n = 2$ we get 9 equally probable situations:

	PO < AVAIL	AVAIL < PO < LINK(AVAIL)	LINK(AVAIL) < PO
ROVER=AVAIL	1 //	1	2
ROVER=LINK(AVAIL)	1	2 //	1
ROVER=Λ	1	2	3

The average is $\frac{1}{9}\left(\binom{2}{2} + \binom{3}{2} + \binom{4}{2} + \binom{3}{2} + \binom{2}{2}\right) = \frac{1}{9}\left(\binom{5}{3} + \binom{4}{3}\right) = \frac{14}{9}$.

12. **A1.** Set P ← ROVER, F ← 0.

A2. If P = LOC(AVAIL) and F = 0, set P ← AVAIL, F ← 1, and repeat step A2. If P = LOC(AVAIL) and F ≠ 0, the algorithm terminates unsuccessfully.

A3. If SIZE(P) ≥ N, go to A4; otherwise set P ← LINK(P) and return to A2.

A4. Set ROVER ← LINK(P), K ← SIZE(P) − N. If K < c (where c is a constant which must equal 2 or more), set LINK(LINK(P + 1)) ← ROVER, LINK(ROVER + 1) ← LINK(P + 1), L ← P; otherwise set L ← P + K, SIZE(P) ← SIZE(L − 1) ← K, TAG(L − 1) ← "−", SIZE(L) ← N. Set TAG(L) ← TAG(L + SIZE(L) − 1) ← "+". ▮

13. rI1 ≡ P, rX ≡ F, rI2 ≡ L.

```
LINK    EQU    4:5
SIZE    EQU    1:2
TSIZE   EQU    0:2
TAG     EQU    0:0
A1      LDA    N              rA ← N.
        SLA    3              Shift into SIZE field.
        ENTX   0              F ← 0.
        LD1    ROVER          P ← ROVER.
        JMP    A2
A3      CMPA   0,1(SIZE)
        JLE    A4             Jump if N ≤ SIZE(P).
        LD1    0,1(LINK)      P ← LINK(P).
A2      ENT2   -AVAIL,1       rI2 ← P — LOC(AVAIL).
        J2NZ   A3
        JXNZ   OVERFLOW       Is F ≠ 0?
        ENTX   1              Set F ← 1.
```

```
        LD1    AVAIL(LINK)   P ← AVAIL.
        JMP    A2
A4      LD2    0,1(LINK)
        ST2    ROVER         ROVER ← LINK(P).
        LDA    0,1(SIZE      rA ≡ K ← SIZE(P) — N.
        SUB    N
        CMPA   =C=
        JGE    1F            Jump if K ≥ c.
        LD3    1,1(LINK)     rI3 ← LINK(P + 1).
        ST2    0,3(LINK)     LINK(rI3) ← ROVER.
        ST3    1,2(LINK)     LINK(ROVER + 1) ← rI3.
        ENT2   0,1           L ← P.
        LD3    0,1(SIZE)     rI3 ← SIZE(P).
        JMP    2F
1H      STA    0,1(SIZE)     SIZE(P) ← K.
        LD2    0,1(SIZE)
        INC2   0,1           L ← P + K.
        LDAN   0,1(SIZE)     rA ← —K.
        STA    -1,2(TSIZE)   SIZE(L — 1) ← K, TAG(L — 1) ← "—".
        LD3    N             rI3 ← N.
2H      ST3    0,2(TSIZE)    TAG(L) ← "+", also set SIZE(L) ← rI3.
        INC3   0,2
        STZ    -1,3(TAG)     TAG(L + SIZE(L) — 1) ← "+". ▮
```

14. (a) This field is needed to locate the beginning of the block, in step C2. It could be replaced (perhaps to advantage) by a link to the first word of the block. (b) This field is needed because it is necessary to reserve more than N words at times (for example if $K = 1$), and the amount reserved must be present when the block is subsequently freed.

15, 16. rI1 ≡ P0, rI2 ≡ P1, rI3 ≡ F, rI4 ≡ B, rI6 ≡ —N.

```
D1  LD1    P0            D1.
    LD2    0,1(SIZE)
    ENN6   0,2           N ← SIZE(P0).
    INC2   0,1           P1 ← P0 + N.
    LD5    0,2(TSIZE)
    J5N    D4            To D4 if TAG(P1) = "—".
D2  LD5    -1,1(TSIZE)   D2.
    J5N    D7            To D7 if TAG(P0 — 1) = "—".
D3  LD3    AVAIL(LINK)   D3. Set F ← AVAIL.
    ENT4   AVAIL         B ← LOC(AVAIL).
    JMP    D5            To D5.
D4  INC6   0,5           D4. N ← N + SIZE(P1).
    LD3    0,2(LINK)     F ← LINK(P1).
    LD4    1,2(LINK)     B ← LINK(P1 + 1).
    CMP2   ROVER         (This part because of the ROVER
    JNE    *+3           feature of Ex. 12:
    ENTX   AVAIL         If P1 = ROVER,
    STX    ROVER              set ROVER ← LOC(AVAIL).)
```

```
     DEC2  0,5            P1 ← P1 + SIZE(P1).
     LD5   -1,1(TSIZE)
     J5N   D6             To D6 if TAG(P0 − 1) = "−".
D5   ST3   0,1(LINK)      D5. LINK(P0) ← F.
     ST4   1,1(LINK)      LINK (P0 + 1) ← B.
     ST1   1,3(LINK)      LINK (F + 1) ← P0.
     ST1   0,4(LINK)      LINK(B) ← P0.
     JMP   D8             To D8.
D6   ST3   0,4(LINK)      D6. LINK(B) ← F.
     ST4   1,3(LINK)      LINK(F + 1) ← B.
D7   INC6  0,5            D7. N ← N + SIZE(P0 − 1).
     INC1  0,5            P0 ← P0 − SIZE(P0 − 1).
D8   ST6   0,1(TSIZE)     D8. SIZE(P0) ← N, TAG(P0) ← "−".
     ST6   -1,2(TSIZE)    SIZE(P1 − 1) ← N, TAG (P1− 1) ← "−". ▮
```

17. Both LINK fields equal to LOC(AVAIL).

18. Algorithm A reserves the upper end of a large block. When storage is completely available, the first fit method actually begins by reserving the high-order locations, but once these become available again they are not re-reserved since a "fit" is usually found already in the lower locations; thus the initial large block at the lower end of memory quickly disappears with "first fit." A large block rarely is the "best fit," however, so the best fit method leaves a large block at the beginning of memory.

19. Use the algorithm of Ex. 12, except delete the references to SIZE (L− 1), TAG(L − 1), and TAG(L + SIZE(L) − 1) from step A4; also insert the following actions at the beginning of step A3: "Set P1 ← P + SIZE(P). If TAG(P1) = "−", set LINK(LINK(P1) + 1) ← LINK(P1 + 1), LINK(LINK(P1 + 1)) ← LINK(P1), SIZE(P) ← SIZE(P) + SIZE(P1), and repeat step A3. Otherwise:"

Clearly the situation of (2), (3), (4) can't occur here; the only real effect on the storage allocation is that the search here will tend to be longer than in Ex. 12, and sometimes K will be less than c although there is really another available block preceding this one that we do not know about.

(An alternative is to take the collapsing out of the inner loop A3, and to do the collapsing only in step A4 before the final allocation or in the inner loop when the algorithm would otherwise have terminated unsuccessfully. This alternative requires a simulation study to see if it is an improvement or not.)

20. When a buddy is found to be available, during the collapsing loop, we want to remove that block from its AVAIL[k] list, but we do not know which links to update unless (a) we do a possibly long search, or (b) the list is doubly-linked.

21. If $n = 2^k\alpha$, where $1 \le \alpha \le 2$, a_n is $2^{2k+1}(\alpha - \frac{2}{3}) + \frac{1}{3}$, and b_n is $2^{2k-1}\alpha^2 + 2^{k-1}\alpha$. The ratio a_n/b_n for large n is essentially $4(\alpha - \frac{2}{3})/\alpha^2$, which takes its minimum value $\frac{4}{3}$ when $\alpha = 1$ and 2, and its maximum value $\frac{3}{2}$ when $\alpha = 1\frac{1}{3}$. So a_n/b_n approaches no limit, it oscillates between these two extremes.

22. This idea requires a TAG field in several words of the 11-word block, not only in the first word. It is a workable idea, provided these extra TAG bits can be spared, and it would appear to be especially suitable for use in computer hardware.

23. 011011110100; 011011100000.

24. This introduces a bug in the program; we may get to step S1 when TAG(0) = 1, since S2 may return to S1. To make it work, add "TAG(L) ← 0" after "L ← P" in step S2. (It is easier to assume instead that TAG(2^m) = 0.)

25. The idea is quite correct. (Note that criticism need not be negative.) The list heads AVAIL[k] may be eliminated for $n < k \le m$; the algorithms of the text may be used if "m" is changed to "n" in steps R1, S1. The initial conditions (13), (14) should be changed to indicate 2^{m-n} blocks of size 2^n instead of one block of size 2^m.

26. Using the binary representation of M, we can easily modify the initial conditions (13), (14) so that all memory locations are divided into blocks whose size is a power of two. In Algorithm S, TAG(P) should be regarded as 0 whenever P $\ge$ M.

27. rI1 $\equiv k$, rI2 $\equiv j$, rI3 $\equiv j - k$, rI4 $\equiv$ L, LOC(AVAIL[j]) = AVAIL + j; assume that there is an auxiliary table TWO[j] = 2, stored in location TWO + j, for $0 \le j \le m$. Assume further that TAG = +, − represents TAG = 0, 1; TAG(LOC(AVAIL[j])) = "−", except, as a sentinel, TAG(LOC(AVAIL[$m + 1$])) = "+".

```
00    KVAL    EQU    5:5
01    TAG     EQU    0:0
02    LINKF   EQU    1:2
03    LINKB   EQU    3:4
04    TLNKF   EQU    0:2
05    R1      LD1    K                    1      R1. Find block.
06            ENT2   0,1                  1      j ← k.
07            ENT3   0                    1
08            LD4    AVAIL,2(LINKF)       1
09    1H      ENT5   AVAIL,2              1 + R
10            DEC5   0,4                  1 + R
11            J5NZ   R2                   1 + R  Jump if AVAILF[j] ≠ LOC(AVAIL[j]).
12            INC2   1                    R      Increase j
13            INC3   1                    R
14            LD4N   AVAIL,2(TLNKF)       R
15            J4NN   1B                   R      Is j ≤ m?
16            JMP    OVERFLOW
17    R2      LD5    0,4(LINKF)           1      R2. Remove from list.
18            ST5    AVAIL,2(LINKF)       1      AVAILF[j] ← LINKF(L).
19            ENTA   AVAIL,2              1
20            STA    0,5(LINKB)           1      LINKB(L) ← LOC(AVAIL[j]).
21            STZ    0,4(TAG)             1      TAG(L) ← 0.
22    R3      J3Z    DONE                 1      R3. Split required?
23    R4      DEC3   1                    R      R4. Split.
24            DEC2   1                    R      Decrease j.
25            LD5    TWO,2                R      rI5 ≡ P
26            INC5   0,4                  R      P ← L + 2^j.
27            ENNA   AVAIL,2              R
28            STA    0,5(TLNKF)           R      TAG(P) ← 1, LINKF(P) ← LOC(AVAIL[j]).
29            STA    0,5(LINKB)           R      LINKB(P) ← LOC(AVAIL[j]).
30            ST5    AVAIL,2(LINKF)       R      AVAILF[j] ← P.
31            ST5    AVAIL,2(LINKB)       R      AVAILB[j] ← P.
32            ST2    0,5(KVAL)            R      KVAL(P) ← j.
33            J3P    R4                   R      Go to R3.
34    DONE    ...                                ∎
```

28. $rI1 \equiv k$, $rI5 \equiv$ P, $rI4 \equiv$ L; assume TAG (2^m) = "+".

```
01  S1  LD4   L                  1       S1. Is buddy available?
02      LD1   K                  1
03  1H  ENTA  0,4              1 + S
04      XOR   TWO,1            1 + S     rA ← buddy_k(L).
05      STA   TEMP             1 + S
06      LD5   TEMP             1 + S     P ← rA.
07      LDA   0,5              1 + S
08      JANN  S3               1 + S     Jump if TAG(P) = 0.
09      CMP1  0,5(KVAL)        B + S
10      JNE   S3               B + S     Jump if KVAL(P) ≠ k.
11  S2  LD2   0,5(LINKF)           S     S2. Combine with buddy.
12      LD3   0,5(LINKB)           S
13      ST3   0,2(LINKF)           S     LINKF(LINKB(P)) ← LINKF(P).
14      ST2   0,3(LINKB)           S     LINKB(LINKF(P)) ← LINKB(P).
15      INC1  1                    S     Increase k.
16      CMP4  TEMP                 S
17      JL    1B                   S
18      ENT4  0,5                  A     If L > P, set L ← P.
19      JMP   1B                   A
20  S3  LD2   AVAIL,1(LINKF)       1     S3. Put on list.
21      ENNA  AVAIL,1              1
22      STA   0,4(0:4)             1     TAG(L) ← 1,LINKB(L) ← LOC(AVAIL[k]).
23      ST2   0,4(LINKF)           1     LINKF(L) ← AVAILF[k].
24      ST1   0,4(KVAL)            1     KVAL(L) ← k.
25      ST4   0,2(LINKB)           1     LINKB(AVAILF[k]) ← L.
26      ST4   AVAIL,1(LINKF)       1     AVAILF[k] ← L.   ▮
```

29. Yes, but only at the expense of some searching, or (better) an additional table of TAG bits packed somehow. (It is tempting to suggest that buddies not be joined together during Algorithm S, but only in Algorithm R if there is no block large enough to meet the request; but this probably leads to a badly fragmented memory.)

33. G1. [Clear LINKs.] Set P ← 1, and repeat the operation LINK(P) ← Λ, P ← P + SIZE(P) until P = AVAIL. (This merely sets the LINK field in the first word of each node to Λ; we may assume in most cases that this step is unnecessary, since LINK(P) is set to Λ in step G9 below and it can be set to Λ by the storage allocator.)

