THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS

Alfred V. Aho
Bell Laboratories

John E. Hopcroft
Cornell University

Jeffrey D. Ullman
Princeton University

Addison-Wesley Publishing Company
Reading, Massachusetts · Menlo Park, California
London · Amsterdam · Don Mills, Ontario · Sydney

PREFACE

The study of algorithms is at the very heart of computer science. In recent years a number of significant advances in the field of algorithms have been made. These advances have ranged from the development of faster algorithms, such as the fast Fourier transform, to the startling discovery of certain natural problems for which all algorithms are inefficient. These results have kindled considerable interest in the study of algorithms, and the area of algorithm design and analysis has blossomed into a field of intense interest. The intent of this book is to bring together the fundamental results in this area, so the unifying principles and underlying concepts of algorithm design may more easily be taught.

THE SCOPE OF THE BOOK

To analyze the performance of an algorithm some model of a computer is necessary. Our book begins by formulating several computer models which are simple enough to establish analytical results but which at the same time accurately reflect the salient features of real machines. These models include the random access register machine, the random access stored program machine, and some specialized variants of these. The Turing machine is introduced in order to prove the exponential lower bounds on efficiency in Chapters 10 and 11. Since the trend in program design is away from machine language, a high-level language called Pidgin ALGOL is introduced as the main vehicle for describing algorithms. The complexity of a Pidgin ALGOL program is related to the machine models.

The second chapter introduces basic data structures and programming techniques often used in efficient algorithms. It covers the use of lists, pushdown stores, queues, trees, and graphs. Detailed explanations of recursion, divide-and-conquer, and dynamic programming are given, along with examples of their use.

Chapters 3 to 9 provide a sampling of the diverse areas to which the fundamental techniques of Chapter 2 can be applied. Our emphasis in these chapters is on developing algorithms that are asymptotically the most efficient known. Because of this emphasis, some of the algorithms presented are suitable only for inputs whose size is much larger than what is currently encountered in practice. This is particularly true of some of the matrix multiplication algorithms in Chapter 6, the Schönhage-Strassen integer-multiplication algorithm of Chapter 7, and some of the polynomial and integer algorithms of Chapter 8.

On the other hand, most of the sorting algorithms of Chapter 3, the searching algorithms of Chapter 4, the graph algorithms of Chapter 5, the fast Fourier transform of Chapter 7, and the string-matching algorithms of Chapter 9 are widely used, since the sizes of inputs for which these algorithms are efficient are sufficiently small to be encountered in many practical situations.

Chapters 10 through 12 discuss lower bounds on computational complexity. The inherent computational difficulty of a problem is of universal interest, both to program design and to an understanding of the nature of computation. In Chapter 10 an important class of problems, the NP-complete problems, is studied. All problems in this class are equivalent in computational difficulty, in that if one problem in the class has an efficient (polynomial time-bounded) solution, then all problems in the class have efficient solutions. Since this class of problems contains many practically important and well-studied problems, such as the integer-programming problem and the traveling salesman problem, there is good reason to suspect that no problem in this class can be solved efficiently. Thus, if a program designer knows that the problem for which he is trying to find an efficient algorithm is in this class, then he may very well be content to try heuristic approaches to the problem. In spite of the overwhelming empirical evidence to the contrary, it is still an open question whether NP-complete problems admit of efficient solutions.

In Chapter 11 certain problems are defined for which we can actually prove that no efficient algorithms exist. The material in Chapters 10 and 11 draws heavily on the concept of Turing machines introduced in Sections 1.6 and 1.7.

In the final chapter of the book we relate concepts of computational difficulty to notions of linear independence in vector spaces. The material in this chapter provides techniques for proving lower bounds for much simpler problems than those considered in Chapters 10 and 11.

THE USE OF THE BOOK

This book is intended as a first course in the design and analysis of algorithms. The emphasis is on ideas and ease of understanding rather than implementation details or programming tricks. Informal, intuitive explanations are often used in place of long tedious proofs. The book is self-contained and assumes no specific background in mathematics or programming languages. However, a certain amount of maturity in being able to handle mathematical concepts is desirable, as is some exposure to a higher-level programming language such as FORTRAN or ALGOL. Some knowledge of linear algebra is needed for a full understanding of Chapters 6, 7, 8, and 12.

This book has been used in graduate and undergraduate courses in algorithm design. In a one-semester course most of Chapters 1–5 and 9–10 were covered, along with a smattering of topics from the remaining chapters. In introductory courses the emphasis was on material from Chapters 1–5, but Sections 1.6, 1.7, 4.13, 5.11, and Theorem 4.5 were generally not covered. The book can also be used in more advanced courses emphasizing the theory of algorithms. Chapters 6–12 could serve as the foundation for such a course.

Numerous exercises have been provided at the end of each chapter to provide an instructor with a wide range of homework problems. The exercises are graded according to difficulty. Exercises with no stars are suitable for introductory courses, singly starred exercises for more advanced courses, and doubly starred exercises for advanced graduate courses. The bibliographic notes at the end of every chapter attempt to point to a published source for each of the algorithms and results contained in the text and the exercises.

ACKNOWLEDGMENTS

The material in this book has been derived from class notes for courses taught by the authors at Cornell, Princeton, and Stevens Institute of Technology. The authors would like to thank the many people who have critically read various portions of the manuscript and offered many helpful suggestions. In particular we would like to thank Kellogg Booth, Stan Brown, Steve Chen, Allen Cypher,

Arch Davis, Mike Fischer, Hania Gajewska, Mike Garey, Udai Gupta, Mike Harrison, Matt Hecht, Harry Hunt, Dave Johnson, Marc Kaplan, Don Johnson, Steve Johnson, Brian Kernighan, Don Knuth, Richard Ladner, Anita LaSalle, Doug McIlroy, Albert Meyer, Christos Papadimitriou, Bill Plauger, John Savage, Howard Siegel, Ken Steiglitz, Larry Stockmeyer, Tom Szymanski, and Theodore Yen.

Special thanks go to Gemma Carnevale, Pauline Cameron, Hannah Kresse, Edith Purser, and Ruth Suzuki for their cheerful and careful typing of the manuscript.

The authors are also grateful to Bell Laboratories and Cornell, Princeton, and the University of California at Berkeley for providing facilities for the preparation of the manuscript.

June 1974

A. V. A.
J. E. H.
J. D. U.

CONTENTS

1 **Models of Computation**

2 **Design of Efficient Algorithms**

3 **Sorting and Order Statistics**

4 **Data Structures for Set Manipulation Problems**

8 Integer and Polynomial Arithmetic

9 Pattern-Matching Algorithms

10 NP-Complete Problems

11 Some Provably Intractable Problems

MODELS OF COMPUTATION

CHAPTER 1

Given a problem, how do we find an efficient algorithm for its solution? Once we have found an algorithm, how can we compare this algorithm with other algorithms that solve the same problem? How should we judge the goodness of an algorithm? Questions of this nature are of interest both to programmers and to theoretically oriented computer scientists. In this book we shall examine various lines of research that attempt to answer questions such as these.

In this chapter we consider several models of a computer—the random access machine, the random access stored program machine, and the Turing machine. We compare these models on the basis of their ability to reflect the complexity of an algorithm, and derive from them several more specialized models of computation, namely, straight-line arithmetic sequences, bitwise computations, bit vector computations, and decision trees. Finally, in the last section of this chapter we introduce a language called "Pidgin ALGOL" for describing algorithms.

1.1 ALGORITHMS AND THEIR COMPLEXITY

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We would like to associate with a problem an integer, called the *size* of the problem, which is a measure of the quantity of input data. For example, the size of a matrix multiplication problem might be the largest dimension of the matrices to be multiplied. The size of a graph problem might be the number of edges.

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The limiting behavior of the complexity as size increases is called the *asymptotic time complexity*. Analogous definitions can be made for *space complexity* and *asymptotic space complexity*.

It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be solved by the algorithm. If an algorithm processes inputs of size n in time cn^2 for some constant c, then we say that the time complexity of that algorithm is $O(n^2)$, read "order n^2." More precisely, a function $g(n)$ is said to be $O(f(n))$ if there exists a constant c such that $g(n) \leq cf(n)$ for all but some finite (possibly empty) set of nonnegative values for n.

One might suspect that the tremendous increase in the speed of calculations brought about by the advent of the present generation of digital computers would decrease the importance of efficient algorithms. However, just the opposite is true. As computers become faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose we have five algorithms A_1–A_5 with the following time complexities.

Algorithm	Time complexity
A_1	n
A_2	$n \log n$[†]
A_3	n^2
A_4	n^3
A_5	2^n

The time complexity here is the number of time units required to process an input of size n. Assuming that one unit of time equals one millisecond, algorithm A_1 can process in one second an input of size 1000, whereas algorithm A_5 can process in one second an input of size at most 9. Figure 1.1 gives the sizes of problems that can be solved in one second, one minute, and one hour by each of these five algorithms.

Algorithm	Time complexity	Maximum problem size		
		1 sec	1 min	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Fig. 1.1. Limits on problem size as determined by growth rate.

Algorithm	Time complexity	Maximum problem size before speed-up	Maximum problem size after speed-up
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	Approximately $10s_2$ for large s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Fig. 1.2. Effect of tenfold speed-up.

[†] Unless otherwise stated, all logarithms in this book are to the base 2.

Suppose that the next generation of computers is ten times faster than the current generation. Figure 1.2 shows the increase in the size of the problem we can solve due to this increase in speed. Note that with algorithm A_5, a tenfold increase in speed only increases by three the size of problem that can be solved, whereas with algorithm A_3 the size more than triples.

Instead of an increase in speed, consider the effect of using a more efficient algorithm. Refer again to Fig. 1.1. Using one minute as a basis for comparison, by replacing algorithm A_4 with A_3 we can solve a problem six times larger; by replacing A_4 with A_2 we can solve a problem 125 times larger. These results are far more impressive than the twofold improvement obtained by a tenfold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant. We conclude that the asymptotic complexity of an algorithm is an important measure of the goodness of an algorithm, one that promises to become even more important with future increases in computing speed.

Despite our concentration on order-of-magnitude performance, we should realize that an algorithm with a rapid growth rate might have a smaller constant of proportionality than one with a lower growth rate. In that case, the rapidly growing algorithm might be superior for small problems, possibly even for all problems of a size that would interest us. For example, suppose the time complexities of algorithms A_1, A_2, A_3, A_4, and A_5 were really $1000n$, $100n \log n$, $10n^2$, n^3, and 2^n. Then A_5 would be best for problems of size $2 \leq n \leq 9$, A_3 would be best for $10 \leq n \leq 58$, A_2 would be best for $59 \leq n \leq 1024$, and A_1 best for problems of size greater than 1024.

Before going further with our discussion of algorithms and their complexity, we must specify a model of a computing device for executing algorithms and define what is meant by a basic step in a computation. Unfortunately, there is no one computational model which is suitable for all situations. One of the main difficulties arises from the size of computer words. For example, if one assumes that a computer word can hold an integer of arbitrary size, then an entire problem could be encoded into a single integer in one computer word. On the other hand, if a computer word is assumed to be finite, one must consider the difficulty of simply storing arbitrarily large integers, as well as other problems which one often avoids when given problems of modest size. For each problem we must select an appropriate model which will accurately reflect the actual computation time on a real computer.

In the following sections we discuss several fundamental models of computing devices, the more important models being the random access machine, the random access stored program machine, and the Turing machine. These three models are equivalent in computational power but not in speed.

Perhaps the most important motivation for formal models of computation is the desire to discover the inherent computational difficulty of various problems. We would like to prove lower bounds on computation time. In order to show that there is no algorithm to perform a given task in less than a certain amount of time, we need a precise and often highly stylized definition of what constitutes an algorithm. Turing machines (Section 1.6) are an example of such a definition.

In describing and communicating algorithms we would like a notation more natural and easy to understand than a program for a random access machine, random access stored program machine, or Turing machine. For this reason we shall also introduce a high-level language called Pidgin ALGOL. This is the language we shall use throughout the book to describe algorithms. However, to understand the computational complexity of an algorithm described in Pidgin ALGOL we must relate Pidgin ALGOL to the more formal models. This we do in the last section of this chapter.

1.2 RANDOM ACCESS MACHINES

A random access machine (RAM) models a one-accumulator computer in which instructions are not permitted to modify themselves.