G2. [Initialize marking phase.] Set TOP ← USE, LINK(TOP) ← AVAIL, LINK(AVAIL) ← Λ. (TOP points to the top of a stack as in Algorithm 2.3.5D.)

G3. [Pop up stack.] Set P ← TOP, TOP ← LINK(TOP). If TOP = Λ, go to G5.

G4. [Put new links on stack.] For $1 \le k \le$ T(P), do the following operations: Set Q ← LINK(P + k), and if Q ≠ Λ, LINK(Q) = Λ set LINK(Q) ← TOP, TOP ← Q.

G5. [Initialize next phase.] (Now P = AVAIL, and the marking phase has been completed so that the first word of each accessible node has a nonnull LINK. Now we wish to combine adjacent inaccessible nodes, for speed in later

steps, and to assign new addresses to the accessible ones.) Set Q ← 1, LINK(AVAIL) ← Q, SIZE(AVAIL) ← 0, P ← 1. (Location AVAIL is being used as a sentinel to signify the end of a loop in subsequent phases.)

G6. [Assign new addresses.] If LINK(P) = Λ, go to G7. Otherwise if SIZE(P) = 0, go to G8. Otherwise set LINK(P) ← Q, Q ← Q + SIZE(P), P ← P + SIZE(P), and repeat this step.

G7. [Collapse available areas.] If LINK(P + SIZE(P)) = Λ, increase SIZE(P) by SIZE(P + SIZE(P)) and repeat this step. Otherwise set P ← P + SIZE(P) and return to G6.

G8. [Translate all links.] (Now the LINK field in the first word of each accessible node contains the address to which the node will be moved.) Set USE ← LINK(USE), and AVAIL ← Q. Then set P ← 1, and repeat the following operation until SIZE(P) = 0: If LINK(P) ≠ Λ, set LINK(Q) ← LINK(LINK(Q)) for P < Q ≤ P + T(P); then regardless of the value of LINK(P), set P ← P + SIZE(P).

G9. [Move.] Set P ← 1, and repeat the following operation until SIZE(P) = 0: Set Q ← LINK(P), and if Q ≠ Λ set LINK(P) ← Λ and NODE(Q) ← NODE(P); then whether Q = Λ or not, set P ← P + SIZE(P). (The operation NODE(Q) ← NODE(P) implies the movement of SIZE(P) words; we always have Q ≤ P, so it is safe to move the words in order from smallest location to largest.)

[This method is called the "LISP 2 garbage collector." Another, somewhat more complicated compacting algorithm has been described by B. K. Haddon and W. M. Waite, *Comp. J.* **10** (1967), 162–165.]

34. Let TOP ≡ rI1, Q ≡ rI2, P ≡ rI3, k ≡ rI4, SIZE(P) ≡ rI5. Assume further that Λ = 0, and LINK(0) ≠ 0 to simplify step G4. Step G1 is omitted.

```
01    LINK  EQU   4:5
02    INFO  EQU   0:3
03    SIZE  EQU   1:2
04    T     EQU   3:3
05    G2    LD1   USE          1      G2. Initialize marking phase. TOP ← USE.
06          LD2   AVAIL        1
07          ST2   0,1(LINK)    1      LINK (TOP) ← AVAIL.
08          STZ   0,2(LINK)    1      LINK(AVAIL) ← Λ.
09    G3    ENT3  0,1        a + 1    G3. Pop up stack. P ← TOP.
10          LD1   0,1(LINK)  a + 1    TOP ← LINK(TOP).
11          J1Z   G5         a + 1    To G5 if TOP = Λ.
12    G4    LD4   0,3(T)       a      G4. Put new links on stack. k ← T(P).
13    1H    J4Z   G3         b + a    k = 0?
14          INC3  1            b      P ← P + 1.
15          DEC4  1            b      k ← k − 1.
16          LD2   0,3(LINK)    b      Q ← LINK(P).
17          LDA   0,2(LINK)    b
18          JANZ  1B           b      Jump if LINK(Q) ≠ Λ.
19          ST1   0,2(LINK)  a − 1    Otherwise set LINK(Q) ← TOP,
20          ENT1  0,2        a − 1        TOP ← Q.
21          JMP   1B         a − 1
22    G5    ENT2  1            1      G5. Initialize next phase. Q ← 1.
23          ST2   0,3          1      LINK(AVAIL) ← 1, SIZE(AVAIL) ← 0.
```

24		ENT3	1	1	P ← 1.
25		JMP	G6	1	
26	1H	ST2	0,3(LINK)	a	LINK(P) ← Q.
27		INC2	0,5	a	Q ← Q + SIZE(P).
28		INC3	0,5	a	P ← P + SIZE(P).
29	G6	LDA	0,3(LINK)	$a+1$	*G6. Assign new addresses.*
30	G6A	LD5	0,3(SIZE)	$a+c+1$	
31		JAZ	G7	$a+c+1$	Jump if LINK(P) = Λ.
32		J5NZ	1B	$a+1$	Jump if SIZE(P) ≠ 0.
33	G8	LD1	USE	1	*G8. Translate all links.*
34		LDA	0,1(LINK)	1	
35		STA	USE	1	USE ← LINK(USE).
36		ST2	AVAIL	1	AVAIL ← Q.
37		ENT3	1	1	P ← 1.
38		JMP	G8P	1	
39	1H	LD6	0,6(SIZE)	d	
40		INC5	0,6	d	rI5 ← rI5 + SIZE(P + SIZE(P)).
41	G7	ENT6	0,3	$c+d$	*G7. Collapse available areas.*
42		INC6	0,5	$c+d$	rI6 ← P + SIZE(P).
43		LDA	0,6(LINK)	$c+d$	
44		JAZ	1B	$c+d$	Jump if LINK(rI6) = Λ.
45		ST5	0,3(SIZE)	c	SIZE(P) ← rI5.
46		INC3	0,5	c	P ← P + SIZE(P).
47		JMP	G6A	c	
48	2H	DEC4	1	b	$k \leftarrow k - 1$.
49		INC2	1	b	Q ← Q + 1.
50		LD6	0,2(LINK)	b	
51		LDA	0,6(LINK)	b	
52		STA	0,2(LINK)	b	LINK(Q) ← LINK(LINK(Q)).
53	1H	J4NZ	2B	$a+b$	Jump if $k \neq 0$.
54	3H	INC3	0,5	$a+c$	P ← P + SIZE(P).
55	G8P	LDA	0,3(LINK)	$1+a+c$	
56		LD5	0,3(SIZE)	$1+a+c$	
57		JAZ	3B	$1+a+c$	Is LINK(P) = Λ?
58		LD4	0,3(T)	$1+a$	$k \leftarrow$ T(P).
59		ENT2	0,3	$1+a$	Q ← P.
60		J5NZ	1B	$1+a$	Jump unless SIZE(P) = 0.
61	G9	ENT3	1	1	*G9. Move.* P ← 1.
62		ENT1	1	1	Set rI1 for MOVE instructions.
63		JMP	G9P	1	
64	1H	STZ	0,3(LINK)	a	LINK(P) ← Λ.
65		ST5	*+1(4:4)	a	
66		MOVE	0,3(*)	a	NODE(rI1) ← NODE(P), rI1 ← rI1 + SIZE(P).
67	3H	INC3	0,5	$a+c$	P ← P + SIZE(P).
68	G9P	LDA	0,3(LINK)	$1+a+c$	
69		LD5	0,3(SIZE)	$1+a+c$	
70		JAZ	3B	$1+a+c$	Jump if LINK(P) = Λ.
71		J5NZ	1B	$1+a$	Jump unless SIZE(P) = 0. ▮

Note that in line 66 we are assuming that the size of each node is sufficiently small that it can be moved with a single MOVE instruction; this seems a fair assumption for most cases when this kind of garbage collection is applicable.

The total running time for this program is $(44a + 17b + 2w + 25c + 8d + 47)u$ where a is the number of accessible nodes, b is the number of link fields therein, c is the number of inaccessible nodes which are *not* preceded by an inaccessible node, d is the number of inaccessible nodes which *are* preceded by an inaccessible node, and w is the total number of words in the accessible nodes. If the memory contains n nodes, with ρn of these inaccessible, then we may estimate $a = (1 - \rho)n$, $c = (1 - \rho)\rho n$, $d = \rho^2 n$. Example: five-word nodes (on the average), with two link fields per node (on the average), and a memory of 1000 nodes. Then when $\rho = \frac{1}{5}$, it takes $374u$ per available node recovered; when $\rho = \frac{1}{2}$, it takes $104u$; and when $\rho = \frac{4}{5}$, it takes only $33u$.

37. First request $n + 2$ 1-cell blocks. If the middle cell a_{n+1} is unavailable, liberate all others and request a block of size $n + 1$. If a_{n+1} is available, find the maximum j and minimum k such that a_j and a_k are unavailable, $1 \leq j \leq n$, $n + 2 \leq k \leq 2n + 1$. Then $k - j \leq n$, so if we liberate all but a_j and a_k we cannot fill a request for n cells in a row.

Note: Considerably more is true; let $N(M, m)$ denote the minimum number of storage cells such that requests can always be filled when at most M cells are requested at a time and no blocks of size $> m$ are requested. Then either the first fit method or the buddy system can be used to show that $N(M, m) = O(M \log (m + 1))$; conversely, lower bounds can be established to show that $N(M, m)$ is asymptotically $cM \log m$ for large m and M. [These results are due to R. L. Graham and J. M. Robson (to be published).]

INDEX TO NOTATIONS

In the following formulas, letters which are not further qualified have the following significance:

j, k	integer-valued arithmetic expression
m, n	nonnegative integer-valued arithmetic expression
x, y, z	real-valued arithmetic expression
f	real-valued function
P	pointer-valued expression, i.e., either Λ or an address within a computer.

Formal symbolism	Meaning	Section reference
NODE(P)	the node (group of variables which are individually distinguished by their field names) whose address is P, P $\neq \Lambda$	2.1
F(P)	the variable in NODE(P) whose field name is F	2.1
CONTENTS(P)	contents of computer "word" whose address is P	2.1
LOC(V)	address of variable V within a computer	2.1
A_n	the nth element of linear array A	
A_{mn}	the element in row m, column n of rectangular array A	
$A[n]$	equivalent to A_n	1.1
$A[m, n]$	equivalent to A_{mn}	1.1
$V \leftarrow E$	give variable V the value of expression E	1.1
$U \leftrightarrow V$	interchange the values of variables U and V	1.1
P $\Leftarrow$ AVAIL	set the value of pointer variable P to the address of a new node, or signal memory overflow if there is no room for a new node	2.2.3

Formal symbolism	Meaning	Section reference
AVAIL $\Leftarrow$ P	NODE(P) is returned to free storage; all its fields lose their identity	2.2.3
top(S)	node at the top of a nonempty stack S	2.2.1
$X \Leftarrow S$	pop up S to X: set $X \leftarrow$ top(S); delete top(S) from nonempty stack S	2.2.1
$S \Leftarrow X$	push down X onto S: insert the value or group of values denoted by X as a new entry on the top of stack S	2.2.1
front(Q)	node at the front of a nonempty queue Q	2.2.1
$X \Leftarrow Q$	set $X \leftarrow$ front(Q); delete front(Q) from nonempty queue Q	2.2.1
$Q \Leftarrow X$	insert the value or group of values denoted by X as a new entry at the rear of queue Q	2.2.1
$(B \Rightarrow E_1; E_2)$	conditional expression: denotes E_1 if B is true, E_2 if B is false	8.1
δ_{jk}	Kronecker delta: $(j = k \Rightarrow 1;\ 0)$	1.2.6
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\prod_{R(k)} f(k)$	product of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\min_{R(k)} f(k)$	minimum value of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\max_{R(k)} f(k)$	maximum value of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$j \backslash k$	j divides k: $k \bmod j = 0$	1.2.4
gcd(j, k)	greatest common divisor of j and k: $$(j = k = 0 \Rightarrow 0; \quad \max_{d \backslash j, d \backslash k} d)$$	1.1
det (A)	determinant of square matrix A	1.2.3
A^T	transpose of rectangular array A: $$A^T[j, k] = A[k, j]$$	1.2.3
x^y	x to the y power, x positive	1.2.2
x^k	x to the kth power: $$\left(k \geq 0 \Rightarrow \prod_{0 \leq j < k} x, \quad 1/x^{-k}\right)$$	1.2.2

Formal symbolism	Meaning	Section reference
$x^{\overline{k}}$	x upper k: $\left(k \geq 0 \Rightarrow x(x+1)\cdots(x+k-1) = \prod_{0 \leq j < k} (x+j);\quad 1/(x+k)^{\overline{-k}}\right)$	1.2.6
$x^{\underline{k}}$	x lower k: $\left(k \geq 0 \Rightarrow x(x-1)\cdots(x-k+1) = \prod_{0 \leq j < k} (x-j);\quad 1/(x-k)^{\underline{-k}}\right) = (-1)^k(-x)^{\overline{k}}$	1.2.6
$n!$	n factorial: $1 \cdot 2 \cdot \cdots \cdot n = n^{\underline{n}}$	1.2.5
$\binom{x}{k}$	binomial coefficient: $(k < 0 \Rightarrow 0;\ x^{\underline{k}}/k!)$	1.2.6
$\binom{n}{n_1, n_2, \ldots, n_m}$	multinomial coefficient, $n = n_1 + n_2 + \cdots + n_m$	1.2.6
$\left[{n \atop m}\right]$	Stirling number of first kind: $\sum_{0<k_1<k_2<\cdots<k_{n-m}<n} k_1 k_2 \cdots k_{n-m}$	1.2.6
$\left\{{n \atop m}\right\}$	Stirling number of second kind: $\sum_{1 \leq k_1 \leq k_2 \leq \cdots \leq k_{n-m} \leq m} k_1 k_2 \cdots k_{n-m}$	1.2.6
$\{a \mid R(a)\}$	set of all a for which the relation $R(a)$ is true	
$\{a_1, \ldots, a_n\}$	the set $\{a_k \mid 1 \leq k \leq n\}$	
$\{x\}$	in contexts where a real value, not a set, is required, denotes fractional part: $x \bmod 1$	1.2.11.2
$\lvert x \rvert$	absolute value of x: $(x < 0 \Rightarrow -x;\ x)$	
$\lfloor x \rfloor$	floor of x, greatest integer function: $\max_{k \leq x} k$	1.2.4
$\lceil x \rceil$	ceiling of x, least integer function: $\min_{k \geq x} k$	1.2.4
$x \bmod y$	mod function: $(y = 0 \Rightarrow x;\ x - y\lfloor x/y \rfloor)$	1.2.4
$x \equiv y \pmod{z}$	relation of congruence: $x \bmod z = y \bmod z$	1.2.4

Formal symbolism	Meaning	Section reference
$\log_b x$	logarithm, base b, of x (real positive $b \neq 1$): $x = b^{\log_b x}$	1.2.2
$\ln x$	natural logarithm: $\log_e x$	1.2.2
$\exp x$	exponential of x: e^x	1.2.2
$\langle X_n \rangle$	the infinite sequence $X_0, X_1, X_2, \ldots$ (here n is a letter which is part of the symbol)	1.2.9
$f'(x)$	derivative of f at x	1.2.9
$f''(x)$	second derivative of f at x	1.2.10
$f^{(n)}(x)$	nth derivative: $(n = 0 \Rightarrow f(x);$ $g'(x)$ where $g(x) = f^{(n-1)}(x))$	1.2.11.2
$H_n^{(x)}$	$1 + 1/2^x + \cdots + 1/n^x = \sum_{1 \le k \le n} 1/k^x$	1.2.7
H_n	harmonic number: $H_n^{(1)}$	1.2.7
F_n	Fibonacci number: $(n \le 1 \Rightarrow n; F_{n-1} + F_{n-2})$	1.2.8
B_n	Bernoulli number	1.2.11.2
$B(x, y)$	Beta function	1.2.6
sign (x)	sign of x: $(x = 0 \Rightarrow 0; (x > 0 \Rightarrow +1; -1))$	
$\zeta(x)$	zeta function: $H_\infty^{(x)}$ when $x > 1$	1.2.7
$\Gamma(x)$	gamma function: $\gamma(x, \infty)$; $(x - 1)!$ when x is a positive integer	1.2.5
$\gamma(x, y)$	incomplete gamma function	1.2.11.3
γ	Euler's constant	1.2.7
e	base of natural logarithms: $\sum_{k \ge 0} 1/k!$	1.2.2
∞	infinity: larger than any number	
Λ	null link (pointer to no address)	2.1
$\emptyset$	empty set (set with no elements)	
ϕ	golden ratio, $\frac{1}{2}(1 + \sqrt{5})$	1.2.8
$\varphi(n)$	Euler's totient function: $\sum_{\substack{0 \le k < n \\ \gcd(k,n)=1}} 1$	1.2.4
$p(n)$	number of partitions of n	1.2.1
$O(f(n))$	big-oh of $f(n)$ as $n \to \infty$	1.2.11.1

Formal symbolism	Meaning	Section reference
$O(f(x))$	big-oh of $f(x)$, for small x (or for x in some specified range)	1.2.11.1
(min x_1, ave x_2, max x_3, dev x_4)	a random variable having minimum value x_1, average ("expected") value x_2, maximum value x_3, standard deviation x_4	1.2.10
mean (g)	mean value of probability distribution represented by generating function g: $g'(1)$	1.2.10
var (g)	variance of probability distribution represented by generating function g: $$g''(1) + g'(1) - g'(1)^2$$	1.2.10
`P*`	address of preorder successor of `NODE(P)` in a tree or binary tree	2.3.1, 2.3.2
`P$`	address of postorder successor of `NODE(P)` in a tree or binary tree	2.3.1, 2.3.2
`P#`	address of endorder successor of `NODE(P)` in a tree or binary tree	2.3.1, 2.3.2
`*P`	address of preorder predecessor of `NODE(P)` in a tree or binary tree	2.3.1, 2.3.2
`$P`	address of postorder predecessor of `NODE(P)` in a tree or binary tree	2.3.1, 2.3.2
`#P`	address of endorder predecessor of `NODE(P)` in a tree or binary tree	2.3.1, 2.3.2
∎	end of algorithm, program, or proof	1.1
␣	one blank space	1.3.1
rA	register A (accumulator) of `MIX`	1.3.1
rX	register X (extension) of `MIX`	1.3.1
rI1, . . . , rI6	(index) registers I1, . . . , I6 of `MIX`	1.3.1
rJ	(jump) register J of `MIX`	1.3.1
`(L:R)`	partial field of `MIX` word, $0 \le$ `L` $\le$ `R` ≤ 5	1.3.1
`OP ADDRESS,I(F)`	notation for `MIX` instruction	1.3.1, 1.3.2
u	unit of time in `MIX`	1.3.1
*	"self" in `MIXAL`	1.3.2
`0F, 1F, 2F, . . . , 9F`	"forward" local symbol in `MIXAL`	1.3.2
`0B, 1B, 2B, . . . , 9B`	"backward" local symbol in `MIXAL`	1.3.2
`0H, 1H, 2H, . . . , 9H`	"here" local symbol in `MIXAL`	1.3.2

TABLES OF NUMERICAL QUANTITIES

Table 1

Quantities which are frequently used in standard subroutines and in analysis of computer programs. (40 decimal places)

$\sqrt{2}$ =	1.41421 35623 73095 04880 16887 24209 69807 85697−
$\sqrt{3}$ =	1.73205 08075 68877 29352 74463 41505 87236 69428+
$\sqrt{5}$ =	2.23606 79774 99789 69640 91736 68731 27623 54406+
$\sqrt{10}$ =	3.16227 76601 68379 33199 88935 44432 71853 37196−
$\sqrt[3]{2}$ =	1.25992 10498 94873 16476 72106 07278 22835 05703−
$\sqrt[3]{3}$ =	1.44224 95703 07408 38232 16383 10780 10958 83919−
$\sqrt[4]{2}$ =	1.18920 71150 02721 06671 74999 70560 47591 52930−
$\ln 2$ =	0.69314 71805 59945 30941 72321 21458 17656 80755+
$\ln 3$ =	1.09861 22886 68109 69139 52452 36922 52570 46475−
$\ln 10$ =	2.30258 50929 94045 68401 79914 54684 36420 76011+
$1/\ln 2$ =	1.44269 50408 88963 40735 99246 81001 89213 74266+
$1/\ln 10$ =	0.43429 44819 03251 82765 11289 18916 60508 22944−
π =	3.14159 26535 89793 23846 26433 83279 50288 41972−
$1° = \pi/180$ =	0.01745 32925 19943 29576 92369 07684 88612 71344+
$1/\pi$ =	0.31830 98861 83790 67153 77675 26745 02872 40689+
π^2 =	9.86960 44010 89358 61883 44909 99876 15113 53137−
$\sqrt{\pi} = \Gamma(1/2)$ =	1.77245 38509 05516 02729 81674 83341 14518 27975+
$\Gamma(1/3)$ =	2.67893 85347 07747 63365 56929 40974 67764 41287−
$\Gamma(2/3)$ =	1.35411 79394 26400 41694 52880 28154 51378 55193+
e =	2.71828 18284 59045 23536 02874 71352 66249 77572+
$1/e$ =	0.36787 94411 71442 32159 55237 70161 46086 74458+
e^2 =	7.38905 60989 30650 22723 04274 60575 00781 31803+
γ =	0.57721 56649 01532 86060 65120 90082 40243 10422−
$\ln \pi$ =	1.14472 98858 49400 17414 34273 51353 05871 16473−
ϕ =	1.61803 39887 49894 84820 45868 34365 63811 77203+
e^γ =	1.78107 24179 90197 98523 65041 03107 17954 91696+
$e^{\pi/4}$ =	2.19328 00507 38015 45655 97696 59278 73822 34616+
$\sin 1$ =	0.84147 09848 07896 50665 25023 21630 29899 96226−
$\cos 1$ =	0.54030 23058 68139 71740 09366 07442 97660 37323+
$\zeta(3)$ =	1.20205 69031 59594 28539 97381 61511 44999 07650−
$\ln \phi$ =	0.48121 18250 59603 44749 77589 13424 36842 31352−
$1/\ln \phi$ =	2.07808 69212 35027 53760 13226 06117 79576 77422−

Table 2

Quantities which are frequently used in standard subroutines and in analysis of computer programs, in *octal* notation. The *name* of each quantity, appearing at the left of the equal sign, is given in decimal notation.

$0.1 =$	*0.06314 63146 31463 14631 46314 63146 31463 14631 4632*
$0.01 =$	*0.00507 53412 17270 24365 60507 53412 17270 24365 6051*
$0.001 =$	*0.00040 61115 64570 65176 76355 44264 16254 02030 4467*
$0.0001 =$	*0.00003 21556 13530 70414 54512 75170 33021 15002 3522*
$0.00001 =$	*0.00000 24761 32610 70664 36041 06077 17401 56063 3442*
$0.000001 =$	*0.00000 02061 57364 05536 66151 55323 07746 44470 2603*
$0.0000001 =$	*0.00000 00153 27745 15274 53644 12741 72312 20354 0215*
$0.00000001 =$	*0.00000 00012 57143 56106 04303 47374 77341 01512 6333*
$0.000000001 =$	*0.00000 00001 04560 27640 46655 12262 71426 40124 2174*
$0.0000000001 =$	*0.00000 00000 06676 33766 35367 55653 37265 34642 0163*
$\sqrt{2} =$	*1.32404 74631 77167 46220 42627 66115 46725 12575 1744*
$\sqrt{3} =$	*1.56663 65641 30231 25163 54453 50265 60361 34073 4222*
$\sqrt{5} =$	*2.17067 36334 57722 47602 57471 63003 00563 55620 3202*
$\sqrt{10} =$	*3.12305 40726 64555 22444 02242 57101 41466 33775 2253*
$\sqrt[3]{2} =$	*1.20505 05746 15345 05342 10756 65334 25574 22415 0303*
$\sqrt[3]{3} =$	*1.34233 50444 22175 73134 67363 76133 05334 31147 6012*
$\sqrt[4]{2} =$	*1.14067 74050 61556 12455 72152 64430 60271 02755 7314*
$\ln 2 =$	*0.54271 02775 75071 73632 57117 07316 30007 71366 5364*
$\ln 3 =$	*1.06237 24752 55006 05227 32440 63065 25012 35574 5534*
$\ln 10 =$	*2.23273 06735 52524 25405 56512 66542 56026 46050 5071*
$1/\ln 2 =$	*1.34252 16624 53405 77027 35750 37766 40644 35175 0435*
$1/\ln 10 =$	*0.33626 75425 11562 41614 52325 33525 27655 14756 0622*
$\pi =$	*3.11037 55242 10264 30215 14230 63050 56006 70163 21122* 0111G
$1^\circ = \pi/180 =$	*0.01073 72152 11224 72344 25603 54276 63351 22056 1154*
$1/\pi =$	*0.24276 30155 62344 20251 23760 47257 50765 15156 7007*
$\pi^2 =$	*11.67517 14467 62135 71322 25561 15466 30021 40654 3410*
$\sqrt{\pi} = \Gamma(1/2) =$	*1.61337 61106 64736 65247 47035 40510 15273 34470 1776*
$\Gamma(1/3) =$	*2.53347 35234 51013 61316 73106 47644 54653 00106 6605*
$\Gamma(2/3) =$	*1.26523 57112 14154 74312 54572 37655 60126 23231 0245*
$e =$	*2.55760 52130 50535 51246 52773 42542 00471 72363 6166*
$1/e =$	*0.27426 53066 13167 46761 52726 75436 02440 52371 0336*
$e^2 =$	*7.30714 45615 23355 33460 63507 35040 32664 25356 5022*
$\gamma =$	*0.44742 14770 67666 06172 23215 74376 01002 51313 2552*
$\ln \pi =$	*1.11206 40443 47503 36413 65374 52661 52410 37511 4606*
$\phi =$	*1.47433 57156 27751 23701 27634 71401 40271 66710 1501*
$e^\gamma =$	*1.61772 13452 61152 65761 22477 36553 53327 17554 2126*
$e^{\pi/4} =$	*2.14275 31512 16162 52370 35530 11342 53525 44307 0217*
$\sin 1 =$	*0.65665 24436 04414 73402 03067 23644 11612 07474 1451*
$\cos 1 =$	*0.42450 50037 32406 42711 07022 14666 27320 70675 1232*
$\zeta(3) =$	*1.14735 00023 60014 20470 15613 42561 31715 10177 0662*
$\ln \phi =$	*0.36630 26256 61213 01145 13700 41004 52264 30700 4065*
$1/\ln \phi =$	*2.04776 60111 17144 41512 11436 16575 00355 43630 4065*

Tables 1 and 2 contain several hitherto unpublished 40-digit values which have been furnished to the author by Dr. John W. Wrench, Jr.

For high-precision values of constants not found in this list, see J. Peters, *Ten Place Logarithms of the Numbers from 1 to 100000*, Appendix to Volume 1 (New York: F. Ungar Publ. Co., 1957); and *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U. S. Govt. Printing Office, 1964), Chapter 1.