A RAM consists of a read-only input tape, a write-only output tape, a program, and a memory (Fig. 1.3). The input tape is a sequence of squares, each of which holds an integer (possibly negative). Whenever a symbol is read from the input tape, the tape head moves one square to the right. The output is a write-only tape ruled into squares which are initially all blank. When a write instruction is executed, an integer is printed in the square of the

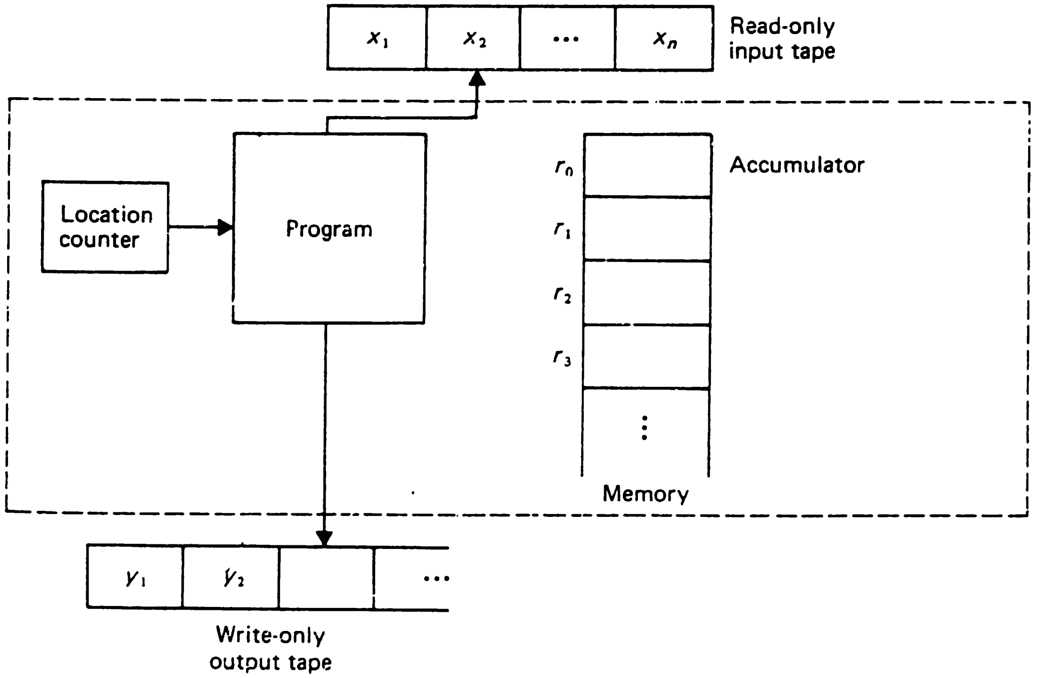


Fig. 1.3 A random access machine.

output tape that is currently under the output tape head, and the tape head is moved one square to the right. Once an output symbol has been written, it cannot be changed.

The memory consists of a sequence of registers, r_0, r_1, . . . , r_i, . . . , each of which is capable of holding an integer of arbitrary size. We place no upper bound on the number of registers that can be used. This abstraction is valid in cases where:

1. the size of the problem is small enough to fit in the main memory of a computer, and
2. the integers used in the computation are small enough to fit in one computer word.

The program for a RAM is not stored in the memory. Thus we are assuming that the program does not modify itself. The program is merely a sequence of (optionally) labeled instructions. The exact nature of the instructions used in the program is not too important, as long as the instructions resemble those usually found in real computers. We assume there are arithmetic instructions, input-output instructions, indirect addressing (for indexing arrays, e.g.) and branching instructions. All computation takes place in the first register r_0, called the *accumulator*, which like every other memory register can hold an arbitrary integer. A sample set of instructions for the RAM is shown in Fig. 1.4. Each instruction consists of two parts—an *operation code* and an *address*.

In principle, we could augment our set with any other instructions found in real computers, such as logical or character operations, without altering the order-of-magnitude complexity of problems. The reader may imagine the instruction set to be so augmented if it suits him.

Operation code	Address
1. LOAD	operand
2. STORE	operand
3. ADD	operand
4. SUB	operand
5. MULT	operand
6. DIV	operand
7. READ	operand
8. WRITE	operand
9. JUMP	label
10. JGTZ	label
11. JZERO	label
12. HALT	

Fig. 1.4. Table of RAM instructions.

An operand can be one of the following:

1. $=i$, indicating the integer i itself.
2. A nonnegative integer i, indicating the contents of register i.
3. $*i$, indicating indirect addressing. That is, the operand is the contents of register j, where j is the integer found in register i. If $j < 0$, then the machine halts.

These instructions should be quite familiar to anyone who has programmed in assembly language. We can define the meaning of a program P with the help of two quantities, a mapping c from nonnegative integers to integers and a "location counter" which determines the next instruction to execute. The function c is a *memory map;* $c(i)$ is the integer stored in register i (the *contents* of register i).

Initially, $c(i) = 0$ for all $i \geq 0$, the location counter is set to the first instruction in P, and the output tape is all blank. After execution of the kth instruction in P, the location counter is automatically set to $k + 1$ (i.e., the next instruction), unless the kth instruction is JUMP, HALT, JGTZ, or JZERO.

To specify the meaning of an instruction we define $v(a)$, *the value of operand a,* as follows:

$$
\begin{aligned}
v(=i) &= i,\\
v(i) &= c(i),\\
v(*i) &= c(c(i)).
\end{aligned}
$$

The table of Fig. 1.5 defines the meaning of each instruction in Fig. 1.4. Instructions not defined, such as STORE $=i$, may be considered equivalent to HALT. Likewise, division by zero halts the machine.

During the execution of each of the first eight instructions the location counter is incremented by one. Thus instructions in the program are executed in sequential order until a JUMP or HALT instruction is encountered, a JGTZ instruction is encountered with the contents of the accumulator greater than zero, or a JZERO instruction is encountered with the contents of the accumulator equal to zero.

In general, a RAM program defines a mapping from input tapes to output tapes. Since the program may not halt on all input tapes, the mapping is a partial mapping (that is, the mapping may be undefined for certain inputs). The mapping can be interpreted in a variety of ways. Two important interpretations are as a function or as a language.

Suppose a program P always reads n integers from the input tape and writes at most one integer on the output tape. If, when $x_1, x_2, \ldots, x_n$ are the integers in the first n squares of the input tape, P writes y on the first square of the output tape and subsequently halts, then we say that P computes the function $f(x_1, x_2, \ldots, x_n) = y$. It is easily shown that a RAM, like any other

Instruction	Meaning
1. LOAD a	$c(0) \leftarrow v(a)$
2. STORE i	$c(i) \leftarrow c(0)$
STORE $*i$	$c(c(i)) \leftarrow c(0)$
3. ADD a	$c(0) \leftarrow c(0) + v(a)$
4. SUB a	$c(0) \leftarrow c(0) - v(a)$
5. MULT a	$c(0) \leftarrow c(0) \times v(a)$
6. DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$†
7. READ i	$c(i) \leftarrow$ current input symbol.
READ $*i$	$c(c(i)) \leftarrow$ current input symbol. The input tape head moves one square right in either case.
8. WRITE a	$v(a)$ is printed on the square of the output tape currently under the output tape head. Then the tape head is moved one square right.
9. JUMP b	The location counter is set to the instruction labeled b.
10. JGTZ b	The location counter is set to the instruction labeled b if $c(0) > 0$; otherwise, the location counter is set to the next instruction.
11. JZERO b	The location counter is set to the instruction labeled b if $c(0) = 0$; otherwise, the location counter is set to the next instruction.
12. HALT	Execution ceases.

† Throughout this book, $\lceil x \rceil$ (*ceiling* of x) denotes the least integer equal to or greater than x, and $\lfloor x \rfloor$ (*floor*, or *integer part* of x) denotes the greatest integer equal to or less than x.

Fig. 1.5. Meaning of RAM instructions. The operand a is either $=i$, i, or $*i$.

reasonable model of a computer, can compute exactly the *partial recursive functions*. That is, given any partial recursive function f we can define a RAM program that computes f, and given any RAM program we can define an equivalent partial recursive function. (See Davis [1958] or Rogers [1967] for a discussion of recursive functions.)

Another way to interpret a RAM program is as an acceptor of a language. An *alphabet* is a finite set of symbols, and a *language* is a set of strings over some alphabet. The symbols of an alphabet can be represented by the integers $1, 2, \ldots, k$ for some k. A RAM can accept a language in the following manner. We place an input string $s = a_1 a_2 \cdots a_n$ on the input tape, placing the symbol a_1 in the first square, the symbol a_2 in the second square, and so on. We place 0, a symbol we shall use as an endmarker, in the $(n + 1)$st square to mark the end of the input string.

The input string s is *accepted* by a RAM program P if P reads all of s and the endmarker, writes a 1 in the first square of the output tape, and halts. The *language accepted by* P is the set of accepted input strings. For input strings not in the language accepted by P, P may print a symbol other than 1 on the output tape and halt, or P may not even halt. It is easily shown that a language is accepted by a RAM program if and only if it is *recursively enumerable*. A language is accepted by a RAM that halts on all inputs if and only if it is a *recursive* language (see Hopcroft and Ullman [1969] for a discussion of recursive and recursively enumerable languages).

Let us consider two examples of RAM programs. The first defines a function: the second accepts a language.

Example 1.1. Consider the function $f(n)$ given by

$$f(n) = \begin{cases} n^n & \text{for all integers } n \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

A Pidgin ALGOL program which computes $f(n)$ by multiplying n by itself $n - 1$ times is illustrated in Fig. 1.6.† A corresponding RAM program is given in Fig. 1.7. The variables $r1$, $r2$, and $r3$ are stored in registers 1, 2, and 3 respectively. Certain obvious optimizations have not been made, so the correspondence between Figs. 1.6 and 1.7 will be transparent. □

```
begin
    read $r1$;
    if $r1 \leq 0$ then write 0
    else
        begin
            $r2 \leftarrow r1$;
            $r3 \leftarrow r1 - 1$;
            while $r3 > 0$ do
                begin
                    $r2 \leftarrow r2 * r1$;
                    $r3 \leftarrow r3 - 1$
                end;
            write $r2$
        end
end
```

Fig. 1.6. Pidgin ALGOL program for n^n.

† See Section 1.8 for a description of Pidgin ALGOL.

```
                                                  Corresponding
                 RAM program                 Pidgin ALGOL statements
-----------------------------------------------------------------------
             READ    1                       read r1
             LOAD    1        ⎫
             JGTZ    pos      ⎬              if r1 ≤ 0 then write 0
             WRITE   =0       ⎭
             JUMP    endif
pos:         LOAD    1        ⎫
             STORE   2        ⎭              r2 ← r1
             LOAD    1        ⎫
             SUB     =1       ⎬              r3 ← r1 − 1
             STORE   3        ⎭
while:       LOAD    3        ⎫
             JGTZ    continue ⎬              while r3 > 0 do
             JUMP    endwhile ⎭
continue:    LOAD    2        ⎫
             MULT    1        ⎬              r2 ← r2 * r1
             STORE   2        ⎭
             LOAD    3        ⎫
             SUB     =1       ⎬              r3 ← r3 − 1
             STORE   3        ⎭
             JUMP    while
endwhile:    WRITE   2                       write r2
endif:       HALT
-----------------------------------------------------------------------
```

Fig. 1.7. RAM program for n^n.

Example 1.2. Consider a RAM program that accepts the language over the input alphabet $\{1, 2\}$ consisting of all strings with the same number of 1's and 2's. The program reads each input symbol into register 1 and in register 2 keeps track of the difference d between the number of 1's and 2's seen so far. When the endmarker 0 is encountered, the program checks that the difference is zero, and if so prints 1 and halts. We assume that 0, 1, and 2 are the only possible input symbols.

The program of Fig. 1.8 contains the essential details of the algorithm. An equivalent RAM program is given in Fig. 1.9; x is stored in register 1 and d in register 2. □

```
begin
    $d \leftarrow 0$;
    read $x$;
    while $x \neq 0$ do
        begin
            if $x \neq 1$ then $d \leftarrow d - 1$ else $d \leftarrow d + 1$;
            read $x$
        end;
    if $d = 0$ then write 1
end
```

Fig. 1.8. Recognizing strings with equal numbers of 1's and 2's.

RAM program			Corresponding Pidgin ALGOL statements
	LOAD	=0	$d \leftarrow 0$
	STORE	2	
	READ	1	**read** x
while:	LOAD	1	**while** $x \neq 0$ **do**
	JZERO	endwhile	
	LOAD	1	**if** $x \neq 1$
	SUB	=1	
	JZERO	one	
	LOAD	2	**then** $d \leftarrow d - 1$
	SUB	=1	
	STORE	2	
	JUMP	endif	
one:	LOAD	2	**else** $d \leftarrow d + 1$
	ADD	=1	
	STORE	2	
endif:	READ	1	**read** x
	JUMP	while	
endwhile:	LOAD	2	**if** $d = 0$ **then write** 1
	JZERO	output	
	HALT		
output:	WRITE	=1	
	HALT		

Fig. 1.9. RAM program corresponding to algorithm in Fig. 1.8.

1.3 COMPUTATIONAL COMPLEXITY OF RAM PROGRAMS

Two important measures of an algorithm are its time and space complexity, measured as functions of the size of the input. If for a given size the complexity is taken as the maximum complexity over all inputs of that size, then the complexity is called the *worst-case complexity*. If the complexity is taken as the "average" complexity over all inputs of given size, then the complexity is called the *expected complexity*. The expected complexity of an algorithm is usually more difficult to ascertain than the worst-case complexity. One must make some assumption about the distribution of inputs, and realistic assumptions are often not mathematically tractable. We shall emphasize the worst case, since it is more tractable and has a universal applicability. However, it should be borne in mind that the algorithm with the best worst-case complexity does not necessarily have the best expected complexity.