Table 3

Values of harmonic numbers, Bernoulli numbers, and Fibonacci numbers for small values of n.

n	H_n	B_n	F_n	n
0	0	1	0	0
1	1	−1/2	1	1
2	3/2	1/6	1	2
3	11/6	0	2	3
4	25/12	−1/30	3	4
5	137/60	0	5	5
6	49/20	1/42	8	6
7	363/140	0	13	7
8	761/280	−1/30	21	8
9	7129/2520	0	34	9
10	7381/2520	5/66	55	10
11	83711/27720	0	89	11
12	86021/27720	−691/2730	144	12
13	1145993/360360	0	233	13
14	1171733/360360	7/6	377	14
15	1195757/360360	0	610	15
16	2436559/720720	−3617/510	987	16
17	42142223/12252240	0	1597	17
18	14274301/4084080	43867/798	2584	18
19	275295799/77597520	0	4181	19
20	55835135/15519504	−174611/330	6765	20
21	18858053/5173168	0	10946	21
22	19093197/5173168	854513/138	17711	22
23	444316699/118982864	0	28657	23
24	1347822955/356948592	−236364091/2730	46368	24
25	34052522467/8923714800	0	75025	25

For any x, let $H_x = \sum_{n\geq 1}\left(\frac{1}{n} - \frac{1}{n+x}\right)$. Then

$$H_{1/2} = 2 - 2\ln 2,$$
$$H_{1/3} = 3 - \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,$$
$$H_{2/3} = \tfrac{3}{2} + \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,$$
$$H_{1/4} = 4 - \tfrac{1}{2}\pi - 3\ln 2,$$
$$H_{3/4} = \tfrac{4}{3} + \tfrac{1}{2}\pi - 3\ln 2,$$
$$H_{1/5} = 5 - \tfrac{1}{2}\pi\phi\sqrt{\frac{2+\phi}{5}} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{2/5} = \tfrac{5}{2} - \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{3/5} = \tfrac{5}{3} + \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{4/5} = \tfrac{5}{4} + \tfrac{1}{2}\pi\phi\sqrt{\frac{2+\phi}{5}} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{1/6} = 6 - \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,$$
$$H_{5/6} = \tfrac{6}{5} + \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,$$

and, in general, when $0 < p < q$ (cf. exercise 1.2.9–19),

$$H_{p/q} = \frac{q}{p} - \tfrac{1}{2}\pi\cot\frac{p}{q}\pi - \ln 2q + 2\sum_{1\leq n<q/2}\cos\frac{2\pi np}{q}\ln\sin\frac{n}{q}\pi.$$

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information; an answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

A-register of MIX, 122.
A-1 compiler, 458.
Aardenne-Ehrenfest, Taniana van, 375, 578.
Abel, Niels Henrik, 56, 86.
  binomial theorem generalized, 56, 70, 72, 398.
  limit theorem, 94.
Abramowitz, Milton, 66, 92, 615.
ACE computer, Pilot, 226.
Adams, Charles William, 226.
ADD, 127, 128, 204.
Add to list: *see* Insertion.
Addition of polynomials, 273–276, 355–359 361.
Address: A number used to identify a position in memory.
  field of MIXAL line, 123, 141, 147, 151, 152.
  of node, 229–230.
  portion of MIX instruction, 123.
Address transfer operators of MIX, 129, 206–207.
Adjacent vertices of a graph, 362.
Agenda, 285, 293, 552.
Aiken, Howard Hathaway, 225.
al-Khowârizmî, Abu Ja'far Mohammed ibn Mûsâ, 1, 78.
Alanen, Jack David. xii.
ALF (alphabetic data), 148, 149, 151.
Algebraic formulas, manipulation of, 335–347, 461.
  differentiation, 337–346, 359, 458.
  representation as trees, 312, 335–336, 458.
  simplification of, 339, 346.
Algorithm, origin of word, 1–2.
Algorithms, 1–9.
  analysis of, vii, 7, 94–104, 166–169, 175, 246–247, 249–250, 265, 276, 323–324, 380–381, 445–446.
  communication of, 16.
  effective, 6, 8, 9.
  equivalence between, 466.
  form of in this book, 2–4.
  hardware-oriented, 26, 249, 600.
  how to read, 4, 16.
  proof of, 14–20, 318–319, 420.
  properties of, 4–6, 9.
  random paths in, 380–381.
  set theoretical definition, 8–9.
  theory of, 7, 9.
Allocation of tables, *see* Dynamic storage allocation, Linked allocation, Representation, Sequential allocation.
Along order, 459.
Alphameric character: A letter, digit, or special character symbol.
  codes for MIX, 132, 134, 136–137.
AMM: American Mathematical Monthly, the official journal of the Mathematical Association of America, Inc.
Analysis of algorithms, vii, 7, 94–104, 166–169, 175, 246–247, 249–250, 265, 276, 323–324, 380–381, 445–446.
Analytical Engine, 1, 225.
Ancestor, in a tree structure, 309.
André, Antoine Désiré, 531.
Anticipated input, 212, *see* Buffering.
Antisymmetric relation, 258.
Apostol, Tom Mike, 28.
Arborescence, 362, *see* Oriented trees.
Arc in a directed graph, 371.
Area of memory, 435.
Arguments of subroutines, 183, 185.
Arithmetic: Addition, subtraction, multiplication, and division.
  fixed-point, 154–157.
  floating-point, 127, 304.
  operators of MIX, 127–128, 135, 204.
  polynomial, 272–277, 355–359, 361.
  scaled decimal, 156–157.
Arithmetic expressions, *see* Algebraic formulas.

Arithmetic progression, sum of, 11, 13, 31, 55.
Array: A table which usually has a k-dimensional rectangular structure, 3, 228, 295–304.
one-dimensional, *see* Linear list.
represented as tree, 310, 312.
sequential allocation, 154, 296–298, 302–303.
tetrahedral, 298, 303, *see* Binomial number system.
two-dimensional, *see* Matrix.
Arrows, used to represent links in diagrams, 230.
Assembly language: A language which is intended to facilitate the construction of programs in machine language by making use of symbolic and mnemonic conventions to denote machine language instructions.
for MIX, 141–153.
Assembly program, 149.
ASSIGN a buffer, 215, 218, 224.
Assignment operation, 3.
Asterisk ("*"), in assembly language, 143, 145, 147, 149, 152.
Asymmetric relations, 258.
Asymptotic values: Functions which express the limiting behavior approached by numerical quantities.
derivation of, 104–119, 239, 395–396.
Atom (in a List), 312–313, 406–409, 417.
Automata theory, 226, 462.
Automaton: An abstract machine which is formally defined in some manner, often intended to be a model of some aspects of actual computers (plural: Automata), 462–463.
AVAIL stack: Available space list, 253.
Available space list, 253–254, 263, 266, 275, 289, 290, 411–413, 419–420, 435–455.
history, 457.
variable-size blocks, 436–455.
Average value of a probability distribution, 96, 98–99, 101.

Babbage, Charles, 1, 225.
Bachmann, Paul Gustav Heinrich, 104.
Backus, John W., 226.
Bailey, Michael John, 461.
Balanced directed graph, 374–377.
Ball, Walter William Rouse, 158–159
Ballot problem, 531–533.
Barnett, Michael Peter, 461.
Base address, 230, 240.
Bead, 229, *see* Node.
Before and after diagrams, 256–257.
Bellman, Richard Ernest, xvii.
Bennett, John Makepeace, 226.
Berge, Claude, 406.
Berger, Robert, 385.
Bernoulli, James (= Jakob = Jacques) 109.
numbers, 74, 90–91, 108–112.
numbers, table, 615.
polynomials, 42, 109–112.
Bernoulli, Nikolaus (= Nicolas), 82, 86.
Bertrand, Joseph Louis François, postulate, 506.
Berztiss, A. T., 461.
Best-fit method of storage allocation, 436–437, 448, 452–453.
Beta function, 71.
Bháscara Áchárya, 52.
"Big-oh" notation, 104–108.
Bigelow, Richard H., 558.
Binary computer: A computer which manipulates numbers primarily in the binary (radix 2) number system.
Binary logarithm, 22, 25.
Binary trees, 308–309, 314–334, 345, 362, 399–405, 458–459.
complete, 400–401.
copying of, 327–328, 332, 346.
correspondence to trees and forests, 333–334, 345.
definition of, 309.
"Dewey" notation for, 315, 329, 345, 405.
enumeration of, 388–389.
equivalent, 326, 331.
erasing of, 331.
extended, 399–405.
oriented, 396.
path length of, 399–405.
right-threaded, 325, 331, 332, 336–346, 459.
representation of, 315–316, 319–322, 325, 332, 401.
similar, 325–326, 331.
threaded, 319–325, 329–332, 334, 420, 459.
traversal of, 316–332.
Binet, Jacques Phillipe Marie, 405, 578.
Binomial coefficients, 51–73, 88.
combinatorial interpretation, 51, 72.
defined, 51.
generalized, 64, 69, 71, 72, 85.
generating functions, 88–90.
history, 52.
sums involving, 53–73, 75–77, 84, 88–90, 93.
table of, 52.
Binomial distribution, 103.
Binomial number system, 72.
Binomial theorem, 55–56, 89–90.
Abel's generalization, 56, 70, 72, 398.
generalizations of, 56, 64, 72, 90, 398.
Hurwitz's generalization, 398, 488.

Bit: "Binary digit," either zero or unity.
BIT: Nordisk Tidskrift for Informationsbehandling, a journal published by Regnecentralen, Copenhagen, Denmark.
Blaauw, Gerrit Anne, 457.
Block of memory, 435.
Blocking of records, 214, 222.
Bobrow, Daniel Gureasko, 459.
Bolzano, Bernhard, theorem, 381.
Boncompagni, Prince Baldassarre, 79.
Boothroyd, John, 174, 518.
Bottom of stack, 237.
Bottom-up process, 351, 362.
Boundary tag method of storage allocation, 441–442, 449–450, 453, 460.
Bourne, Charles Percy, 511.
Branch instruction: A conditional "jump" instruction.
Branch node of tree, 305.
Brother, in a tree structure, 307.
BROTHER link in tree, 426–432, see RLINK.
Brouwer, Luitzen Egbertus Jan, 405.
Buddy system for storage allocation, 442–445, 448–450, 453–454, 460, 605.
Buffering of input-output, 154, 155, 212–225,
  history, 227.
  swapping, 143–144, 155, 213–215, 222.
Buniakovskiĭ, Viktor Yakovlevich, 631.
Burke's *Peerage*, 308.
Burks, Arthur Walter, 359.
Burleson, Peter Barrus, 461.
Burroughs B220, xii, 120.
Burroughs B5000–B5500, xii, 460.
Byte: Basic unit of data, usually associated with alphameric characters.
  in MIX, 120–121, 135.
Byte size in MIX: The number of distinct values that might be stored in a byte.

CACM: Communications of the ACM, a publication of the Association for Computing Machinery.
Cajori, Florian, 23.
Calendar, 156.
California Institute of Technology (Caltech), xii, 280.
Call: To activate another routine in a program.
Calling sequence, 183–186, 189, 192–193.
Canonical cycle notation for permutations, 176.
Canonical representation of oriented trees, 390–391, 397–398.
Car: LISP terminology for the first component of a List; analogous to INFO and DLINK on p. 410, or to ALINK on p. 417.
Card format for MIXAL programs, 148–149.
Cards, playing, 49, 68, 229–233, 377.
Carlitz, Leonard, 501.
Carr, John W., III, 457.
Catalan, Eugène Charles, 405.
Cauchy, Augustin Louis, 36–37, 578.
Cayley, Arthur, 396, 405–406.
CDC G20, 120.
CDC 1604, 120, 523.
Cdr: LISP terminology for the remainder of a List with its first component deleted; analogous to RLINK on p. 410 or to BLINK on p. 417.
Ceiling function, 37, 40–44.
Cell: A word of the computer memory, 123
Cellar, 236.
Centroid of a free tree, 387–388, 396.
Chain: A word used by some authors to denote a linked linear list.
Chain rule for differentiation, 50, *see* Faà di Bruno's formula.
Chaining: A word used by some authors in place of "linking."
Channel: A data-transmission device connected to a computer, 221.
CHAR (convert to characters), 134.
Character code of MIX, 132, 134, 136–137.
Characteristic function of a probability distribution, 101.
Chebyshev, Pafnuĭ L'vovich, polynomials, 493.
Checkerboard, 435.
Checkerboarding, *see* Fragmentation.
Chess, 6, 270.
Chung, Kai Lai, 103.
Church, Robert, 117.
CI: The comparison indicator of MIX, 136–137, 224.
Circle of buffers, 214–225.
Circuit, Eulerian, in a directed graph, 373–375, 378–379.
Circuit, Hamiltonian, in a directed graph, 334, 378.
Circular definition, 260, *see* Definition, circular.
Circular linkage, 270–277, 300, 355, 409–410, 416, 458.
Circular list, 270–277, 409–410, 458.
Circular store, 236.
Circulating shift, 131.
CITRUS, 456.
Clavius, Christopher, S. J., 155–156.
Clock, real time, 224.
Clock, simulated, 281, 285, 451.
Clock, solitaire game, 377.
Closed subroutine, *see* Subroutine.
CMPA (compare A), 130, 206–207.
CMPX (compare X), 130, 206–207.
CMP1 (compare 1), 130, 206–207.