The *worst-case time complexity* (or just *time complexity*) of a RAM program is the function $f(n)$ which is the maximum, over all inputs of size n, of the sum of the "time" taken by each instruction executed. The *expected time complexity* is the average, over all inputs of size n, of the same sum. The same terms are defined for space if we substitute "'space' used by each register referenced" for "'time' taken by each instruction executed."

To specify the time and space complexity exactly, we must specify the time required to execute each RAM instruction and the space used by each register. We shall consider two such cost criteria for RAM programs. Under the *uniform cost criterion* each RAM instruction requires one unit of time and each register requires one unit of space. Unless otherwise mentioned, the complexity of a RAM program will be measured according to the uniform cost criterion.

A second, sometimes more realistic definition takes into account the limited size of a real memory word and is called the *logarithmic cost criterion*. Let $l(i)$ be the following *logarithmic function* on the integers:

$$l(i) = \begin{cases} \lfloor \log |i| \rfloor + 1, & i \neq 0 \\ 1, & i = 0 \end{cases}$$

The table of Fig. 1.10 summarizes the logarithmic cost $t(a)$ for the three possible forms of an operand a. Figure 1.11 summarizes the time required by each instruction.

Operand a	Cost $t(a)$
$=i$	$l(i)$
i	$l(i) + l(c(i))$
$*i$	$l(i) + l(c(i)) + l(c(c(i)))$

Fig. 1.10. Logarithmic cost of an operand.

Instruction	Cost
1. LOAD a	$t(a)$
2. STORE i	$l(c(0)) + l(i)$
STORE $*i$	$l(c(0)) + l(i) + l(c(i))$
3. ADD a	$l(c(0)) + t(a)$
4. SUB a	$l(c(0)) + t(a)$
5. MULT a	$l(c(0)) +.t(a)$
6. DIV a	$l(c(0)) + t(a)$
7. READ i	$l(\text{input}) + l(i)$
READ $*i$	$l(\text{input}) + l(i) + l(c(i))$
8. WRITE a	$t(a)$
9. JUMP b	1
10. JGTZ b	$l(c(0))$
11. JZERO b	$l(c(0))$
12. HALT	1

Fig. 1.11. Logarithmic cost of RAM instructions, where $t(a)$ is the cost of operand a, and b denotes a label.

The cost takes into account the fact that $\lfloor \log n \rfloor + 1$ bits are required to represent the integer n in a register. Registers, we recall, can hold arbitrarily large integers.

The logarithmic cost criterion is based on the crude assumption that the cost of performing an instruction is proportional to the length of the operands of the instructions. For example, consider the cost of instruction ADD $*i$. First we must determine the cost of decoding the operand represented by the address. To examine the integer i requires time $l(i)$. Then to read $c(i)$, the contents of register i, and locate register $c(i)$ requires time $l(c(i))$. Finally, reading the contents of register $c(i)$ costs $l(c(c(i)))$. Since the instruction ADD $*i$ adds the integer $c(c(i))$ to $c(0)$, the integer in the accumulator, we see that $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$ is a reasonable cost to assign to the instruction ADD $*i$.

We define the *logarithmic space complexity* of a RAM program to be the sum over all registers, including the accumulator, of $l(x_i)$, where x_i is the integer of largest magnitude stored in register i at any time during the computation.

It goes without saying that a given program may have radically different time complexities depending on whether the uniform or logarithmic cost is used. If it is reasonable to assume that each number encountered in a problem can be stored in one computer word, then the uniform cost function is appropriate. Otherwise the logarithmic cost might be more appropriate for a realistic complexity analysis.

Let us compute the time and space complexity of the RAM program in Example 1.1, which evaluates n^n. The time complexity of the program is dominated by the loop with the MULT instruction. The ith time the MULT instruction is executed the accumulator contains n^i and register 2 contains n. A total of $n - 1$ MULT instructions are executed. Under the uniform cost criterion, each MULT instruction costs one unit of time, and thus $O(n)$ time is spent in executing all the MULT instructions. Under the logarithmic cost criterion, the cost of executing the ith MULT instruction is $l(n^i) + l(n) \simeq (i + 1) \log n$, and thus the total cost of the MULT instructions is

$$\sum_{i=1}^{n-1} (i + 1) \log n,$$

which is $O(n^2 \log n)$.

The space complexity is determined by the integers stored in registers 0 to 3. Under the uniform cost the space complexity is simply $O(1)$. Under the logarithmic cost, the space complexity is $O(n \log n)$, since the largest integer stored in any of these registers is n^n, and $l(n^n) \simeq n \log n$. Thus we have the following complexities for the program of Example 1.1.

	Uniform cost	Logarithmic cost
Time complexity	$O(n)$	$O(n^2 \log n)$
Space complexity	$O(1)$	$O(n \log n)$

For this program the uniform cost is realistic only if a single computer word can store an integer as large as n^n. If n^n is larger than what can be stored in one computer word, then even the logarithmic time complexity is somewhat unrealistic, since it assumes that two integers i and j can be multiplied together in time $O(l(i) + l(j))$, which is not known to be possible.

For the RAM program in Example 1.2, assuming n is the length of the input string, the time and space complexities are:

	Uniform cost	Logarithmic cost
Time complexity	$O(n)$	$O(n \log n)$
Space complexity	$O(1)$	$O(\log n)$

For this program, if n is larger than what can be stored in one computer word, the logarithmic cost is quite realistic.

1.4 A STORED PROGRAM MODEL

Since a RAM program is not stored in the memory of the RAM, the program cannot modify itself. We now consider another model of a computer, called a *random access stored program* machine (RASP), which is similar to a RAM with the exception that the program is in memory and can modify itself.

The instruction set for a RASP is identical to that for a RAM, except indirect addressing is not permitted since it is not needed. We shall see that a RASP can simulate indirect addressing by the modification of instructions during program execution.

The overall structure of a RASP is also similar to that of a RAM, but the program of a RASP is assumed to be in the registers of the memory. Each RASP instruction occupies two consecutive memory registers. The first register holds an encoding of the operation code; the second register holds the address. If the address is of the form $=i$, then the first register will also encode the fact that the operand is a literal, and the second register will contain i. Integers are used to encode the instructions. Figure 1.12 gives one possible encoding. For example, the instruction LOAD =32 would be stored with 2 in one register and 32 in the following register.

As for a RAM, the state of a RASP can be represented by:

1. the memory map c, where $c(i)$, $i \geq 0$, is the contents of register i, and
2. the location counter, which indicates the first of the two consecutive memory registers from which the current instruction is to be taken.

Initially, the location counter is set at some specified register. The initial contents of the memory registers are usually not all 0, since the program has been loaded into the memory at the start. However, we insist that all but a finite number of the registers contain 0 at the start and that the accumulator contain 0. After each instruction is executed, the location counter is increased

Instruction	Encoding	Instruction	Encoding
LOAD i	1	DIV i	10
LOAD $=i$	2	DIV $=i$	11
STORE i	3	READ i	12
ADD i	4	WRITE i	13
ADD $=i$	5	WRITE $=i$	14
SUB i	6	JUMP i	15
SUB $=i$	7	JGTZ i	16
MULT i	8	JZERO i	17
MULT $=i$	9	HALT	18

Fig. 1.12. Code for RASP instructions.

by 2, except in the case of JUMP i, JGTZ i (when the accumulator is positive), or JZERO i (when the accumulator contains 0), in which case the location counter is set to i. The effect of each instruction is the same as the corresponding RAM instruction.

The time complexity of a RASP program can be defined in much the same manner as that of a RAM program. We can use either the uniform cost criterion or the logarithmic cost. In the latter case, however, we must charge not only for evaluating an operand, but also for accessing the instruction itself. The accessing cost is l(LC) where LC denotes the contents of the location counter. For example, the cost of executing the instruction ADD $=i$, stored in registers j and $j+1$, is $l(j) + l(c(0)) + l(i)$.† The cost of the instruction ADD i, stored in registers j and $j+1$, is $l(j) + l(c(0)) + l(i) + l(c(i))$.

It is interesting to ask what is the difference in complexity between a RAM program and the corresponding RASP program. The answer is not surprising. Any input–output mapping that can be performed in time $T(n)$ by one model can be performed in time $kT(n)$ by the other, for some constant k, whether cost is taken to be uniform or logarithmic. Likewise, the space used by either model differs by only a constant factor under these two cost measures.

These relationships are stated formally in the following two theorems. Both theorems are proved by exhibiting algorithms whereby a RAM can simulate a RASP, and vice versa.

Theorem 1.1. If costs of instructions are either uniform or logarithmic, for every RAM program of time complexity $T(n)$ there is a constant k such that there is an equivalent RASP program of time complexity $kT(n)$.

Proof. We show how to simulate a RAM program P by a RASP program. Register 1 of the RASP will be used to store temporarily the contents of the RAM accumulator. From P we shall construct a RASP program P_s which will occupy the next $r-1$ registers of the RASP. The constant r is determined by the RAM program P. The contents of RAM register i, $i \geq 1$, will be stored in RASP register $r+i$, so all memory references in the RASP program are r more than the corresponding references in the RAM program.

Each RAM instruction in P not involving indirect addressing is directly encoded into the identical RASP instruction (with memory references appropriately incremented). Each RAM instruction in P involving indirect addressing is mapped into a sequence of six RASP instructions that simulates the indirect addressing via instruction modification.

† We could also charge for reading register $j+1$, but this cost cannot differ greatly from $l(j)$. Throughout this chapter we are not concerned with constant factors, but rather with the growth rate of functions. Thus $l(j) + l(j+1)$ is "approximately" $l(j)$, that is, within a factor of 3 at most.

Register	Contents	Meaning	
100	3	STORE 1	
101	1		
102	1	LOAD $r+i$	
103	$r+i$		
104	5	ADD $=r$	
105	r		
106	3	STORE 111	
107	111		
108	1	LOAD 1	
109	1		
110	6	SUB b	where b is the contents of RAM register i
111	−		

Fig. 1.13. Simulation of SUB $*i$ by RASP.

An example should suffice to illustrate the simulation of indirect addressing. To simulate the RAM instruction SUB $*i$, where i is a positive integer, we shall compile a sequence of RASP instructions that

1. temporarily stores the contents of the accumulator in register 1,
2. loads the contents of register $r + i$ into the accumulator (register $r + i$ of the RASP corresponds to register i of the RAM),
3. adds r to the accumulator,
4. stores the number calculated by step 3 into the address field of a SUB instruction,
5. restores the accumulator from the temporary register 1, and finally
6. uses the SUB instruction created in step 4 to perform the subtraction.

For example, using the encoding for RASP instructions given in Fig. 1.12 and assuming the sequence of RASP instructions begins in register 100, we would simulate SUB $*i$ with the sequence shown in Fig. 1.13. The offset r can be determined once the number of instructions in the RASP program P_s is known.

We observe that each RAM instruction requires at most six RASP instructions, so under the uniform cost criterion the time complexity of the RASP program is at most $6T(n)$. (Note that this measure is independent of the way in which the "size" of the input is determined.)

Under the logarithmic cost criterion, we observe that each RAM instruction I in P is simulated by a sequence S of either one or six RASP instructions in P_s. We can show that there is a constant k dependent on P such that the cost of the instructions in S is not greater than k times the cost of instruction I.

RASP register	Instruction	Cost
j	STORE 1	$l(j) + l(1) + l(c(0))$
$j+2$	LOAD $r+i$	$l(j+2) + l(r+i) + l(c(i))$
$j+4$	ADD $=r$	$l(j+4) + l(c(i)) + l(r)$
$j+6$	STORE $j+11$	$l(j+6) + l(j+11) + l(c(i)+r)$
$j+8$	LOAD 1	$l(j+8) + l(1) + l(c(0))$
$j+10$	SUB —	$l(j+10) + l(c(i)+r) + l(c(0))$ $+ l(c(c(i)))$

Fig. 1.14. Cost of RASP instructions.

For example, the RAM instruction SUB $*i$ has cost

$$M = l(c(0)) + l(i) + l(c(i)) + l(c(c(i))).$$

The sequence S that simulates this RAM instruction is shown in Fig. 1.14. $c(0)$, $c(i)$, and $c(c(i))$ in Fig. 1.14 refer to the contents of RAM registers. Since P_s occupies the registers 2 through r of the RASP, we have $j \leq r - 11$. Also, $l(x+y) \leq l(x) + l(y)$, so the cost of S is certainly less than

$$2l(1) + 4M + 11l(r) < (6 + 11l(r))M.$$

Thus we can conclude that there is a constant $k = 6 + 11l(r)$ such that if P is of time complexity $T(n)$, then P_s is of time complexity at most $kT(n)$. □

Theorem 1.2. If costs of instructions are either uniform or logarithmic, for every RASP program of time complexity $T(n)$ there is a constant k such that there is an equivalent RAM program of time complexity at most $kT(n)$.

Proof. The RAM program we shall construct to simulate the RASP will use indirect addressing to decode and simulate RASP instructions stored in the memory of the RAM. Certain registers of the RAM will have special purposes:

register 1 — used for indirect addressing,
register 2 — the RASP's location counter,
register 3 — storage for the RASP's accumulator.

Register i of the RASP will be stored in register $i + 3$ of the RAM for $i \geq 1$.

The RAM begins with the finite-length RASP program loaded in its memory starting at register 4. Register 2, the location counter, holds 4; registers 1 and 3 hold 0. The RAM program consists of a simulation loop which begins by reading an instruction of the RASP (with a LOAD $*2$ RAM instruction), decoding it and branching to one of 18 sets of instructions, each designed to handle one type of RASP instruction. On an invalid operation code the RAM, like the RASP, will halt.

The decoding and branching operations are straightforward; Example 1.2 can serve as a model (although the symbol decoded there was read from the

LOAD 2 ADD =1 STORE 2	Increment the location counter by 1, so it points to the register holding the operand *i* of the SUB *i* instruction.
LOAD *2 ADD =3 STORE 1	Bring *i* to the accumulator, add 3, and store the result in register 1.
LOAD 3 SUB *1 STORE 3	Fetch the contents of the RASP accumulator from register 3. Subtract the contents of register $i + 3$ and place the result back in register 3.
LOAD 2 ADD =1 STORE 2	Increment the location counter by 1 again so it now points to the next RASP instruction.
JUMP *a*	Return to the beginning of the simulation loop (here named "*a*").

Fig. 1.15. Simulation of SUB *i* by RAM.

input, and here it is read from memory). We shall give an example of the RAM instructions to simulate RASP instruction 6, i.e., SUB *i*. This program, shown in Fig. 1.15, is invoked when $c(c(2)) = 6$, that is, the location counter points to a register holding 6, the code for SUB.

We omit further details of the RAM program construction. It is left as an exercise to show that under the uniform or logarithmic cost criterion the time complexity of the RAM program is at most a constant times that of the RASP. □

It follows from Theorems 1.1 and 1.2 that as far as time complexity (and also space complexity—which is left as an exercise) is concerned, the RAM and RASP models are equivalent within a constant factor, i.e., their order-of-magnitude complexities are the same for the same algorithm. Of the two models, in this text we shall usually use the RAM model, since it is somewhat simpler.

1.5 ABSTRACTIONS OF THE RAM

The RAM and RASP are more complicated models of computation than are needed for many situations. Thus we define a number of models which abstract certain features of the RAM and ignore others. The justification for such models is that the ignored instructions represent at most a constant fraction of the cost of any efficient algorithm for problems to which the model is applied.

I. Straight-Line Programs

The first model we consider is the straight-line program. For many problems it is reasonable to restrict attention to the class of RAM programs in which branching instructions are used solely to cause a sequence of instructions to be repeated a number of times proportional to n, the size of the input. In this case we may "unroll" the program for each size n by duplicating the instructions to be repeated an appropriate number of times. This results in a sequence of straight-line (loop-free) programs of presumably increasing length, one for each n.

Example 1.3. Consider the multiplication of two $n \times n$ matrices of integers. It is often reasonable to expect that in a RAM program, the number of times a loop is executed is independent of the actual entries of the matrices. We may therefore find it a useful simplification to assume that the only loops permitted are those whose test instructions involve only n, the size of the problem. For example, the obvious matrix multiplication algorithm has loops which must be executed exactly n times, requiring branch instructions that compare an index to n. □

Unrolling a program into a straight line allows us to dispense with branching instructions. The justification is that in many problems no more than a constant fraction of the cost of a RAM program is devoted to branch instructions controlling loops. Likewise, it may often be assumed that input statements form only a constant fraction of the cost of the program, and we eliminate them by assuming the finite set of inputs needed for a particular n to be in memory at the start of the program. The effect of indirect addressing can be determined when n is fixed, provided the registers used for indirection contain values depending only on n, not on the values of the input variables. We therefore assume that our straight-line programs have no indirect addressing.

In addition, since each straight-line program can reference only a finite number of memory registers, it is convenient to name the registers used by the program. Thus registers are referred to by *symbolic addresses* (symbols or strings of letters) rather than integer numbers.

Having eliminated the need for READ, JUMP, JGTZ, and JZERO, we are left with the LOAD, STORE, WRITE, HALT, and arithmetic operations from the RAM repertoire. We don't need HALT, since the end of the program must indicate the halt. We can dispense with WRITE by designating certain symbolic addresses to be *output variables;* the output of the program is the value held by these variables upon termination.

Finally, we can combine LOAD and STORE into the arithmetic operations by replacing sequences such as

```
LOAD  a
ADD   b
STORE c
```

by $c \leftarrow a + b$. The entire repertoire of straight-line program instructions is:

$$\begin{aligned}
&x \leftarrow y + z\\
&x \leftarrow y - z\\
&z \leftarrow y * z\\
&z \leftarrow y/z\\
&x \leftarrow i
\end{aligned}$$

where x, y, and z are symbolic addresses (or *variables*) and i is a constant. It is easy to see that any sequence of LOAD, STORE, and arithmetic operations on the accumulator can be replaced by a sequence of the five instructions above.

Associated with a straight-line program are two designated sets of variables, the *inputs* and *outputs*. The function computed by the straight-line program is the set of values of the output variables (in designated order) expressed in terms of the values of its input variables.

Example 1.4. Consider evaluating the polynomial

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0.$$

The input variables are the coefficients $a_0, a_1, \ldots, a_n$ and the indeterminate x. The output variable is p. *Horner's rule* evaluates $p(x)$ as

1. $a_1 x + a_0$ for $n = 1$,
2. $(a_2 x + a_1)x + a_0$ for $n = 2$,
3. $((a_3 x + a_2)x + a_1)x + a_0$ for $n = 3$.

The straight-line programs of Fig. 1.16 correspond to these expressions. Horner's rule for arbitrary n should now be clear. For each n we have a straight-line program of $2n$ steps evaluating a general nth-degree polynomial. In Chapter 12 we show that n multiplications and n additions are necessary to evaluate an arbitrary nth-degree polynomial given the coefficients as input. Thus Horner's rule is optimal under the straight-line program model. □

$n = 1$	$n = 2$	$n = 3$
$t \leftarrow a_1 * x$	$t \leftarrow a_2 * x$	$t \leftarrow a_3 * x$
$p \leftarrow t + a_0$	$t \leftarrow t + a_1$	$t \leftarrow t + a_2$
	$t \leftarrow t * x$	$t \leftarrow t * x$
	$p \leftarrow t + a_0$	$t \leftarrow t + a_1$
		$t \leftarrow t * x$
		$p \leftarrow t + a_0$

Fig. 1.16. Straight-line programs corresponding to Horner's rule.

Under the straight-line program model of computation, the time complexity of a sequence of programs is the number of steps in the nth program, as a function of n. For example, Horner's rule yields a sequence of time complexity $2n$. Note that measuring time complexity is the same as measuring the number of arithmetic operations. The space complexity of a sequence of programs is the number of variables mentioned, again as a function of n. The programs of Example 1.4 have space complexity $n + 4$.

> **Definition.** When the straight-line program model is intended, we say a problem is of time or space complexity $O_A(f(n))$ if there is a sequence of programs of time or space complexity at most $cf(n)$ for some constant c. (The notation $O_A(f(n))$ stands for "on the order of $f(n)$ steps using the straight-line program model." The subscript A stands for "arithmetic," which is the chief characteristic of straight-line code.) Thus polynomial evaluation is of time complexity $O_A(n)$ and space complexity $O_A(n)$, as well.

II. Bitwise Computations

The straight-line program model is clearly based on the uniform cost function. As we have mentioned, this cost is appropriate under the assumption that all computed quantities are of "reasonable" size. There is a simple modification of the straight-line program model which reflects the logarithmic cost function. This model, which we call *bitwise computation,* is essentially the same as straight-line code, except that:

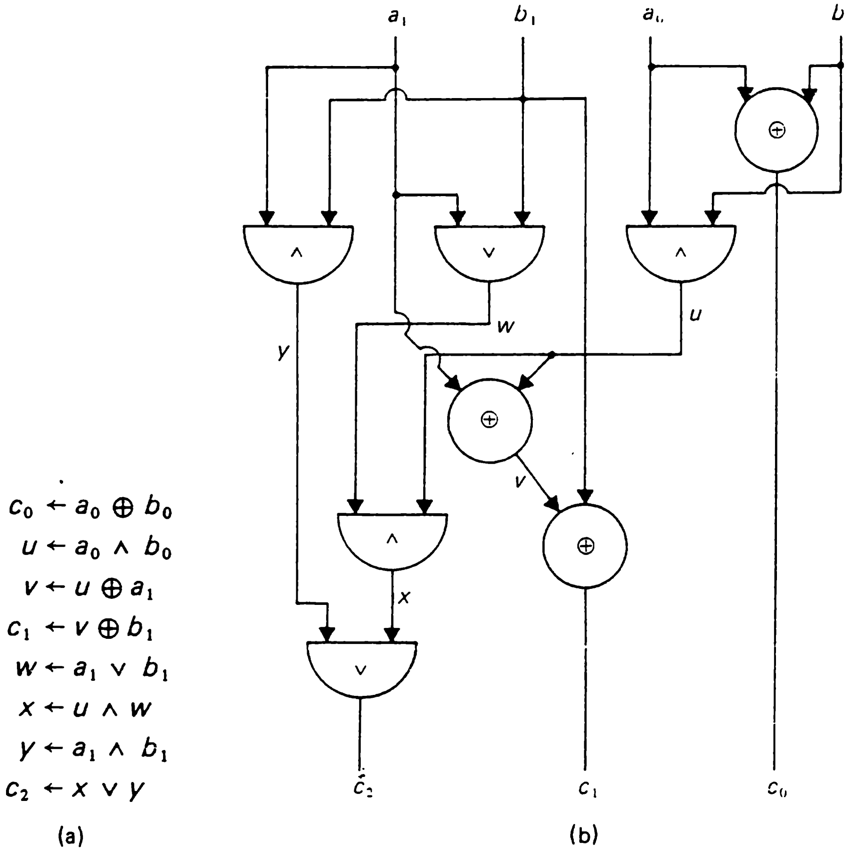
1. All variables are assumed to have the values 0 or 1, i.e., they are bits.
2. The operations used are logical rather than arithmetic.† We use $\wedge$ for **and**, $\vee$ for **or**, $\oplus$ for **exclusive or**, and $\neg$ for **not**.

Under the bitwise model, arithmetic operations on integers i and j take at least $l(i) + l(j)$ steps, reflecting the logarithmic cost of operands. In fact, multiplication and division by the best algorithms known require more than $l(i) + l(j)$ steps to multiply or divide i by j.

We use O_B to indicate order of magnitude under the bitwise computation model. The bitwise model is useful for talking about basic operations, such as the arithmetic ones, which are primitive in other models. For example, under the straight-line program model, multiplication of two n-bit integers can be done in $O_A(1)$ step whereas under the bitwise model the best result known is $O_B(n \log n \operatorname{loglog} n)$ steps.

Another application of the bitwise model is to logic circuits. Straight-line programs with binary inputs and operations have a one-to-one correspondence with combinational logic circuits computing a set of Boolean functions. The number of steps in the program is the number of logic elements in the circuit.

† Thus the instruction set of our RAM must include these operations.



```
$c_0 \leftarrow a_0 \oplus b_0$
$u \leftarrow a_0 \wedge b_0$
$v \leftarrow u \oplus a_1$
$c_1 \leftarrow v \oplus b_1$
$w \leftarrow a_1 \vee b_1$
$x \leftarrow u \wedge w$
$y \leftarrow a_1 \wedge b_1$
$c_2 \leftarrow x \vee y$
```

Fig. 1.17 (a) Bitwise addition program; (b) equivalent logical circuit.

Example 1.5. Figure 1.17(a) contains a program to add two 2-bit numbers $[a_1a_0]$ and $[b_1b_0]$. The output variables are c_2, c_1, and c_0 such that $[a_1a_0] + [b_1b_0] = [c_2c_1c_0]$. The straight-line program in Fig. 1.17(a) computes:

$$c_0 = a_0 \oplus b_0,$$
$$c_1 = ((a_0 \wedge b_0) \oplus a_1) \oplus b_1,$$
$$c_2 = ((a_0 \wedge b_0) \wedge (a_1 \vee b_1)) \vee (a_1 \wedge b_1).$$

Figure 1.17(b) shows the corresponding logical circuit. We leave it as an exercise to show that addition of two n-bit numbers can be performed in $O_B(n)$ steps. □

III. Bit Vector Operations

Instead of restricting the value of a variable to be 0 or 1, we might go in the opposite direction and allow variables to assume any vector of bits as a value. Actually, bit vectors of fixed length correspond to integers in an obvious way, so we have not taken substantial liberties beyond what was done in the RAM model, i.e., we still assume unbounded size for registers when convenient.