COBOL: "Common Business-Oriented Language," 423–434, 456, 552, 572.
Coding: Synonym for "programming," but with even less prestige associated.
Cofactor of element in square matrix: Determinant of the matrix obtained by replacing this element by unity and replacing all other elements having the same row or column by zero, 35.
Cohen, Jacques, 460.
Coin tossing, 100–101.
Collins, George Edwin, 460.
Combinations of n objects taken k at a time, 51, 68.
  with repetitions permitted, 72–73, 93, 386, 388.
  with restricted repetitions, 93.
Combinatorial matrix, 36, 584.
Comfort, Webb T., xii, 460.
COMIT, 460.
Command: Synonym for "instruction."
Comment in assembly language, 145, 149.
Comp. J.: The Computer Journal, published by The British Computer Society.
Compacting memory, 421, 439–440, 450, 451, 454–455.
Comparison indicator of MIX, 122, 129–130, 138, 202, 224.
Comparison operators of MIX, 130, 206–207.
Compiler: Program which translates programming languages.
  algorithms especially for use in, 360–361, 423–424, 552.
Complete binary tree, 400–401.
Compound interest, 23.
Computational method, 5, 8.
Compute: To process data.
Computer: A data processor, 1.
Computer language, *see* Assembly language, Machine language, Programming language.
CON (constant), 146, 151–152.
Concatenation of strings, 271–272.
Conditional expression, 459, 608.
Congruence, 38–39.
Connected directed graph, 372, 376, 377.
  strongly, 372, 377.
Connected graph, 362.
Conservative law, 167, *see* Kirchhoff's law.
Constants in assembly language, 146, 151–152.
Construction of trees, 339, 342, 426–427.
CONTENTS, 123, 231–233.
Continuous simulation, 279.
Convergence: An infinite sequence $\langle X_n \rangle$ converges if it approaches a limit as n approaches infinity; an infinite sum or product is said to "converge" or to "exist" if it has a value according to the conventions of mathematical calculus; see Eq. 1.2.3-3 and exercise 1.2.3-21.
  of power series, 86, 395.
Conversion operators of MIX, 134.
Convolution of probability distributions: The distribution obtained by adding two independent variables, 99, 101.
Conway, Melvin Edward, xii, 147, 226.
Copy a data structure: To duplicate a structured object by producing another distinct object having the same data values and structural relationships.
  binary tree, 327–328, 332, 346.
  linear list, 277.
  List, 421.
  tree, 327–328, 332, 346.
  two-dimensional linked list, 304.
Coroutine, 190–196, 218–220, 281–293, 318.
  history, 226.
  linkage, 190, 196, 220, 288–289.
Correspondence between binary trees and forests, 333–334, 345.
Cousins, 314.
Coxeter, Harold Scott MacDonald, 79, 158.
Critical path time, 213.
Crossword puzzle, 159–160.
Cumulants of probability distribution, 101–103.
Cycle: Path from vertex to itself.
  detection of, 268, 369.
  fundamental, 366–368, 376.
  in directed graph, 371–372.
  in graph, 362.
  in permutation, 160–164, 173, 176–181.
  in random permutation, 176–181.
  notation for permutations, 160–164, 169–170, 176, 179–181.
  oriented, in directed graph, 371.
  singleton, 160–161, 164, 168, 177–179.

Dahm, David Michael, 432, 434.
Data (originally plural of the word "datum," but now used as singular or plural): Representation in a precise, formalized language of some facts or concepts, often numeric or alphabetic values, in a manner which can be manipulated by a computational method, 211.
  packed, 124, 153.
Data structure: A table of data including structural relationships, 228–463.
  linear list structures, 234–295.
  List structures, 406–422.
  multilinked structures, 423–434.
  orthogonal lists, 295–304, 423–434.
  tree structures, 305–406.
Daughter, 307, *see* Son.

de Bruijn, Nicolaas Govert, xii, 118, 119, 375, 379, 538, 578.
de La Loubère, Simon, 158.
de Moivre, Abraham, 82, 103.
De Morgan, Augustus, 17.
Debugging: Detecting and removing bugs (errors), 189, 197, 294.
DECA (decrease A), 129, 206.
Decimal computer: A computer which manipulates numbers primarily in the decimal (radix ten) number system.
DECX (decrease X), 129, 206.
DEC1 (decrease 1), 129, 206.
Defined symbol, in assembly language, 149.
Definition, circular, *see* Circular definition.
Degree, of node in tree, 305, 314, 345, 350–351, 376.
  of vertex in directed graph, 371.
Deletion of node: Removing it from a data structure and possibly returning it to available storage.
  from available space list, *see* Reservation.
  from deque, 248, 266, 271, 294.
  from doubly linked list, 278–279, 288, 294, 444–445.
  from linear list, 235.
  from linked list, 232, 252, 274, 278–279, 288, 294, 301–302, 357–358, 440, 442, 444–445.
  from queue, 237–238, 240–241, 257–258, 262, 271.
  from stack, 237–238, 240–241, 243, 255–256, 265–266, 271, 276, 278–279, 323, 415–416.
  from tree, 357–358.
  from two-dimensional list, 301–302.
Deque: Double-ended queue, 235–239, 458.
  deletion from, 248, 266, 271, 294.
  input-restricted, 235–239, 415.
  insertion into, 248, 266, 271, 294.
  linked allocation, 251, 270, 278.
  output-restricted, 235–239, 271.
  sequential allocation, 240.
Derivative of a formula, 89, 337.
Descendant, in a tree structure, 309.
Determinant of a square matrix, 35–37, 377–378, 474.
Deuel, Phillip D., 552.
Deutsch, L. Peter, 417, 421.
Dewey, Melvil, notation for binary trees (due to Galton), 315, 329, 345, 405.
  for trees, 310–311, 314–315, 345, 381–382, 459.
Diagrams of structural information, 230–231.
  before-and-after, 256–257.
  List structures, 312–313, 407.
  tree structures, 306–307, 309.
Dickson, Leonard Eugene, 80.
Difference of function, 64.
  divided, 472.
Differentiation, algebraic, 89, 337–346, 359, 458.
  chain rule for, 50, *see* Faà di Bruno's formula.
Digamma function, 94, 490, 616.
Digit: One of the symbols used in radix notation; usually a decimal digit, one of the symbols 0, 1, . . . , or 9.
Digraph, 371, *see* Directed graph.
Dijkstra, Edsger Wybe, 227, 236, 458.
Dilworth, Robert Palmer, xii.
d'Imperio, Mary E., 461.
Directed graph, 371–381, 420.
  as flow chart, 365, 380–381.
  balanced, 374, 377.
  connected 372, 376, 377.
  regular, 378.
  representation of, 380.
  rooted, 372.
  strongly connected, 372, 377.
Discrete system simulation, 199, 279–295.
  synchronous, 280, 295.
Disjoint sets: Sets with no common elements.
Disk files, 132–133, 436, 461–462.
Distribution: A specification of probabilities which govern the value of a random variable.
  binomial, 100, 103.
  normal, 102–103.
  Poisson, 103, 519.
  uniform, 100–101.
Distributive law, 27, 35, 40.
DIV (divide), 127–128, 135, 204.
Divided differences, 472.
Division converted to multiplication, 513.
Divisor: x is a divisor of y if $y \bmod x = 0$; it is a *proper* divisor if in addition $1 < x < y$.
Dixon, Alfred Cardew, 489.
Dixon, Robert Dan, 504.
DLINK: Link downward, 408, 410.
Doig, Alison, 406.
Domino problem, 382–385.
Doubly linked list, 278–279, 285–288, 294–295, 409–411, 441–442, 443–445, 453, 458.
Dougall, John, 489.
Drum, 132–133, 456, 461–462.
Dual order for traversing tree, 330, 331.
Dull, Brutus Cyclops, 107.
Dummy variable, 27.
Dunham, Bradford, 346.
Dunlap, James Robert, xii, 456.
Dvoretzky, Aryah, 531.

Dynamic storage allocation, 242–251, 253–254, 411–413, 419–420, 435–455.
history 456–457, 460–461.
running time estimates, 419–420, 445–450.
Dynastic order, 335, *see* Preorder.

Easter date, 155–156.
Edge in a graph, 362.
Edwards, Daniel James, 421.
Effective algorithm, 6, 8, 9.
Eisenstein, Ferdinand Gotthold, 479.
Elementary symmetric functions, 93, 94, 494.
Elevator system, 280–295.
Embedding of partial order into linear order, 259, *see* Topological sorting.
Embedding of tree in another tree, 347, 385.
END, 148, 151.
End of file, 212–213, 224 (exercise 12).
Endorder for binary tree, 316–317, 319, 324, 328–330.
Endorder for tree, 334–335, 345, 349.
English letter frequencies, 155.
ENNA (enter negative A), 129, 206.
ENNX (enter negative X), 129, 206.
ENN1 (enter negative 1), 129, 206.
ENTA (enter A), 129, 206.
Entity, 229, *see* Node.
Entrances to subroutines, 183–187.
ENTX (enter X), 129, 206.
ENT1 (enter 1), 129, 206.
Enumeration of tree structures, 377–378, 385–399, 404.
history, 405–406.
EQU (equivalent to), 142, 145–146, 151, 152.
Equivalence algorithm (Algorithm 2.3.3E), 354–355, 360–361, 376, 572, 575.
Equivalence between two algorithms, 466.
Equivalence classes, 353.
Equivalence declaration, 355, 360–361.
Equivalence problem, 353, 360–361.
Equivalence relation, 353.
Equivalent of a MIXAL symbol, 152.
Equivalent trees, 326, 331, 345 (exercise 10).
Erase a data structure: To return all its nodes to available storage.
binary tree, 331.
linear list, 270, 271, 277.
List, 412–413.
Erdélyi, Arthur, 398, 531.
Erdwinn, Joel Dyne, 226.
Errors, avoiding, 256–257.
computational, 24, 26, 302.
detection of, 189, 197, 294.
Etherington, Ivor Malcolm Haddon, 398, 531.
Ettingshausen, Andreas von, 52.
Euclides (= Euclid), 2, 4, 5.
algorithm for gcd, 2, 4–9, 13–17, 19, 40, 79, 80–81.
Euler, Leonhard, 48, 51, 56, 86, 108, 110, 373, 405, 494, 531.
constant, 74, 110.
summation formula, 108–112, 116, 119.
theorem of, 41.
totient function $\varphi(n)$, 41, 181 (exercise 27).
Eulerian circuit, 373–375, 377–379.
Evaluate tree function, 351, 362.
Evans, Arthur, Jr., 198.
Exchange operation, 3, 179.
Exclusive or, 443, 454, 550.
Execution time, methods for determining, 95–104, 166–169.
for MIX instructions, 134–137.
Exercises, notes on, xvii–xix.
Exit: Place where control leaves a routine.
Exits from subroutines, multiple, 186, 266.
Expected value of a probability distribution: The average or "mean" value, 98.
Exponents, laws of, 21–22, 25.
Expressions, arithmetic, *see* Algebraic formulas.
Extended binary tree, 399–405.
External path length, 399–405.

Faà di Bruno, Francesco, formula of, 50, 92, 103.
Factorial, 45–51, 53, 111.
Factorial powers, 70, 609.
Factorization into primes, 18, 41, 46, 49, 68.
FADD (floating add), 127, 304.
Fail-safe program, 267–268.
Family-order sequential representation of trees, 349–350, 359.
Family tree, 307–309, 314.
Farber, David Jack, 460.
Farey, John, series, 157.
Father, in a tree structure, 307, 314, 333–334.
FATHER link in tree, 352–355, 359–360, 426–432.
FCMP (floating compare), 127, 502, 556.
FDIV (floating divide), 127, 304.
Ferguson, David Elton, xii, 227, 332.
Fermat, Pierre de, 17.
theorem, 39.
Feynman, Richard Phillips, 26.
Fibonacci, Leonardo, 78–79.
number system, 85.
numbers: elements of the Fibonacci sequence, 13, 18, 78–85, 454.
numbers, generating function, 81–82.
numbers, table of, 615.
Quarterly, 80.
sequence, 13, 18, 78–85, 454.
string sequence, 85.

Field: A designated portion of a set of data, usually consisting of contiguous (adjacent) symbols; e.g., a field of a punched card is usually a set of adjacent column positions.
partial, of MIX word, 122–127, 135, 139, 203, 232–233.
within a node, 229.
within a node, notations for, 231–233, 457–458.
FIFO, 236, 458, *see* Queue.
Fifty percent rule, 445–446.
Final vertex of arc, 371.
Fine, Nathan Jacob, 483.
First-fit method of storage allocation, 436–438, 447–450, 452–453, 605.
First-in-first-out, 236, *see* Queue.
Fischer, Michael John, 353.
Fixed element of permutation, 177–179.
Fixed-point arithmetic, 154–157.
Flag, *see* Sentinel.
Floating-point arithmetic, 127, 304.
Floating-point operators of MIX, 127, 554–556.
Floor function, 37–38, 40–44.
Flow chart, 2, 18, 364–365.
Floyd, Robert W, xii, 17, 19, 20, 420, 473, 504, 552.
FLPL, 459–460.
FMUL (floating multiply), 127, 304.
Förstemann, Wilhelm, 488.
Ford, Donald Floyd, 511.
Forecasting, 221.
Forest: Zero or more trees, 306, 407, *see* Trees.
Formulas, algebraic, *see* Algebraic formulas.
logical, 346.
FORTRAN, 355, 459.
Fractional part, 38.
Fragmentation problem, 438–440, 450.
Franklin, Joel N., xii.
Free lattice, 346–347.
Free storage, *see* Available space.
Free trees, 362–371, 373, 377–378, 386–388, 396, 397.
enumeration, 377–378, 388, 396, 397.
minimum cost, 370.
Fridshal, R., 346.
Front of queue, 237.
FSUB (floating subtract), 127, 304.
Fundamental cycles in graph, 366–368, 376.
Furch, Robert, 117.
Future reference (in MIXAL), 149, 151.
restrictions on, 152.

Galler, Bernard Aaron, 353.
Galton, Francis, 558.
Games, solution of, 85, 270.
Gamma function, 48–51, 71, 78, 112, 115–116.
incomplete, 113–119.
Garbage collection, 254, 412–422, 438–440, 450, 454–455, 460, 541.
Gardner, Martin, 79.
Garwick, Jan Vaumund, 244, 456.
Gaskell, Robert Eugene, 85.
Gauss, Karl Friedrich, 48, 56, 94.
reduction algorithm for matrix inversion, 304.
gcd: Greatest common divisor.
Gelernter, Herbert Leo, 460.
General Electric Corporation, 456.
Generating functions, 81–83, 86–93, 97–104, 178, 239, 386, 388–389, 391–399, 404, 532–533.
asymptotic values from, 239, 395–396.
for a discrete probability distribution, 97–104, 178.
Geometric progression, sum of, 31, 87.
Gerberich, Carl Luther, 459.
Gill, Stanley, 226, 456.
Glaisher, James Whitbread Lee, constant, 499.
Gnedenko, Boris Vladimirovich, 103.
GO-button of MIX, 140, 208.
Goldbach, Christian, 48.
Goldberg, Joel, 522.
Golden ratio, 13, 18, 21, 79, 81–85, 613, 614.
Goldstine, Herman Heine, 18, 225.
Golomb, Solomon Wolf, 181.
Goncharov, Vasiliĭ Leonidovich, 103.
Good, Irving John, 374, 395, 482.
Gorn, Saul, 459.
Gould, Henry Wadsworth, xii, 62, 117, 484, 490.
Gower, John Clifford, 458.
Graham, Ronald Lewis, 605.
Graph, 362–372, 377–378, 406.
connected, 362.
directed, *see* Directed graph.
Graphical display, 159–160.
Greatest common divisor, 2, 4–6, 9, 14–15, 38–39, 42, 80–81.
Greatest integer function, *see* Floor function.
Grid, 228, 371.
Griswold, Ralph Edward, 460.

Haddon, Bruce Kenneth, 603.
Halāyudha, 52.
Hamilton, Dennis Eugene, xii.
Hamilton, Sir William Rowan, circuit, 374, 378.
Hamlet, Prince of Denmark, 228.
Hansen, James Rone, 459.
Harary, Frank, 406.
Hardware-oriented algorithms, 26, 249, 600.

Hardy, Godfrey Harold, 12, 490, 515.
Harmonic numbers, 73–78, 89, 110–111, 156.
  generating function, 89.
  table, 615–616.
Harmonic series, 74, 156–157.
Harrison, Michael Alexander, iv.
Hartmanis, Juris, 462.
Head of list, 272, 278, 286–287, 299–300, 322, 332, 336, 408–410, 443.
Heap sort, 552.
Height of vertex in free tree, 387.
Hellerman, Herbert, 458.
Henkin, Leon Albert, 17.
Hermite, Charles, 48.
Heyting, Arend, 405.
Hilbert, David, matrix, 37.
HLT (halt), 132, 139.
Hoare, Charles Antony Richard, 457.
Holmes, Thomas Sherlock Scott, 463.
Holt Hopfenberg, Anatol Wolf, 459.
Honeywell H800, 120.
Hopper, Grace Murray, 458.
Huffman, David Albert, 402–405.
Hurwitz, Adolf, 42.
  generalized binomial formula, 398, 488.

IBM 650, i, 120, 226, 456, 523.
IBM 701, 226.
IBM 705, 227.
IBM 709, 120, 523.
IBM 7070, 120.
Identity permutation, 161, 172.
Iff: If and only if.
Iliffe, John Kenneth, 460.
Illiac I, 226.
IN (input), 132–133, 211–212.
In-degree of vertex, 371.
INCA (increase A), 129, 206.
Incidence matrix, 267.
Inclusion and exclusion principle, 178–179, 181.
Incomplete gamma function, 113–119.
INCX (increase X), 129, 206.
INC1 (increase 1), 129, 206.
Indentation, 309.
Index: A number which indicates a particular element of an array (sometimes called a "subscript"), 3–4, 295–298, 310, 313, 315.
Index register, 122–123, 153, 263.
  modification of MIX instructions, 123, 248.
Indirect addressing, 248–249, 303.
Induction, mathematical, 11–21, 32.
  generalized, 20–21.
Infinite series: A sum over infinitely many values.
Infinite trees, 314–315, 381–385.
Infinity lemma, 381–385.
Information: The meaning associated with data, the facts or concepts represented by data; often used also in a narrower sense as a synonym for "data," or in a wider sense to include any concepts which can be deduced from data.
Information structure, *see* Data structure.
Ingerman, Peter Zilahy, xii.
Initial vertex of arc, 371.
Input, 5, 211–225.
  anticipated, 212.
  buffering, 212–225.
  operators of MIX, 132–134, 211–212.
Input-restricted deque, 235–239, 415.
Insertion of node: Entering it into a data structure.
  into available space list, *see* Liberation.
  into deque, 248, 266, 271, 294.
  into doubly linked list, 279, 288, 294, 442, 444–445.
  into linear list, 235.
  into linked list, 231–232, 252, 274, 279, 288, 294, 301–302, 357–358, 442, 444–445.
  into queue, 237–238, 240–241, 257, 262, 271.
  into tree, 325, 331, 357–358.
  into two-dimensional list, 301–302.
  onto stack, 237–238, 240–241, 243–244, 254–256, 265–266, 271, 276, 279, 323, 415–416.
Instruction, machine language: A code which, when interpreted by the circuitry of a computer, causes the computer to perform some action.
  in MIX, 123–137.
  symbolic form, 123–124, 141–153.
INT (interrupt), 225.
Integer, 21.
Integration, 89.
  related to summation, 108–112, 116.
Interchange of values, 3, 179.
Interchanging the order of summation, 28–30, 33, 41.
Interest, compound, 23.
Interlock time: Delay of one part of a system while another part is busy completing some action.
Internal path length, 399–400, 405.
Interpreter (interpretive routine), 197–208, 226, 338.
Interrupt, 224–225.
Inverse (modulo m), 40.
Inverse of matrix, 35–37, 72, 304.
Inverse of permutation, 172–175, 180.

Inversion problem, 63.
Invert a linked list, 266.
I/O: Input or output, 211.
IOC (input-output control), 133.
IPL, 226, 229, 457, 458, 459, 460.
Irons, Edgar Towar, xii.
Irreflexive relation, 258.
Isolated vertex, 374.
Iverson, Kenneth Eugene, 37, 117, 458, 459.
I1-register of MIX, 122, 138.

J-register of MIX, 122, 130, 139, 182–183, 185, 208–210.
JACM: Journal of the ACM, a publication of the Association for Computing Machinery.
Jacquard, Joseph Marie, loom, 225.
JAN (jump A negative), 130, 206.
JANN (jump A nonnegative), 130, 206.
JANP (jump A nonpositive), 130, 206.
JANZ (jump A nonzero), 130, 206.
JAP (jump A positive), 130, 206.
Jarden, Dov, 85.
JAZ (jump A zero), 130, 206.
JBUS (jump busy), 133, 208, 212, 222.
JE (jump on equal), 130, 205–206.
Jenkins, D. P., 459.
JG (jump on greater), 130, 205–206.
JGE (jump on greater-or-equal), 130, 205–206.
JL (jump on less), 130, 205–206.
JLE (jump on less-or-equal), 130, 205–206.
JMP (jump), 130, 183, 205.
JNE (jump on unequal), 130, 205–206.
JNOV (jump on no overflow), 130, 138, 205.
Jodeit, Jane G., 461.
Johnson, Lyle Robert, 458, 459.
Joke, 53, 196.
Jordan, Camille, 405.
Jordán, Károly (= Charles), 68.
Jordan, Wilhelm, reduction algorithm for matrix inversion, 304.
Josephus' problem, 158–159.
JOV (jump on overflow), 130, 138, 205.
JRED (jump ready), 133, 218–219.
JSJ (jump, save J), 130, 185, 205.
Jump operators of MIX, 130.
JXN (jump X negative), 131, 206.
JXNN (jump X nonnegative), 131, 206.
JXNP (jump X nonpositive), 131, 206.
JXNZ (jump X nonzero), 131, 206.
JXP (jump X positive), 131, 206.
JXZ (jump X zero), 131, 206.
J1N (jump 1 negative), 131, 206.
J1NN (jump 1 nonnegative), 131, 206.
J1NP (jump 1 nonpositive), 131, 206.
J1NZ (jump 1 nonzero), 131, 206.
J1P (jump 1 positive), 131, 206.
J1Z (jump 1 zero), 131, 206.

Kahn, Arthur B., 265.
Kahrimanian, Harry George, 458.
Kaucký, Josef, 62.
Kepler, Johann, 79.
Kilmer, Joyce, 228.
Kirchhoff, Gustav Robert, 405.
  law of conservation of flow, 95, 167–168, 265, 276, 323, 364–370, 374, 379–380.
Knopp, Konrad, 47, 75, 110.
Knotted List, 458.
Knowlton, Kenneth C., 460.
Knuth, Donald Ervin, ii, xiii, 198, 456, 578, 587.
Knuth, Ervin Henry, xii.
Knuth, Jill Carter, xii.
Kolmogorov, Andreĭ Nikolaevich, 103.
König, Dénes, 381, 382, 385, 405.
Kozelka, Robert M., 539.
Kramp, Christian, 48.
Kronecker, Leopold, delta notation, 60.
Kruskal, Joseph Bernard, 385.
Kummer, Ernst Eduard, 68.

La Loubère, Simon de, 158.
Labeled trees, enumeration of, 389–395, 397–398.
Lagrange, Joseph Louis, comte, 27.
  identity, 34.
  inversion formula, 392, 588.
Lamé, Gabriel, 79, 405.
Language: A set of strings of symbols, usually accompanied by conventions for assigning a "meaning" to each string in the set.
Laplace, Pierre Simon, marquis de, 86.
  transform, 86, 93.
Large programs, writing, 187–189.
Last-in-first-out, 236, 452, *see* Stack.
  almost, 447, 454.
Lattice, free, 346–347.
Lawson, Harold W., Jr., 432.
LDA (load A), 124–125, 204–205.
LDAN (load A negative), 125, 135, 204–205.
LDX (load X), 125, 135, 204–205.
LDXN (load X negative), 125, 135, 204–205.
LD1 (load 1), 125, 135, 204–205.
LD1N (load 1 negative), 125, 135, 204–205.
Left subtree in a binary tree, 309.
Legendre, Adrien Marie, 48, 49.
  symbol, 43.
Leibnitz (= Leibniz), Gottfried Wilhelm, freiherr von, 2, 49.
Leonardo of Pisa, 78.
Letter frequencies in English, 155.
Level of node in tree, 305, 314.

Level order, 350.
LeVeque, William Judson, 465.
Lévy, Paul, 103.
Lexicographic order, 20, 296–297, 303, 332.
L'Hospital, Guillaume François Antoine de, marquis de Sainte-Mesme, rule of, 102.
Liberation of reserved storage, 253, 275, 290, 411–413, 419–420, 438–442, 444–445, 449–450, 452–455.
LIFO, 236, 458, *see* Stack.
Lilius, Aloysius, 155.
Lineal chart, 307–308.
Linear lists, 228, 234–304.
Linear ordering, 20, 259, 267.
  embed partial ordering into, 259, *see* Topological sorting.
  of trees, 331, 332, 345.
Linear recurrence, 87.
Link, 229–231.
  diagram of, 230–231.
  field, purpose of, 231, 431, 461.
  manipulation, avoiding errors in, 256–257.
  null, 230.
Link variable, 231–233.
Linkage: Manner of setting links.
  circular, 270–277, 300, 355, 409–410, 458.
  coroutine, 190, 196, 220, 288–289.
  double, 278, 286, 355, 410.
  orthogonal, 286, 298–300.
  straight, 230, 251, 256, 410, 416.
  subroutine, 182–183, 189.
  two way, 278, 286, 355, 410.
Linked allocation of tables, 230–231, 251–253.
  array, 286, 299–300.
  contrasted to sequential allocation, 251–253, 433 (exercise 5).
  linear list, 230–231, 251–258, 261–263, 265, 270–273, 276–277, 278–279, 330, 416, 433.
  tree structures, 315–316, 319–322, 325, 333–334, 351–359.
Linked-memory philosophy, 251–253, 435.
Linking automaton, 462–463.
LISP, 229, 459, 603.
List: Ordered sequence of zero or more elements.
  circular, 270–277, 409–410, 458.
  doubly linked, 278–279, 285–288, 294–295, 409–411, 441–442, 443–445, 453, 458.
  linear, 228, 234–304.
  of available space, *see* Available space list.
List (capital-List) structures, 312–313, 315, 406–422.
  copying, 421.
  diagrams of, 312–313, 315, 407.
  distinguished from lists, 229, 409, 411.
  equivalence between, 421–422.
  notations for, 312–313, 315, 407.
  representation of, 408–411, 417, 459–460.
List head, 272, 278, 286–287, 299–300, 322, 332, 336, 408–410, 443.
List processing systems, 229, 411, 459–460.
Listing, Johann Benedict, 405.
Literal constants in MIXAL, 146, 151.
LLINK: Link to the left.
  in binary tree, 315, 319–325, 328–332.
  in doubly linked list, 278–279, 285–289.
  in List, 410–411.
  in tree, 337, 347–349, 352, 355, 380.
Lloyd, Stuart Phinney, 180, 181.
Loading operators of MIX, 124–125, 135, 204–205.
Loading routine, 139–140, 225, 268.
LOC, 231–232.
Local symbols in MIXAL, 147, 149, 153.
Locally defined function in tree, 351, 362.
Location: The memory address of a computer word or node; or the memory cell itself.
Location counter in MIXAL, 150–151.
Location field of MIXAL line, 141–142, 148.
Logarithm, 22–26.
  binary, 22, 25.
  common, 22.
  natural, 23, 25, 26.
  power series, 89–90.
Logical formulas, 346.
Loop detection, 268.
Lovelace, Ada Augusta, countess of, 1.
LSON, 352, 359.
LTAG, 319–320, 332, 348–349, 352.
Lucas, Édouard, 68, 79, 80, 270.
Lukasiewicz, Jan, 336.
Lynch, William Charles, xii, 581.