However, in those few algorithms where the bit vector model is used, it will be seen that the length of the vectors used is considerably above the number of bits required to represent the size of the problem. The magnitude of most integers used in the algorithm will be of the same order as the size of the problem. For example, dealing with path problems in a 100-vertex graph, we might use bit vectors of length 100 to indicate whether there was a path from a given vertex v to each of the vertices; i.e., the ith position in the vector for vertex v is 1 if and only if there is a path from v to v_i. In the same problem we might also use integers (for counting and indexing, for example) and they would likely be of size on the order of 100. Thus 7 bits would be required for integers, while 100 bits would be needed for the vectors.

The comparison might not be all that lopsided, however, since most computers do logical operations on full-word bit vectors in one instruction cycle. Thus bit vectors of length 100 could be manipulated in three or four steps, in comparison to one step for integers. Nevertheless, we must take *cum grano salis* the results on time and space complexity of algorithms using the bit vector model, as the problem size at which the model becomes unrealistic is much smaller than for the RAM and straight-line code models. We use O_{BV} to denote order of magnitude using the bit vector model.

IV. Decision Trees

We have considered three abstractions of the RAM that ignored branch instructions and that considered only the program steps which involve calculation. There are certain problems where it is realistic to consider the number of branch instructions executed as the primary measure of complexity. In the case of sorting, for example, the outputs are identical to the inputs except for order. It thus becomes reasonable to consider a model in which all steps are two-way branches based on a comparison between two quantities.

The usual representation for a program of branches is a binary tree† called a *decision tree*. Each interior vertex represents a decision. The test represented by the root is made first, and "control" then passes to one of its sons, depending on the outcome. In general, control continues to pass from a vertex to one of its sons, the choice in each case depending on the outcome of the test at the vertex, until a leaf is reached. The desired output is available at the leaf reached.

Example 1.6. Figure 1.18 illustrates a decision tree for a program that sorts three numbers a, b, and c. Tests are indicated by the circled comparisons at the vertices; control moves to the left if the answer to the test is "yes," and to the right if "no." □

† See Section 2.4 for definitions concerning trees.

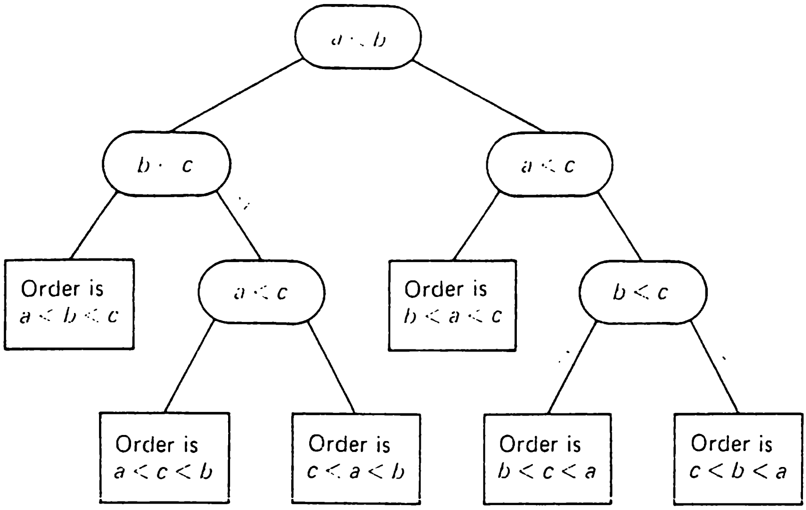


Fig. 1.18 A decision tree.

The time complexity of a decision tree is the height of the tree, as a function of the size of the problem. Normally we wish to measure the maximum number of comparisons which must be made to find our way from the root to a leaf. We use O_C for order of magnitude under the decision tree (comparison) model. Note that the total number of vertices in the tree may greatly exceed its height. For example, a decision tree to sort n numbers must have at least $n!$ leaves, although a tree of height about $n \log n$ suffices.

1.6 A PRIMITIVE MODEL OF COMPUTATION: THE TURING MACHINE

To prove that a particular function requires a certain minimum amount of time, we need a model which is as general as, but more primitive than, the models we have seen. The instruction repertoire must be as limited as possible, yet the model must be able not only to compute anything the RAM can compute, but to do so "almost" as fast. The definition of "almost" that we shall use is "polynomially related."

> **Definition.** We say that functions $f_1(n)$ and $f_2(n)$ are *polynomially related* if there are polynomials $p_1(x)$ and $p_2(x)$ such that for all values of n, $f_1(n) \leq p_1(f_2(n))$ and $f_2(n) \leq p_2(f_1(n))$.

Example 1.7. The two functions $f_1(n) = 2n^2$ and $f_2(n) = n^5$ are polynomially related; we may let $p_1(x) = 2x$ since $2n^2 \leq 2n^5$, and $p_2(x) = x^3$ since $n^5 \leq (2n^2)^3$. However, n^2 and 2^n are not polynomially related, since there is no polynomial $p(x)$ such that $p(n^2) \geq 2^n$ for all n. □

At present, the only range in which we have been able to use general computational models such as the Turing machine to prove lower bounds on computational complexity is the "higher range." For example, in Chapter 11 we show that certain problems require exponential time and space. ($f(n)$ is an *exponential* function if there exist constants $c_1 > 0$, $k_1 > 1$, $c_2 > 0$, and $k_2 > 1$ such that $c_1 k_1^n \leq f(n) \leq c_2 k_2^n$ for all but a finite number of values of n.) In the exponential range, polynomially related functions are essentially the same, since any function which is polynomially related to an exponential function is itself an exponential function.

Thus there is motivation for us to use a primitive model on which the time complexity of problems is polynomially related to their complexity on the RAM model. In particular, the model we use—the multitape Turing machine—may require ($[f(n)]^4$) time† to do what a RAM, under the logarithmic cost function, can do in time $f(n)$, but no more. Thus time complexity on the RAM and Turing machine models will be seen to be polynomially related.

Definition. A *multitape Turing machine* (TM) is pictured in Fig. 1.19. It consists of some number k of *tapes*, which are infinite to the right. Each tape is marked off into *cells*, each of which holds one of a finite number of *tape symbols*. One cell on each tape is scanned by a *tape head*, which can read and write. Operation of the Turing machine is determined by a primitive program called a *finite control*. The finite control is always in one of a finite number of *states*, which can be regarded as positions in a program.

One computational step of a Turing machine consists of the following. In accordance with the current state of the finite control and the tape symbols which are under (*scanned by*) each of the tape heads, the Turing machine may do any or all of these operations:

1. Change the state of the finite control.
2. Print new tape symbols over the current symbols in any or all of the cells under tape heads.
3. Move any or all of the tape heads, independently, one cell left (L) or right (R) or keep them stationary (S).

Formally, we denote a k-tape Turing machine by the seven-tuple

$$(Q, T, I, \delta, \mathrm{b}, q_0, q_f).$$

† Actually, a tighter bound of $O([f(n) \log f(n) \log\log f(n)]^2)$ may be shown, but since we are not concerned with polynomial factors here, the fourth-power result will do (see Section 7.5).

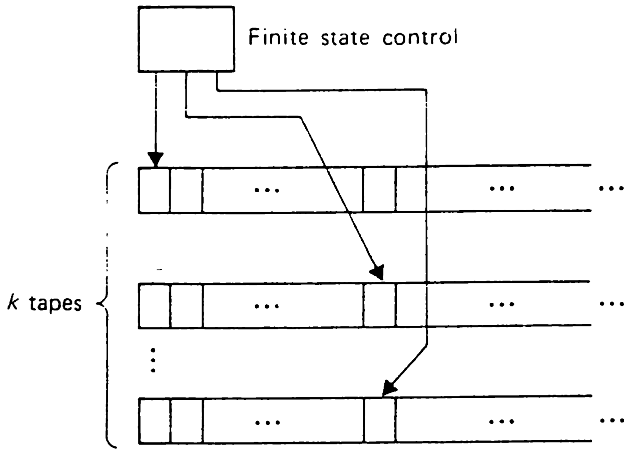


Fig. 1.19 A multitape Turing machine.

where:

1. Q is the set of *states*.
2. T is the set of *tape symbols*.
3. I is the set of *input symbols*; $I \subseteq T$.
4. b. in $T - I$, is the *blank*.
5. q_0 is the *initial state*.
6. q_f is the *final* (or *accepting*) *state*.
7. δ, the *next-move function*, maps a subset of $Q \times T^k$ to $Q \times (T \times \{L, R, S\})^k$. That is, for some $(k + 1)$-tuples consisting of a state and k tape symbols, it gives a new state and k pairs, each pair consisting of a new tape symbol and a direction for the tape head. Suppose $\delta(q, a_1, a_2, \ldots, a_k) = (q', (a_1', d_1), (a_2', d_2), \ldots, (a_k', d_k))$, and the Turing machine is in state q with the ith tape head scanning tape symbol a_i for $1 \leq i \leq k$. Then in one move the Turing machine enters state q', changes symbol a_i to a_i', and moves the ith tape head in the direction d_i for $1 \leq i \leq k$.

A Turing machine can be made to recognize a language as follows. The tape symbols of the Turing machine include the alphabet of the language, called the *input symbols*, a special symbol *blank*, denoted b, and perhaps other symbols. Initially, the first tape holds a string of input symbols, one symbol per cell starting with the leftmost cell. All cells to the right of the cells containing the input string are blank. All other tapes are completely blank. The string of input symbols is *accepted* if and only if the Turing machine, started in the designated initial state, with all tape heads at the left ends of their tapes, makes a sequence of moves in which it eventually enters the accepting state. The *language accepted* by the Turing machine is the set of strings of input symbols so accepted.

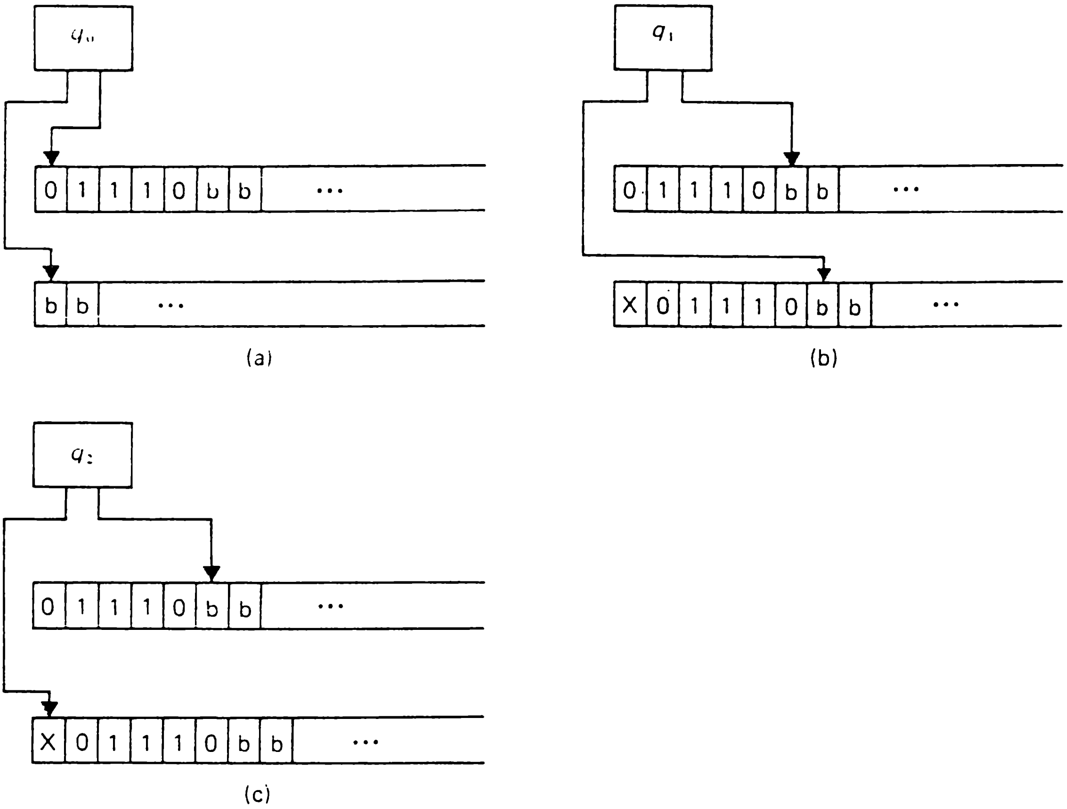


Fig. 1.20 Turing machine processing 01110.

Example 1.8. The two-tape Turing machine in Fig. 1.20 recognizes palindromes[†] on the alphabet $\{0, 1\}$ as follows.