MacGinitie, Gordon Frank, 503.
Machine language: A language which directly governs a computer's actions, as it is interpreted by a computer's circuitry, 120.
  symbolic, 141, *see* Assembly language.
MacMahon, Major Percy Alexander, 489.
Macro instruction: Specification of a pattern of instructions and/or pseudo-operators which may be frequently repeated within a program.
Madnick, Stuart E., 460.
Magic square, 158.
Magnetic tape, 132–134, 462.
Mark I calculator, 225.
Marking algorithms: Algorithms which "mark" all nodes that are accessible

from some given nodes, 268–269, 413–422.
Markov, Andreĭ Andreevich (the elder), 380.
process, 250 (exercise 13), 380–381.
Markov, Andreĭ Andreevich (the younger), 9.
Markowitz, Harry Max, 460.
Math. Comp.: Mathematics of Computation, a journal published by the American Mathematical Society.
Mathematical induction, 11–21, 32.
generalized, 20–21.
Matrix, 228, 295–296.
Cauchy, 36–37.
combinatorial, 36–37, 584.
determinant of, 35–37.
Hilbert, 37.
incidence, 267.
inverse of, 35–37, 304.
multiplication, 304.
representation of, 154, 295–304.
sparse, 299–304.
transpose of, 180.
triangular, 297–298, 303.
Vandermonde, 36–37.
Matrix, Dr. Irving Joshua, 33, 34.
Maurolico, Francesco, 17.
Maximum, algorithm to find, 95, 141, 182.
McCall's, v.
McCarthy, John, 459, 460.
McCracken, Daniel Delbert, xii–xiii.
McEliece, Robert James, 476, 481.
McNeley, John Louis, xiii.
Mean (average) of a probability distribution, 96, 98–99, 101.
Meek, H. V., 227.
Meggitt, John E., 470.
Memory: Part of a computer system used to store data, 122, 195, 234.
cell of, 123.
types of, 195, 234, 461–462.
update, 295.
Memory map, 435–436, 448–449.
Merging, 402.
Merner, Jack Newton Forsythe, xiii, 226.
Metcalfe, Howard Hurtig, xiii.
Military game, 270.
Minimum path length, 400–405.
Minimum wire length, 370–371.
Minsky, Marvin Lee, 422.
Mitchell, William Charles, 520.
MIX computer, xi, 120–140.
assembly language for, 141–153.
extensions to, 139, 225–226, 248–249, 454.
instructions, form of, 123.
instructions, summary, 136–137.
simulator of, 198–208.
MIXAL: MIX Assembly Language, 141–153.
Mixed-radix number system, 297.
Mock, Owen Russell, 227.
mod, 38.
modulo, 38.
Moments of probability distribution, 103.
Monitor routine, 208, *see* Trace routine.
Monte Carlo method: Experiments with random data, 446.
Moon, John Wesley, 406.
Mordell, Louis Joel, 42.
Mother, 307, *see* Father.
Motzkin, Theodor Samuel, 531.
Mouse algorithm, *see* Traversal.
MOVE, 131, 189, 207.
MOVE CORRESPONDING in COBOL, 425, 429–431, 434.
Moyse, Alphonse, Jr., 377.
MUG: MIX User's Group, 627.
MUL (multiply), 127–128, 204.
Multilinked structures, 228, 285–286, 356–359, 423–434, 457.
Multinomial coefficient, 64, 394.
Multinomial theorem, 64.
Multipass algorithm, 194–196, 197–198.
Multiple: x is a multiple of y if y is a divisor of x, i.e., $x = ky$ for some integer k.
Multiple entrances to subroutines, 185–186
Multiple exits from subroutines, 186.
Multiple precision arithmetic, 198.
Multiplication of permutations, 161–164, 169–170, 371.
Multiplication of polynomials, 274, 276–277.
Multiplicative function, 41.
Multiway decisions, 153.

Napier, John, 23.
National Science Foundation, xii.
Natural correspondence between binary trees and forests, 333–334, 345.
Natural logarithm, 23.
Naur, Peter, xiii, 18.
Negative: Less than zero (*not* zero).
Nested parentheses, 309.
Nested sets, 309, 314.
Nesting store, 236.
Network, 258, *see* Graph.
Neville, Eric Harold, 585.
Newell, Allen, 226, 456–457, 459.
Newton, Sir Isaac, 22, 56.
identities, 494.
Nicomachus of Gerasa, 19.
Nil link, *see* Null link.
Noah, 308.

Node: Basic component of data structures, 229.
address of, 229.
diagram of, 230.
link to, 229.
notations for fields, 231–233, 457–458.
size of, 240, 254, 296, 435, 452.
variable-size, 435–455.
NODE, 232.
Node variable, 232–233.
Nonnegative: Zero or positive.
NOP (no operation), 132.
Normal distribution, 102, 103.
Notations, index to, 607–611.
Notes on the exercises, xvii–xix.
Null link, 230–231.
in tree, 315–316, 319–320, 329.
NUM (convert to numeric), 134.
Number, definitions, 21.
Number system: A language for representing numbers.
binomial, 72.
decimal, 21.
Fibonacci, 85.
mixed-radix, 297.
phi, 85.
Number theory, elementary, 38–44.

O-notation, 104–108.
O'Beirne, Thomas Hay, 155.
Oettinger, Anthony G., 459.
Office of Naval Research, xii, 226.
Okada, Satio, 377.
One-plus-one address computer, 456.
One-way equalities, 105–107.
One-way linkage, *see* Straight linkage, Circular linkage.
Onodera, Rikio, 377.
Open subroutine, *see* Macro instruction.
Operation code field, of MIX instruction, 123.
of MIXAL line, 142, 148, 151.
Optimal search procedure, 402.
Order of succession to throne, 335.
Ordered tree, 306, 373, 388–389, *see* Tree.
Ordering: A transitive relation between objects of a set.
lexicographic, 20, 296–297, 303, 322.
linear, 20, 259, 267.
linear, of trees, 331, 332, 345.
partial, 258–262, 266–267, 314, 345.
well, 20–21, 332.
Ore, Øystein, 406.
Oresme, Nicole, 22.
Oriented binary tree, 396.
Oriented cycle in directed graph, 371.
Oriented path in directed graph, 371, 376.
Oriented trees, 306, 353–355, 359, 372–379, 386, 389.
canonical representation, 390.
enumeration, 386, 389–397.
representation of, 353–355.
root changed in, 376.
ORIG (origin), 142, 148, 151.
Orthogonal lists, 295–304.
Otter, Richard Robert, 388, 395.
OUT (output), 132–133, 222.
Out-degree of vertex, 371.
Output, 5, 211, 215–225.
buffering, 215–225.
operators of MIX, 132–134.
Output-restricted deque, 235–239, 266, 271.
OVERFLOW, 241–248, 253–254, 265–266, 274, 451.
Overflow toggle of MIX, 122, 127, 129, 130, 138, 205, 210, 224.

Packed data: Data which has been compressed into a small space, e.g., by putting two or more elements of data into the same word of memory, 124, 153.
Paging, 451.
Pall, Gordon, 518.
Parallelism, 293, 295, *see* Discrete system simulation.
Parameters of subroutines, 183, 185.
Parker, William Wayne, xiii.
Partial field designations in MIX, 122–123, 203.
Partial fractions, 62, 71, 82.
Partial ordering, 258–262, 266–267, 314, 345.
Partitions of a set, 73, 481.
Partitions of an integer, 12, 32, 92, 93.
Pascal, Blaise, 17, 52.
triangle, 52, 68–69, 72, 84, *see* Binomial coefficients.
Pass, in a program, 194–196.
Path, in a graph or directed graph, 362, 372.
oriented, 371.
random, 380–381.
simple, 362, 369, 371, 376.
Pawlak, Zdzisław, 459.
PDP-4, 120.
Pedigree, 307–308.
Peripheral device: An I/O component of a computer system, 132.
Perlis, Alan J., 319, 459.
Permanent of a square matrix, 50.
Permutations, 44–45, 49, 96–97, 160–164, 169–170, 172–181, 238–239, 329, 371.
inverse of, 172–175, 180.

multiplication of, 161–164, 169–170, 371.
notations for, 160–161.
PERT network, 258–259.
Peters, Johann (= Jean) Theodor, 615.
Peterson, William Wesley, xiii.
Phi, 79, *see* Golden ratio.
number system, 85.
Phidias, 79.
Philco S2000, 120.
Pile, 236.
Pilot ACE computer, 226.
Pisano, Leonardo, 78.
Pivot step, 301–302, 304.
PL/I, 433, 552.
Planar tree, *see* Ordered tree.
Playing cards, 49, 68, 229–233, 377.
Plex, 457.
Pointer, *see* Link.
Pointer variable: A variable whose values are links.
Poisson, Siméon Denis, distribution, 103, 519
Polish notation, *see* Prefix notation, Postfix notation.
Polonsky, Ivan Paul, 460.
Pólya, György (= George), 17, 395, 406, 494.
Polynomials, 55, 65, 105.
addition of, 273–276, 355–359, 361.
Bernoulli, 42, 109–112.
Chebyshev, 493.
differences of, 64.
multiplication of, 274, 276–277.
representation of, 273, 277, 356–359.
Pool of available nodes, *see* Available space list.
Pooled buffers, 224.
Pop up a stack: Delete its top element, 237–238, 240–241, 243, 255–256, 265–266, 271, 276, 278–279, 323, 415–416.
Positive: Greater than zero (*not* zero).
Postfix notation, 336, 351, 362.
Postorder for binary tree, 316–320, 328–330.
Postorder for tree, 334–336, 338, 345, 350–351.
Postorder sequential representation of trees, 350–351, 362.
Power of number, 21–22.
factorial, 70, 609.
Power series: Sum of the form $\Sigma_{k\geq 0} a_k z^k$, *see* Generating function.
convergence of, 86.
manipulation of, 115.
Prefix notation, 336, 359, 587–588.
Preorder for binary tree, 316–317, 326–331.
Preorder for tree, 334–336, 348–349, 359, 459.
Preorder sequential representation of trees, 348–349.
with degrees, 359, 459.
Prim, Robert Clay, 370.
Prime numbers, 18, 39, 41, 43–44, 46–47, 68, 143–145, 153.
algorithm to compute, 143–145, 153.
factorization into, 41, 46–47, 68.
Printer, 132–133.
Prinz, D. G., 226.
Probability distribution: A specification of probabilities which govern the value of a random variable, 96–104, 178.
average ("mean") value of, 96, 98–99, 101.
generating function for, 98–101, 103–104.
variance of, 96, 98–99, 101.
Procedure, *see* Subroutine.
Procedure for reading this set of books, xiv–xvi.
Program: Representation in some precise, formalized language of a computational method, 5.
Programming language: A precise, formalized language in which programs are written.
Programs, hints for construction of, 187–189, 293.
Progression, arithmetic, sum of, 11, 13, 31, 55.
Progression, geometric, sum of, 31, 87.
Proof of algorithms, 14–20, 318–319, 420.
Proper divisor, *see* Divisor.
Propositional calculus, 346.
Prüfer, Heinz, 406.
Pseudo-operator: A construction in a programming language which is used to control the translation of that language into machine language, 142.
Psi function, 94, 490, 616.
Purdom, Paul Walton, Jr., xiii.
Push down list, 236, *see* Stack.
Push down onto a stack: Insert a new top element, 237–238. 240–241, 243–244, 254–256, 265–266, 271, 276, 279, 323, 415–416.

q-binomial theorem, 72.
Quadratic reciprocity law, 44.
Qualification of names, 423–434.
Quasi-parallel processing, 293, *see* Discrete system simulation.
Queue, 235–239, 240–241, 248–249, 261–263, 271, 458.
deletion from front, 240–241, 257–258, 262–263, 271.
insertion at rear, 240–241, 257, 262–263, 271.

Queue (cont.)
linked allocation, 257, 270–271, 278.
sequential allocation, 240–241, 248–249.
Quick, Jonathan Horatio, 498.

Rāmānuian Aivaṅgār, Srīnivāsa, 12, 117. 119.
Ramus, Christian, 70.
Randell, Brian, 198.
Random path, 380–381.
Raney, George N., 392, 394, 588.
Raphael, Bertram, 459.
Rational number, 21, 157.
RCA 601, 120.
Reading: Doing input, 211.
Real number, 21.
Real time, 422, 442.
Reallocate sequentially stored tables, 244–246.
Rear of queue, 237–238.
Recipe, 6.
Reciprocity formulas, 43–44.
Recomp II, 120.
Record: A set of data that is input or output at one time, 132–133; *see also* Node, 229.
Records, blocking of, 214, 222.
Rectangular arrays, 295–304.
Recurrence relation: A rule which defines each element of a sequence in terms of the preceding elements.
Recursive definition, 305, 309, 312, 315–317, 334.
Recursive List, 313.
Recursive use of subroutine, 187.
Ref, *see* Link.
Reference, 229, *see* Link.
Reference counter technique, 412–413, 460.
Reflexive relation, 258, 353.
Registers: Portions of a computer's internal circuitry in which data is processed; the most accessible data kept in a machine appears in its registers.
of MIX, 122.
saving and restoring contents of, 184, 194, 224–225.
Regular directed graph, 378.
Relation: A property which holds for certain sets (usually ordered pairs) of elements; for example, "$<$" is a relation defined for ordered pairs (x, y) of integers, and the property "$x < y$" holds if and only if x is less than y.
antisymmetric, 258.
asymmetric, 258.
equivalence, 353.
irreflexive, 258.
reflexive, 258, 353.
symmetric, 353.
transitive, 105, 258, 353, *see* Ordering.
Relatively prime integers, 38–41.
RELEASE a buffer, 215, 218, 224.
Remove from structure, *see* Deletion.
Rényi, Alfréd, 590.
Repacking, 243–246.
Replacement operation, 3.
Replicative function, 42.
Representation (inside a computer), methods for choosing, 234–235, 423.
of algebraic formulas, 335–336, 458.
of arrays, 154, 296–300.
of binary trees, 315–316, 319–322, 325, 332, 401.
of deques, 248, 278.
of directed graphs, 380.
of forests, 333, 347–362.
of Lists, 408–411, 417, 459–460.
of oriented trees, 353, 376.
of polynomials, 273, 277, 356–359.
of queues, 240–241, 256, 270, 278, 286.
of stacks, 240–241, 251, 270, 272, 278.
of trees, 333–334, 347–362, 459.
Reservation of free storage, 253–254, 263, 266, 275, 289, 436–438, 444, 449–450, 452–454.
Reversion storage, 236.
Riemann, Georg Friedrich Bernhard, 74.
Right subtree in a binary tree, 309.
Right-threaded tree structure, 325, 331, 336, 380.
Ring structure, 355.
Riordan, John, 397, 406, 492, 532, 590.
RLINK: Link to the right.
in binary tree, 315, 319–325, 328–332.
in doubly linked list, 278–279, 285–289, 315, 319–325.
in List, 408, 410–411.
in tree, 337, 347–349, 352, 355, 380, *see* BROTHER link.
Robertson, James C., xiii.
Robson, J. M., 605.
Rodrigues, Olinde, 405.
Roll, 236.
Root of number, 21, 25.
Root of tree, 305–309, 314, 372–373, 381, 383.
change of, 376.
Rooted directed graph, 372, 377.
Rooted tree, 372, *see* Oriented tree.
Ross, Douglas Taylor, xiii, 451, 457, 461.
Rothe, Heinrich August, 62.
Rounding, 40, 82, 156.
Row major order, 296.
RTAG, 319–320, 331, 337, 349, 350.
Running time, *see* Execution time.
Russell, Lawford John, 198.