1. The first cell on tape 2 is marked with a special symbol X, and the input is copied from tape 1, where it appears initially (see Fig. 1.20a), onto tape 2 (see Fig. 1.20b).
2. Then the tape head on tape 2 is moved to the X (Fig. 1.20c).
3. Repeatedly, the head of tape 2 is moved right one cell and the head of tape 1 left one cell, comparing the respective symbols. If all symbols match, the input is a palindrome and the Turing machine enters the accepting state q_5. Otherwise, the Turing machine will at some point have no legal move to make; it will halt without accepting.

The next-move function of the Turing machine is given by the table of Fig. 1.21. □

[†] A string which reads the same backwards as forwards, e.g., 0100010, is called a *palindrome*.

Current state	Symbol on:		(New symbol, head move)		New state	Comments
	Tape 1	Tape 2	Tape 1	Tape 2		
q_0	0	b	0,S	X,R	q_1	If input is nonempty, print X on tape 2 and move head right: go to state q_1. Otherwise, go to state q_5.
	1	b	1,S	X,R	q_1	
	b	b	b,S	b,S	q_5	
q_1	0	b	0,R	0,R	q_1	Stay in state q_1 copying tape 1 onto tape 2 until b is reached on tape 1. Then go to state q_2.
	1	b	1,R	1,R	q_1	
	b	b	b,S	b,L	q_2	
q_2	b	0	b,S	0,L	q_2	Keep tape 1's head fixed and move tape 2's left until X is reached. Then go to state q_3.
	b	1	b,S	1,L	q_2	
	b	X	b,L	X,R	q_3	
q_3	0	0	0,S	0,R	q_4	Control alternates between states q_3 and q_4. In q_3 compare the symbols on the two tapes, move tape 2's head right, and go to q_4. In q_4 go to q_5 and accept if head has reached b on tape 2. Otherwise move tape 1's head left and go back to q_3. The alternation q_3, q_4 prevents the input head from falling off the left end of tape 1.
	1	1	1,S	1,R	q_4	
q_4	0	0	0,L	0,S	q_3	
	0	1	0,L	1,S	q_3	
	1	0	1,L	0,S	q_3	
	1	1	1,L	1,S	q_3	
	0	b	0,S	b,S	q_5	
	1	b	1,S	b,S	q_5	
q_5						Accept

Fig. 1.21. Next-move function for Turing machine recognizing palindromes.

The activity of a Turing machine can be described formally by means of "instantaneous descriptions." An *instantaneous description* (ID) of a k-tape Turing machine M is a k-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ where each α_i is a string of the form xqy such that xy is the string on the ith tape of M (with trailing blanks omitted) and q is the current state of M. The symbol immediately to the right of the ith q is the symbol being scanned on the ith tape.

If instantaneous description D_1 becomes instantaneous description D_2 after one move of the Turing machine M, then we write $D_1 \vdash_M D_2$ (read $\vdash$ as "goes to"). If $D_1 \vdash_M D_2 \vdash_M \cdots \vdash_M D_n$ for some $n \geq 2$, then we write $D_1 \overset{+}{\vdash}_M D_n$. If either $D = D'$ or $D \overset{+}{\vdash}_M D'$, then we write $D \overset{*}{\vdash}_M D'$.

The k-tape Turing machine $M = (Q, T, I, \delta, b, q_0, q_f)$ *accepts* string $a_1a_2 \cdots a_n$, where the a's are in I, if $(q_0a_1a_2 \cdots a_n, q_0, q_0, \ldots, q_0) \overset{*}{\underset{M}{\vdash}} (\alpha_1, \alpha_2, \ldots, \alpha_k)$ for some α_i's with q_f in them.

Example 1.9. The sequence of instantaneous descriptions entered by the Turing machine of Fig. 1.21 when presented with the input 010 is shown in Fig. 1.22. Since q_5 is the final state, the Turing machine accepts 010. □

In addition to its natural interpretation as a language acceptor, a Turing machine can be regarded as a device that computes a function f. The arguments of the function are encoded on the input tape as a string x with a special marker such as # separating the arguments. If the Turing machine halts with an integer y (the value of the function) written on a tape designated as the *output tape*, we say $f(x) = y$. Thus the process of computing a function is little different from that of accepting a language.

The *time complexity* $T(n)$ of a Turing machine M is the maximum number of moves made by M in processing any input of length n, taken over all inputs of length n. If for some input of length n the Turing machine does not halt, then $T(n)$ is undefined for that value of n. The *space complexity* $S(n)$ of a Turing machine is the maximum distance from the left end of a tape which

$$
\begin{array}{rl}
(q_0010, q_0) & \vdash (q_1010, Xq_1) \\
& \vdash (0q_110, X0q_1) \\
& \vdash (01q_10, X01q_1) \\
& \vdash (010q_1, X010q_1) \\
& \vdash (010q_2, X01q_20) \\
& \vdash (010q_2, X0q_210) \\
& \vdash (010q_2, Xq_2010) \\
& \vdash (010q_2, q_2X010) \\
& \vdash (01q_30, Xq_3010) \\
& \vdash (01q_40, X0q_410) \\
& \vdash (0q_310, X0q_310) \\
& \vdash (0q_410, X01q_40) \\
& \vdash (q_3010, X01q_30) \\
& \vdash (q_4010, X010q_4) \\
& \vdash (q_5010, X010q_5)
\end{array}
$$

Fig. 1.22. Sequence of Turing machine ID's.

any tape head travels in processing any input of length n. If a tape head moves indefinitely to the right, then $S(n)$ is undefined. We use O_{TM} for order of magnitude under the Turing machine model.

Example 1.10. The time complexity of the Turing machine of Fig. 1.21 is $T(n) = 4n + 3$, and its space complexity is $S(n) = n + 2$, as can be checked by examining the case when the input actually is a palindrome. □

1.7 RELATIONSHIP BETWEEN THE TURING MACHINE AND RAM MODELS

The principal application of the Turing machine (TM) model is in determining lower bounds on the space or time necessary to solve a given problem. For the most part, we can determine lower bounds only to within a polynomially related function. Deriving tighter bounds involves more specific details of a particular model. Fortunately, computations on a RAM or RASP are polynomially related to computations on a TM.

Consider the relationship between the RAM and TM models. Clearly a RAM can simulate a k-tape TM by holding one cell of a TM tape in each of its registers. In particular, the ith cell of the jth tape can be stored in register $ki + j + c$, where c is a constant designed to allow the RAM some "scratch space." Included in the scratch space are k registers to hold the positions of the k heads of the TM. Cells of the TM's tape can be read by the RAM by using indirect addressing through the registers holding the tape head positions.

Suppose the TM is of time complexity $T(n) \geq n$. Then the RAM can read its input, store it in the registers representing the first tape, and simulate the TM in $O(T(n))$ time if the uniform cost function is used or in $O(T(n) \log T(n))$ time if the logarithmic cost function is used. In either case, the time on the RAM is bounded above by a polynomial function of the time on the TM, since any $O(T(n) \log T(n))$ function is certainly $O(T^2(n))$.

A converse result holds only under the logarithmic cost for RAM's. Under the uniform cost an n-step RAM program, without input, can compute numbers as high as 2^{2^n}, which requires 2^n TM cells just to store and read. Thus under the uniform cost no polynomial relationship between RAM's and TM's is apparent (Exercise 1.19).

Although we prefer the uniform cost for its simplicity when analyzing algorithms, we must reject it when attempting to prove lower bounds on time complexity. The RAM with uniform cost is quite reasonable when numbers do not grow out of proportion with the size of the problem. But, as we said previously, with the RAM model the size of numbers is "swept under the rug," and rarely can useful lower bounds be obtained. For the logarithmic cost, however, we have the following theorem.

Theorem 1.3. Let L be a language that is accepted by a RAM program of time complexity $T(n)$ under the logarithmic cost criterion. If the RAM program uses no multiplications or divisions, then L is of time complexity at most $O(T^2(n))$ on a multitape Turing machine.

#	#	i_1	#	c_1	#	#	i_2	#	c_2	#	#	· · ·	i_k	#	c_k	#	#	b	· · ·

Fig. 1.23. TM representation of RAM.

Proof. We represent all the RAM registers not holding 0 as shown in Fig. 1.23. The tape consists of a sequence of pairs (i_j, c_j) written in binary with no leading 0's and separated by marker symbols. For each j, c_j is the contents of RAM register i_j. The contents of the RAM accumulator is stored in binary on a second tape, and a third tape is used for scratch memory. Two other tapes serve to hold the input and output of the RAM. Each step of the RAM program is represented by a finite set of states of the TM. We shall not describe the simulation of an arbitrary RAM instruction, but shall consider only the instructions ADD *20 and STORE 30, which should make the ideas clear. For ADD *20, we can design the TM to do the following:

1. Search tape 1 for an entry for RAM register 20, i.e., a sequence ##10100#. If found, place the integer following, which will be the contents of register 20, on tape 3. If not found, then halt. The contents of register 20 is 0 and thus the indirect addressing cannot be done.
2. Look on tape 1 for an entry for the RAM register whose number is stored on tape 3. If found, copy the contents of that register onto tape 3. If not found, place 0 there.
3. Add the number placed on tape 3 in step 2 to the contents of the accumulator, which is held on tape 2.

To simulate the instruction STORE 30, we can design the TM to do the following:

1. Search for an entry for RAM register 30, i.e., ##11110#.
2. If found, copy everything to the right of ##11110#, except for the integer immediately following (the old contents of register 30), onto tape 3. Then copy the contents of the accumulator (tape 2) immediately to the right of ##11110# and follow it by the string copied onto tape 3.
3. If no entry for register 30 was found on tape 1, instead go to the leftmost blank, print 11110#, followed by the contents of the accumulator, followed by ##.

With a little thought, it should be evident that the TM can be designed to simulate the RAM faithfully. We must show that a RAM computation of logarithmic cost k requires at most $O(k^2)$ steps of the Turing machine. We begin by observing that a register will not appear on tape 1 unless its current value was stored into the register at some previous time. The cost of storing c_j into register i_j is $l(c_j) + l(i_j)$, which is, to within a constant, the length of the representation $\#\#i_j\#c_j\#\#$. We conclude that the length of the nonblank portion of tape 1 is $O(k)$.

The simulation of any instruction other than a STORE is on the order of the length of tape 1, that is, $O(k)$, since the dominant cost is a search of the tape. Similarly, the cost of a STORE is no more than the cost of searching tape 1 plus the cost of copying it, both $O(k)$. Hence one RAM instruction (except for multiply and divide) can be simulated in at most $O(k)$ steps of the TM. Since a RAM instruction costs at least one time unit under the logarithmic cost criterion, the total time spent by the TM is $O(k^2)$, as was to be proved. □

If a RAM program employs multiply and divide instructions, then we can write subroutines of the TM to implement these instructions by means of additions and subtractions. We leave it to the reader to show that the logarithmic cost of the subroutines is no greater than the square of the logarithmic cost of the instructions they simulate. It is thus not hard to prove the following theorem.

Theorem 1.4. The RAM and RASP under logarithmic cost and the multitape Turing machine are all polynomially related models.

Proof. Use Theorems 1.1, 1.2, and 1.3 and your analysis of multiplication and division subroutines. □

An analogous result holds for space complexity, although the result appears less interesting.

1.8 PIDGIN ALGOL—A HIGH-LEVEL LANGUAGE

Although our basic measures of complexity are in terms of operations on a RAM, RASP, or Turing machine, we do not generally want to describe algorithms in terms of such primitive machines, nor is it necessary. In order to describe algorithms more clearly we shall use a high-level language called Pidgin ALGOL.

A Pidgin ALGOL program can be translated into a RAM or RASP program in a straightforward manner. Indeed this would be precisely the role of a Pidgin ALGOL compiler. We shall not, however, concern ourselves with the details of translating Pidgin ALGOL into RAM or RASP code. For our purposes, it is necessary to consider only the time and space necessary to execute the code corresponding to a Pidgin ALGOL statement.

Pidgin ALGOL is unlike any conventional programming language in that it allows the use of any type of mathematical statement as long as its meaning is clear and the translation into RAM or RASP code is evident. Similarly, the language does not have a fixed set of data types. Variables can represent integers, strings, and arrays. Additional data types such as sets, graphs, lists, and queues can be introduced as needed. Formal declarations of data types are avoided as much as possible. The data type of a variable and its scope[†] should be evident either from its name or from its context.

Pidgin ALGOL uses traditional mathematical and programming language constructs such as expressions, conditions, statements, and procedures. Informal descriptions of some of these constructs are given below. No attempt is made to give a precise definition, as that would be far beyond the scope of the book. It should be recognized that one can easily write programs whose meaning depends on details not covered here, but one should refrain from doing so, and we have (hopefully) done so in this book.

A Pidgin ALGOL *program* is a statement of one of the following types.

```
1.  variable ← expression
2.  if condition then statement else statement‡
3a. while condition do statement
 b. repeat statement until condition
4.  for variable ← initial-value step step-size§ until final-value do statement
5.  label: statement
6.  goto label
7.  begin
        statement:
        statement:
            .
            .
            .
        statement;
        statement
    end
8a. procedure name (list of parameters): statement
 b. return expression
 c. procedure-name (arguments)
9a. read variable
 b. write expression
10. comment comment
11. any other miscellaneous statement
```

[†] The *scope* of a variable is the environment in which it has a meaning. For example, the scope of an index of a summation is defined only within the summation and has no meaning outside the summation.

[‡] "else statement" is optional. This option leads to the usual "dangling else" ambiguity. We take the traditional way out and assume **else** to be matched with the closest unmatched **then.**

[§] "**step** step-size" is optional if step-size is 1.

We shall give a brief synopsis of each of these statement types.

1. The assignment statement

variable ← expression

causes the expression to the right of ← to be evaluated and the resulting value to be assigned to the variable on the left. The time complexity of the assignment statement is the time taken to evaluate the expression and to assign the value to the variable. If the value of the expression is not a basic data type, such as an integer, one may in some cases reduce the cost by means of pointers. For example, the assignment $A \leftarrow B$ where A and B are $n \times n$ matrices would normally require $O(n^2)$ time. However, if B is no longer used, then the time can be made finite and independent of n by simply renaming the array.

2. In the **if** statement

if condition **then** statement **else** statement

the condition following the **if** can be any expression that has a value **true** or **false.** If the condition has the value **true,** the statement following **then** is to be executed. Otherwise, the statement following **else** (if present) is to be executed. The cost of the **if** statement is the sum of the costs required to evaluate and test the expression plus the cost of the statement following **then** or the cost of the statement following **else,** whichever is actually executed.

3. The purpose of the **while** statement

while condition **do** statement

and the **repeat** statement

repeat statement **until** condition

is to create a loop. In the **while** statement the condition following **while** is evaluated. If the condition is **true,** the statement after the **do** is executed. This process is repeated until the condition becomes **false.** If the condition is originally **true,** then eventually an execution of the statement must cause the condition to become **false** if the execution of the **while** statement is to terminate. The cost of the **while** statement is the sum of the costs of evaluating the condition as many times as it is evaluated plus the sum of the costs of executing the statement as many times as it is executed.

The **repeat** statement is similar except that the statement following **repeat** is executed before the condition is evaluated.

4. In the **for** statement

for variable ← initial-value **step** step-size **until** final-value **do** statement

initial-value, step-size, and final-value are all expressions. In the case where step-size is positive the variable (called the *index*) is set equal to the value of the initial-value expression. If this value exceeds the final-value, then execution terminates. Otherwise the statement following **do** is executed, the value of the variable is incremented by step-size and compared with the final-value The process is repeated until the value of the variable exceeds the final-value The case where the step-size is negative is similar, but termination occurs when the value of the variable is less than the final-value. The cost of the **for** statement should be obvious in light of the preceding analysis of the **while** statement.

The above description completely ignores such details as when the expressions for initial-value, step-size, and final-value are evaluated. It is possible that the execution of the statement following **do** modifies the value of the expression step-size, in which case evaluating the expression for step-size every time the variable is incremented has an effect different from evaluating step-size once and for all. Similarly, evaluating step-size can affect the value of final-value, and a change in sign of step-size changes the test for termination. We resolve these problems by not writing programs where such phenomena would make the meaning unclear.

5. Any statement can be made into a *labeled statement* by prefixing it with a label followed by a colon. The primary purpose of the label is to establish a target for a **goto** statement. There is no cost associated with the label.

6. The **goto** statement

goto label

causes the statement with the given label to be executed next. The statement so labeled is not allowed to be inside a block-statement (7) unless the **goto** statement is inside the same block-statement. The cost of the **goto** statement is one. **goto** statements should be used sparingly, since they generally make programs difficult to understand. The primary use of **goto** statements is to break out of **while** statements.

7. A sequence of statements separated by semicolons and nested between the keywords **begin** and **end** is a statement which is called a *block*. Since a block is a statement, it can be used wherever a statement can be used. Normally, a program will be a block. The cost of a block is the sum of the costs of the statements appearing within the block.

8. *Procedures*. In Pidgin ALGOL procedures can be defined and subsequently invoked. Procedures are defined by the *procedure-definition statement* which is of the form:

procedure name (list of parameters): statement

The list of parameters is a sequence of dummy variables called *formal parameters*. For example, the following statement defines a function procedure named MIN.

```
procedure MIN($x$, $y$):
if $x > y$ then return $y$ else return $x$
```

The arguments x and y are formal parameters.

Procedures are used in one of two ways. One way is as a *function*. After a function procedure has been defined, it can be invoked in an expression by using its name with the desired arguments. In this case the last statement executed in the procedure must be a **return** statement 8(b). The **return** statement causes the expression following the keyword **return** to be evaluated and execution of the procedure to terminate. The value of the function is the value of this expression. For example,

$$A \leftarrow \text{MIN}(2+3, 7)$$

causes A to receive the value 5. The expressions $2 + 3$ and 7 are called the *actual parameters* of this procedure invocation.

The second method of using a procedure is to call it by means of the procedure-calling statement 8(c). This statement is merely the name of the procedure followed by a list of actual parameters. The procedure-calling statement can (and usually does) modify the data of the calling program. A procedure called this way does not need a **return** statement in its definition. Completion of execution of the last statement in the procedure completes the execution of the procedure-calling statement. For example, the following statement defines a procedure named INTERCHANGE.

```
procedure INTERCHANGE($x$, $y$):
begin
    $t \leftarrow x$;
    $x \leftarrow y$;
    $y \leftarrow t$
end
```

To invoke this procedure we could write a procedure-calling statement such as

$$\text{INTERCHANGE}(A[i], A[j])$$

There are two methods by which a procedure can communicate with other procedures. One way is by global variables. We assume that global variables are implicitly declared in some universal environment. Within this environment is a subenvironment in which procedures are defined.

The other method of communicating with procedures is by means of the parameters. ALGOL 60 uses call-by-value and call-by-name. In *call-by-value* the formal parameters of a procedure are treated as local variables which are initialized to the values of the actual parameters. In *call-by-name* formal parameters serve as place holders in the program, actual parameters being substituted for every occurrence of the corresponding formal parameters. For simplicity we depart from ALGOL 60 and use call-by-reference. In *call-by-reference* parameters are passed by means of pointers to the actual parameters. If an actual parameter is an expression (possibly a constant), then the corresponding formal parameter is treated as a local variable initialized to the value of the expression. Thus the cost of a function or procedure-call in a RAM or RASP implementation is the sum of the costs of executing the statements in the definition of the procedure. The cost and implementation of a procedure that calls other procedures, possibly itself, is discussed in Chapter 2.

9. The **read** statement and **write** statement have the obvious meaning. The **read** statement has a cost of one. The **write** statement has a cost of one plus the cost of evaluating the expression following the keyword **write.**

10. The **comment** statement allows insertion of remarks to aid in the understanding of the program and has zero cost.

11. In addition to the conventional programming language statements we include under "miscellaneous" any statement which makes an algorithm more understandable than an equivalent sequence of programming language statements. Such statements are used when the details of implementation are either irrelevant or obvious, or when a higher level of description is desirable. Some examples of commonly used miscellaneous statements are:

a) let a be the smallest element of set S
b) mark element a as being "old"†
c) **without loss of generality (wlg)** assume that . . . **otherwise** . . . **in** statement
For example,

wlg assume $a \leq b$ **otherwise** interchange c and d **in** statement

means that if $a \leq b$ the following statement is to be executed as written. If $a > b$, a duplicate of the statement with the roles of c and d interchanged is to be executed.

Implementation of these statements in terms of conventional programming language statements or in terms of RAM code is straightforward but tedious. Assignment of a cost to statements of this nature depends on the context in which the statement is found. Further examples of statements of this nature will be found throughout the Pidgin ALGOL programs in this book.

Since variables will usually not be declared, we should state some conventions concerning the scope of variables. In a given program or procedure we do not use the same name for two different variables. Thus the scope of a variable can usually be taken to be the entire procedure or program in which

† By this we suppose there is an array STATUS, such that STATUS[a] is 1 if a is "old" and 0 if a is "new."

it occurs.† One important exception occurs when there is a common data base on which several procedures operate. In this case the variables of the common data base are assumed to be global and the variables used by the procedure for temporary storage in manipulating the data base are assumed to be local to the procedure. Whenever confusion can arise concerning the scope of a variable, an explicit declaration will be made.

EXERCISES

1.1 Prove that $g(n)$ is $O(f(n))$ if (a) $f(n) \geq \epsilon$, for some $\epsilon > 0$ and for all but some finite set of n and (b) there exist constants $c_1 > 0$ and $c_2 > 0$ such that $g(n) \leq c_1 f(n) + c_2$ for almost all $n \geq 0$.

1.2 Write $f(n) \preccurlyeq g(n)$ if there exists a positive constant c such that $f(n) \leq cg(n)$ for all n. Show that $f_1 \preccurlyeq g_1$ and $f_2 \preccurlyeq g_2$ imply $f_1 + f_2 \preccurlyeq g_1 + g_2$. What other properties are enjoyed by the relation $\preccurlyeq$?

1.3 Give RAM, RASP, and Pidgin ALGOL programs to do the following:

a) Compute $n!$ given input n.

b) Read n positive integers followed by an endmarker (0) and then print the n numbers in sorted order.

c) Accept all inputs of the form $1^n 2^{n^2} 0$.

1.4 Analyze the time and space complexities of your answers to Exercise 1.3 under (a) the uniform and (b) the logarithmic cost. State your measure of the "size" of the input.

*1.5 Write a RAM program of uniform-cost time complexity $O(\log n)$ to compute n^n. Prove that your program is correct.

*1.6 Show that for each RAM program of time complexity $T(n)$ under the logarithmic cost function there is an equivalent RAM program of time complexity $O(T^2(n))$ which has no MULT or DIV instructions. [*Hint:* Simulate MULT and DIV by subroutines that use even-numbered registers for scratch storage. For MULT, show that if i is to be multiplied by j, you can compute each of the $l(j)$ partial products and sum them in $O(l(j))$ steps, each step requiring $O(l(i))$ time.]

*1.7 What happens to the computing power of a RAM or RASP if both MULT and ADD are removed from the instruction repertoire? How is the cost of computation affected?

**1.8 Show that any language accepted by a RAM can be accepted by a RAM without indirect addressing. [*Hint:* Show that an entire Turing machine tape can be encoded as a single integer. Thus any TM can be simulated in a finite number of registers of a RAM.]

† There are some unimportant exceptions to this statement. For example, a procedure may have two unnested **for** statements both with the index i. Strictly speaking, the scope of the index of a **for** statement is the **for** statement itself, and thus the i's are different variables.

1.9 Show that under (a) uniform and (b) logarithmic cost, the RAM and RASP are equivalent in space complexity, to within a constant factor.

1.10 Find a straight-line program that computes the determinant of a 3×3 matrix, given its nine scalar elements as inputs.

1.11 Write a sequence of bit operations to compute the product of two 2-bit integers.

1.12 Show that the set of functions computed by any n-statement straight-line program with binary inputs and Boolean operators can be implemented by a combinational logic circuit with n Boolean circuit elements.

1.13 Show that any Boolean function can be computed by a straight-line program.

*1.14 Suppose an n-vertex graph is represented by a set of bit vectors $\mathbf{v}_i$, where the jth component of $\mathbf{v}_i$ is 1 if and only if there is an edge from vertex i to vertex j. Find an $O_{BV}(n)$ algorithm to determine the vector $\mathbf{p}_1$ which has 1 in position j if and only if there is a path from 1 to vertex j. The operations you may use are the bitwise logical operations on bit vectors, arithmetic operations (on variables which are of "integer type"), instructions which set particular bits of particular vectors to 0 or 1, and an instruction which assigns j to integer variable a if the leftmost 1 in vector $\mathbf{v}$ is in position j, and sets $a = 0$ if $\mathbf{v}$ is all 0's.

*1.15 Specify a Turing machine which when given two binary integers on tapes 1 and 2 will print their sum on tape 3. You may assume the left ends of the tapes are marked by a special symbol #.

1.16 Give the sequence of configurations entered by the TM of Fig. 1.21 (p. 29) when presented with input (a) 0010, (b) 01110.

*1.17 Give a TM which does the following:
a) Prints 0^{n^2} on tape 2 when started with 0^n on tape 1.
b) Accepts inputs of the form $0^n10^{n^2}$.

1.18 Give a set of TM states and a next-move function to allow a TM to simulate the RAM instruction LOAD 3 as in the proof of Theorem 1.3.

1.19 Give an $O(n)$ step RAM program which computes 2^{2^n} given n. What is the (a) uniform and (b) logarithmic cost of your program?

*1.20 Define $g(m, n)$ by $g(0, n) = n$ and $g(m, n) = 2^{g(m-1, n)}$ for $m > 0$. Give a RAM program to compute $g(n, n)$, given n. How do the uniform and logarithmic costs of your program compare?

1.21 Execute the procedure INTERCHANGE of Section 1.8 with actual parameters i and $A[i]$ using call-by-name, then using call-by-reference. Are the results the same?