Saddle point, 155.
Salton, Gerard Anton, 350, 458.
SAME AREA, 572.
Sammet, Jean Elaine, 346, 461.
Scaled decimal arithmetic, 156–157.
Scherk, Heinrich Ferdinand, 488.
Schlatter, Charles Fordemwalt, 458.
Schlatter. William Joseph, 458.
Schorr, Herbert, 417, 420.
Schorr-Kon, Jacques J., 9.
Schorre, Dewey Val, xiii.
Schröder, Ernst, 587.
Schützenberger, Marcel Paul, xiii.
Schwartz, Eugene Sidney, 404.
Schwarz, Hermann Amandus, inequality: $(\Sigma a_k b_k)^2 \leq (\Sigma a_k^2)(\Sigma b_k^2)$ [due to Buniakovskiĭ, 1859], *see* Lagrange's identity.
Schwenk, Allen John, 493.
Schweppe, Earl Justin, xiii, 458.
SCOPE link, 349, 434.
Scroll, 236.
SDS 920, 120.
Segner, Johann Andreas von, 405, 531.
Seiden, Esther, 518.
Selfridge, John L., 77.
Semi-invariants of a probability distribution, 101–103.
Sentinel: A special value placed in a table, e.g., to mark the boundaries of the table, designed to be easily recognizable by the accompanying program.
Sequential (consecutive) allocation of tables, 240.
  array, 154, 295–298.
  contrasted to linked allocation, 251–253, 433 (exercise 5).
  linear list, 240–251, 261–263, 323, 414–416.
  tree structures, 347–350, 359–362, 401, 434.
Series, infinite: An infinite sum.
Sets, partition of, 73, 481.
Shaw, Christopher Joseph, xiii.
Shaw, John Clifford, 456–457.
Shelf, 236.
Shell, Donald Lewis, xiii.
Shepp, Lawrence Alan, 180, 181.
Shift operators of MIX, 131, 207.
Shih-chieh, Chu, 52.
Sibling, 307, 347, *see* Brother.
Sister, 307, *see* Brother.
Similar trees, 325–327, 345 (exercise 10).
Simon, Herbert Alexander, 226, 456–457.
Simple oriented path, 371, 376.
Simple path, 362, 369.
Simplification, algebraic, 339, 346.
SIMSCRIPT, 460.
Simulated time, 281, 285, 451.
Simulation: Imitation of some system.
  continuous, 279.
  discrete. 199, 279–295.
  of one computer on another, 198–208.
  of one computer on itself, 208–211.
Singleton cycle of permutation, 160–161, 164, 168, 177–179.
Skalsky, Michael, 484.
SLA (shift left A), 131, 207.
SLAX (shift left AX), 131, 207.
SLC (shift left AX circularly), 131, 207.
SLIP, 229, 458, 459, 460.
Smallest-in-first-out, 552.
SNOBOL, 460.
Solitaire game, 377.
Son, in a tree structure, 307, 333–334, 347, 352, 426–432.
Sorting, topological, 258–268, 345, 376, 397.
Sparse matrix, 299–304.
Speedcoding, 226.
SRA (shift right A), 131, 207.
SRAX (shift right AX), 131, 207.
SRC (shift right AX circularly), 131, 207.
STA (store A), 125–126, 205.
Stack, 235–239, 240–250, 254–256, 265–267, 271, 276, 317–319, 323–324, 329–330, 414–417, 427–428.
  deletion ("popping"), 237–238, 240–241, 243–244, 255–256, 265–266, 271, 276, 278–279, 323, 415–416.
  insertion ("pushing"), 237–238, 240–241, 243–244, 254–256, 265–266, 271, 276, 279, 323, 415–416.
  linked allocation, 254–256, 265–267, 271, 276, 278–279, 330, 416.
  pointer to, 240, 243, 254.
  sequential allocation, 240–250, 323, 414–415.
Standard deviation of probability distribution: The square root of the variance, an indication of how much a random quantity may be expected to deviate from its mean value, 96, 99, 102.
Stearns, Richard Edwin, 462.
Stegun, Irene Anne, 66, 92, 615.
Stickelberger, Ludwig, 50.
Stirling, James, 46–48, 72, 86, 111, 178.
  approximation, 46, 49, 71, 111–112, 113, 115–116, 538.
Stirling numbers, 65–68, 70, 73, 77, 90, 94 (exercise 18), 97, 102, 501, 578.
  combinatorial interpretations, 73, 176.
  generating functions, 90.
  tables of, 66.
STJ (store J), 126, 142, 183, 205.

Storage allocation: Choosing memory cells in which to store data, *see* Available space list, Dynamic storage allocation, Linked allocation, Sequential allocation.
Storage mapping function: The function whose value is the location of an array node, given the indices of that node, 240, 296–298, 303.
Store: British word for "memory."
Storing operators of MIX, 125–126, 205.
Straight linkage, 230, 251, 256, 410, 416.
String: A finite sequence of zero or more symbols, 8–9, 85, *see* Linear list.
  concatenation, 272.
  manipulation, 460, 461.
Strongly connected directed graph, 372, 377.
Structure, how to represent, 234–235, 423–432, 461, *see* Representation.
STX (store X), 126, 205.
STZ (store zero), 126, 205.
ST1 (store 1), 126, 205.
SUB (subtract), 127, 128, 204.
Subroutine, 154, 156, 182–189, 190–192, 198, 202–203, 207, 225–226, 288–289.
  allocation, 268–269.
  closed, *see* Subroutine.
  history, 225–226.
  linkage, 182–183, 187.
  open, *see* Macro instruction.
Subscript, 3, *see* Index.
Substitution operation, 3.
Subtree order, 459.
Subtrees, 305–307.
  enumeration of, 377–378.
  free, 365–368.
Summation, 26–37.
  by parts, 43 (exercise 42), 75, 77.
  Euler's formula, 108–112, 116, 119.
  interchange of order, 28–30, 33, 41.
  of arithmetic progression, 11, 13, 31, 55.
  of binomial coefficients, 54–64, 68–73.
  of geometric progression, 31, 87.
  relation to integration, 108–112, 116.
Swapping buffers, 143–144, 155, 213–215, 222.
Sward, Gilbert L., 346.
Swift, Charles James, 227.
Switching table, 154, 200–201, 204–205.
Symbol manipulation: A general term for data processing, usually applied to nonnumeric processes such as manipulation of strings or algebraic formulas.
Symbol table algorithms, 172, 263, 425.
Symbolic machine language, *see* Assembly language.
Symmetric function, elementary, 93, 94, 494.
Symmetric order for binary tree, 317, *see* Postorder.
Symmetric relation, 353.
Synchronous discrete simulation, 280, 295.
System: A set of objects or processes which are interconnected or which interact with each other.
System/360, 120, 523.
Szekeres, George, 590.

Table-driven program, *see* Interpreter, Switching table.
Tables, arrangement of, inside a computer, *see* Representation.
Tables of numerical quantities, 66, 613–616.
Tag field in tree node, 319, *see* LTAG, RTAG.
Takács, Lajos, 532, 588.
Tape, 132–133.
Taussky, Olga, xiii.
Tautology, 346.
Taylor, Brook, formula with remainder, 113.
Temp storage: Part of memory used to hold a value for a comparatively short time while other values occupy the registers, 188.
Terminal node of tree, 305, 315.
Terminology, 237, 307, 362.
Ternary tree, 332, 396, 401, 404–405.
Tetrahedral array, 298, 303, *see* Binomial number system.
Theile, Thorvald Nicolai, 101.
Theory of automata, 462–463.
Theory of algorithms, 7, 9.
Thornton, Charles, 319, 459.
Thread an unthreaded tree, 330–331.
Thread links, 319–321, 334.
Threaded trees, 319–325, 329–332, 334, 420, 459.
  compared to unthreaded, 324, 420.
  insertion into, 325.
  list head in, 322, 336.
Three-address code, 336, 458.
Tiling the plane, 382–385.
Time, simulated, 281, 285, 451.
Time taken by program, *see* Execution time.
Timer, *see* Clock.
Todd, John, xiii, 474.
Todd, Olga Taussky, xiii.
Tonge, Frederic McLanahan, 459.
Top of stack, 237–238.
Top-down process, 362.
Topological sorting, 258–268, 345, 376, 397.
Torelli, Gabriele, 70, 487.
Totient function $\varphi(n)$, 41, 181 (exercise 27).
Trace routine, 208–211, 226–227, 293.
Traffic signal, 157–158.

Transfer instruction: A "jump" instruction.
Transitive relation, 105, 258, 353, *see* Ordering.
Transpose of matrix, 180.
Traversal of tree structure, 316–324, 328–332, 334–335, 345.
Tree function, evaluation of, 351, 362.
Trees, 228, 305–422, 426–434.
  binary, *see* Binary trees.
  comparison of different types, 306, 373.
  complete t-ary, 401.
  construction of, 339, 342, 426–428.
  copying of, 327–328, 332, 346.
  definition of, 305–306, 309, 312, 314–315, 363, 371, 372.
  deletion from, 357–358.
  Dewey notation for, 310–311, 314–315, 345, 381–382.
  diagrams of, 306–307, 309.
  embedding of, 347, 385.
  enumeration of, 377–378, 385–399, 404.
  equivalent, 326, 331, 345 (exercise 10).
  erasing of, 331.
  free, *see* Free trees.
  history, 405–406, 458–459.
  index notation for, 310, 312, 313, 315.
  infinite, 314–315, 381–385.
  insertion into, 325, 331, 357–358.
  labeled, enumeration of, 389–395, 397–398.
  linear ordering for, 331, 332, 345.
  linked allocation for, 315–316, 319–322, 325, 333–334, 351–359.
  mathematical theory of, 362–406.
  n-tuply rooted, 306, *see* Forest.
  ordered, 306, 373, 388–389, *see* Trees.
  oriented, *see* Oriented trees.
  representation of, 333–334, 347–362, 459.
  right-threaded, 325, 331, 336, 380.
  sequential allocation for, 347–350, 359–362, 401, 434.
  similar, 325–327, 345 (exercise 10).
  t-ary, 332, 396, 401, 404–405.
  ternary, 332, 401, 405.
  threaded, *see* Threaded trees.
  traversal of, 316–324, 328–332, 334–335, 345.
  triply linked, 352, 359, 426–434.
  unordered, *see* Oriented trees.
  unrooted, 363, *see* Free trees.
Triangular matrix, 297–298, 303.
Tricomi, Francesco Giacomo Filippo, 118.
Trigonometric functions, 42, 470.
Trilling, Laurent, 460.
Triply linked tree, 352, 359, 426–434.
Tritter, Alan L., 572.
Turing, Alan Mathison, 226, 458.
  machine, 9, 226, 462–463.
Tutte, William Thomas, 378.
Twain, Mark (= Clemens, Samuel Langhorne), 53.
Tweedie, Charles, 86.
Two-way linkage, 278, 286, 410.

Uhler, Horace Scudder, 479.
UNDERFLOW, 241–242, 255, 265–266, 271.
Uniform distribution: A probability distribution in which every value is equally probable, 265–266, 271.
UNIVAC 1, 147.
UNIVAC 3, 120.
UNIVAC 1107, 120.
UNIVAC SS80, 120.
Unpacking, 153.
Update-memory, 295.

van Aardenne-Ehrenfest, Taniana, 375, 578.
van der Waerden, Bartel Leendert, 385.
Vandermonde, Alexander Théophile, matrix, 36–37.
Varga, Richard S., iv.
Variable: A quantity in a program which may possess different values as the calculation proceeds, 3, 231.
  link or pointer, 231.
Variable-size nodes, 435–455.
Variance of a probability distribution, 96, 98, 99, 101.
Vector, *see* Linear lists.
Vertex in a graph, 362, 371.
  isolated, 374.
Victorius of Aquitania, 155.
Visit a node, 318.
von Ettingshausen, Andreas, 52.
von Neumann, John, 18, 225, 456.
von Staudt, Karl Georg Christian, 405.

W-value (in MIXAL), 150–151.
Wait list, *see* Agenda.
Waite, William McCastline, 417, 420, 603.
Wallis, John, 22.
  product, 50, 112.
Wang, Hao, 346, 382, 383, 384.
Waring, Edward, 77.
Warren, Don W., 359.
Watson, Rev. Henry William, 382.
Wegner, Peter, 303.
Weierstrass, Karl, theorem, 381.
Weighted path length, 401–405.
Weizenbaum, Joseph, 413, 458, 459, 460.
Welch, John Tunstall, Jr., 369.
Well-ordering, 20–21, 332.
Wheeler, David John, 226, 227, 456.
Whinihan, Michael J., 85.
Whirlwind I, 226.
Wilkes, Maurice Vincent, 226, 456.

Wilson, Sir John, theorem, 49, 50.
Windsor, House of, 308.
Wire length, minimum, 370–371.
Wirth, Niklaus, 457.
Wolman, Eric, 452.
Wolontis, Vidar Michael, 226.
Woodger, Michael, xiii.
Woods, M. L., 226.
Woodward, Philip Mayne, 459.
Word: Addressable unit of computer memory, 122.
Word size, for MIX: The number of different values that might be stored in five bytes.
Worst-fit method of storage allocation, 452.
Wrench, John William, Jr., xiii, 615.
Wright, Edward Maitland, 490, 515.
Wright, Jesse B., 359.
Writing: Doing output, 211.
Writing large programs, 187–189.

X-register of MIX, 122.
XOR (exclusive or), 454.

Yngve, Victor Huse, 460.
Yo-yo list, 236.
Youden, William Wallace, xiii.
Young, Rosalind Cecily Hildegard, 75

Zeta function, 42, 74–75.
Zimmerman, Seth, 406.

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer. For additional definitions of computer terminology, see the *IFIP-ICC Vocabulary of Information Processing* (Amsterdam: North Holland Publishing Co., 1966).