Research Problem

1.22 Can the $O(T^2(n))$ upper bound on the time required for a Turing machine to simulate a RAM, as in Theorem 1.3, be improved?

BIBLIOGRAPHIC NOTES

The RAM and RASP have received formal treatment in Shepherdson and Sturgis [1963], Elgot and Robinson [1964], and Hartmanis [1971]. Most of the results on RAM's and RASP's presented here are patterned after Cook and Reckhow [1973].

The Turing machine is due to Turing [1936]. A more detailed exposition of the concept can be found in Minsky [1967] or Hopcroft and Ullman [1969], as can the answer to Exercise 1.8. Time complexity of Turing machines was first studied by Hartmanis and Stearns [1965], and space complexity by Hartmanis, Lewis, and Stearns [1965] and Lewis, Stearns, and Hartmanis [1965]. The notion of computational complexity has received much abstract treatment, beginning with Blum [1967]. Surveys can be found in Hartmanis and Hopcroft [1971] and Borodin [1973a].

Rabin [1972] provides an interesting extension to the decision tree model of computation.

DESIGN OF EFFICIENT ALGORITHMS

CHAPTER 2

The purpose of this chapter is twofold. First, we introduce some basic data structures that are useful in designing efficient algorithms for large classes of problems. Second, we introduce some "programming" techniques, such as recursion and dynamic programming, that are common to many efficient algorithms.

In Chapter 1 we considered the basic models of computation. Although our primary model is the RAM, we do not normally want to describe algorithms in terms of such a basic device. We therefore introduced Pidgin ALGOL (Section 1.8). But even this language is too primitive unless we introduce data structures that are more complex than arrays. We begin this chapter by familiarizing the reader with elementary data structures, such as lists and stacks, which are frequently used in efficient algorithms. We indicate how these structures can be used to represent sets, graphs, and trees. The treatment is necessarily brief, and the reader not familiar with list processing should consult one of the more basic references at the end of the chapter or give special attention to the exercises.

We have also included a section on recursion. One of the important aspects of recursion is the resulting conceptual simplification of algorithms. Although the examples in this chapter are too simple to substantiate this claim fully, the suppression of bookkeeping details by the use of recursion is most useful in being able to express concisely the more complex algorithms in later chapters. Recursion by itself does not necessarily lead to more efficient algorithms. However, when it is combined with other techniques such as balancing, divide-and-conquer, and algebraic simplification, we shall see that it often yields algorithms that are both efficient and elegant.

2.1 DATA STRUCTURES: LISTS, QUEUES, AND STACKS

We assume that the reader is familiar both with elementary concepts of mathematics such as sets and relations and with basic data types such as integers, strings, and arrays. In this section we provide a quick review of basic list operations.

Mathematically, a *list* is a finite sequence of items drawn from some set pertinent to the application at hand. Often the description of an algorithm will involve a list to which items are added and deleted. In particular, we may want to add or delete an item somewhere in the middle of a list. For this reason we wish to develop data structures that allow us to implement lists in which items can be added or deleted at will.

Consider the list

$$\text{Item 1, Item 2, Item 3, Item 4} \tag{2.1}$$

The simplest implementation of this list is the singly linked structure illustrated in Fig. 2.1. Each element in the structure consists of two memory

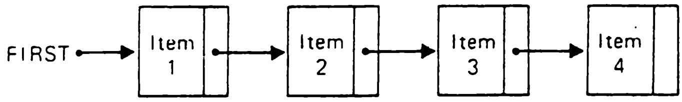


Fig. 2.1 A linked list.

	NAME	NEXT
0	–	1
1	Item 1	3
2	Item 4	0
3	Item 2	4
4	Item 3	2

Fig. 2.2. Representation of a list with four items.

locations. The first contains the item itself,[†] the second contains a pointer to the next element. One possible implementation is in terms of two arrays which in Fig. 2.2 are called NAME and NEXT.[‡] If ELEMENT is an index into the arrays, then NAME[ELEMENT] is the item stored and NEXT[ELEMENT] is the index of the next item on the list, provided there is a next item. If ELEMENT is the index of the last item on the list, then NEXT[ELEMENT] = 0.

In Fig. 2.2 we have used NEXT[0] as a permanent pointer to the first element of the list. Note that the order of the items in the array NAME is not the same as their order on the list. However, Fig. 2.2 is a faithful representation of Fig. 2.1, since the NEXT array sequences the items as they appear on the list (2.1).

The following procedure inserts an element into a list. It assumes that FREE is the index of an unused location in the arrays NAME and NEXT

[†] If the item is itself a complex structure, then the first location might contain a pointer to the item.

[‡] An alternative (and equivalent) view is that there is a "cell" for each element. Each cell has an "address," which is the first (possibly only) memory register in a block of registers reserved for that element. Within each cell are one or more "fields." Here the fields are NAME and NEXT, and NAME[ELEMENT] and NEXT[ELEMENT] are used to refer to the contents of these fields in the cell whose address is ELEMENT.

is obtained by deleting the first cell on the free list. On deleting an item from list *A*, the cell is returned to the free list for future use.

This method of storage management is not the only method in use but is presented to establish that the operation of adding or deleting items from lists can be accomplished in a bounded number of operations once we have determined where the item is to be inserted or deleted.

Other basic operations on lists are concatenation of two lists to form a single list and the inverse operation of cutting a list after some element to make two lists. The operation of concatenation can be performed in bounded time by adding another pointer to the representation of a list. This pointer gives the index of the last element on the list and obviates the need to search the entire list to find the last element. The cutting operation can be made bounded if we are given the index of the element immediately preceding the cut.

Lists can be traversed in both directions by adding another array called PREVIOUS. The value of PREVIOUS[*I*] is the location of the item on the list immediately before the item which is in location *I*. A list of this nature is said to be *doubly linked.* In a doubly linked list we can delete an item or insert an item without being given the location of the previous item.

Often a list is manipulated in a very restricted manner. For example, items might be added or deleted only at the end of a list. That is, items are inserted and deleted in a last-in, first-out fashion. In this case the list is referred to as a *stack* or *pushdown store.*

Often a stack is implemented as a single array. For example, the list

Item 1, Item 2, Item 3

could be stored in the array NAME as shown in Fig. 2.4. The variable TOP

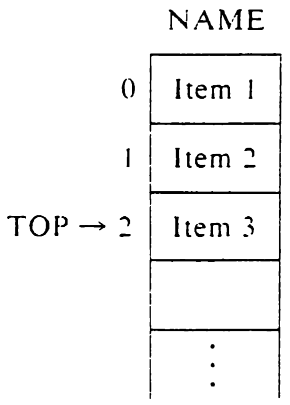


Fig. 2.4. Implementation of a stack.

is a pointer to the last item added to the stack. To add (PUSH) a new item onto the stack, we set TOP to TOP + 1 and then place the new item in NAME[TOP]. (Since the array NAME is of finite length l, we should check that TOP $< l - 1$ before trying to insert the new item.) To delete (POP) an item from the top of the stack, we simply set TOP to TOP − 1. Note that it is not necessary to physically erase the item deleted from the stack. An empty stack is detected by checking to see whether TOP has a value less than zero. Clearly, the execution time of the operations PUSH, POP, and the test for emptiness are independent of the number of elements on the stack.

Another special form of list is the *queue*, a list in which items are always added to one end (the front) and removed from the other. As with a stack, we may implement a queue by a single array as shown in Fig. 2.5, which shows a queue containing the list of items P, Q, R, S, T. Two pointers indicate the locations of the current front and rear of the queue. To add (ENQUEUE) a new item to a queue we set FRONT to FRONT + 1 and store the new item in NAME[FRONT], as for a stack. To remove (DEQUEUE) an item from

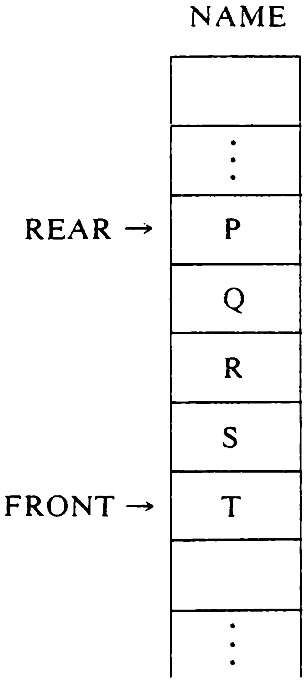


Fig. 2.5. Single array implementation of a queue.

a queue we set REAR to REAR + 1. Note that this technique causes items to be accessed on a first-in, first-out basis.

Since the array NAME is of finite length, say l, the pointers FRONT and REAR will eventually reach the end of the array. If the length of the list represented by the queue never exceeds l, then we may treat NAME[0] as though it followed NAME[$l-1$].

Items stored on a list may themselves be complicated structures. In manipulating a list of arrays, for example, one does not actually add or delete arrays, since each addition or deletion would require time proportional to the size of the array. Instead one adds or deletes pointers to the arrays. Thus a complex structure can be added or deleted in fixed time independent of its size.

2.2 SET REPRESENTATIONS

A common use of a list is to represent a set. With this representation the amount of memory required to represent a set is proportional to the number of elements in the set. The amount of time required to perform a set operation depends on the nature of the operation. For example, suppose A and B are two sets. An operation such as $A \cap B$ requires time at least proportional to the sum of the sizes of the two sets, since the list representing A and the list representing B must each be scanned at least once.†

The operation $A \cup B$ likewise requires time at least proportional to the sum of the set sizes, since we must check for the same element appearing in both sets and delete one instance of each such element. If A and B are disjoint, however, we may find $A \cup B$ in time independent of the size of A and B by simply concatenating the two lists representing A and B. The matter of disjoint set unions is made more complicated if we also require a fast method of determining whether a given element is in a given set. We discuss this subject more fully in Sections 4.6 and 4.7.

An alternative to the list is a bit vector representation of sets. Assume the universe of discourse U (of which all sets are subsets) has n members. Linearly order the elements of U. A subset $S \subseteq U$ is represented as a vector $\mathbf{v}_S$ of n bits, where the ith bit in $\mathbf{v}_S$ is 1 if and only if the ith element of U is an element of S. We call $\mathbf{v}_S$ the *characteristic vector* for S.

The bit vector representation has the advantage that one can determine whether the ith element of U is an element of a set in time independent of the size of the set. Furthermore, basic operations on sets such as union and intersection can be carried out by the bit vector operations $\vee$ and $\wedge$.

If we do not wish to consider bit vector operations to be primitive (requiring one time unit), then we can achieve the effect of a characteristic vector

† If the two lists are sorted, then a linear algorithm to find their intersection exists.

by defining an array A, such that $A[i] = 1$ if and only if the ith member of U is in set S. In this representation, it is still easy to determine whether an element is a member of a set. The disadvantage is that unions and intersections require time proportional to $\|U\|$† rather than the sizes of the sets involved. Likewise, the space required to store set S is proportional to $\|U\|$ rather than $\|S\|$.

2.3 GRAPHS

We now introduce the mathematical notion of a graph and the data structures commonly used to represent a graph.

> **Definition.** A *graph* $G = (V, E)$ consists of a finite, nonempty set of *vertices* V and a set of *edges* E. If the edges are ordered pairs (v, w) of vertices, then the graph is said to be *directed;* v is called the *tail* and w the *head* of the edge (v, w). If the edges are unordered pairs (sets) of distinct vertices, also denoted by (v, w), then the graph is said to be *undirected.*‡

In a directed graph $G = (V, E)$, if (v, w) is an edge in E, then we say vertex w is *adjacent* to vertex v. We also say edge (v, w) is *from* v *to* w. The number of vertices adjacent to v is called the *(out-) degree* of v.

In an undirected graph $G = (V, E)$, if (v, w) is an edge in E we assume $(w, v) = (v, w)$, so (w, v) is the same edge. We say w is *adjacent* to v if (v, w) [and therefore (w, v)] is in E. The *degree* of a vertex is the number of vertices adjacent to it.

A *path* in a directed or undirected graph is a sequence of edges of the form $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$. We say that the path is *from* v_1 *to* v_n and is of *length* $n - 1$. Often such a path is represented by the sequence $v_1, v_2, \ldots, v_n$ of vertices on the path. As a special case, a single vertex denotes a path of length 0 from itself to itself. A path is *simple* if all edges and all vertices on the path, except possibly the first and last vertices, are distinct. A *cycle* is a simple path of length at least 1 which begins and ends at the same vertex. Note that in an undirected graph, a cycle must be of length at least 3.

There are several common representations for a graph $G = (V, E)$. One such is the *adjacency matrix,* a $\|V\| \times \|V\|$ matrix A of 0's and 1's, where the ijth element, $A[i, j]$, is 1 if and only if there is an edge from vertex i to vertex j. The adjacency matrix representation is convenient for graph algorithms which frequently require knowledge of whether certain edges are present, since the time needed to determine whether an edge is present is fixed and independent of $\|V\|$ and $\|E\|$. The main drawback to using an adjacency matrix is that it requires $\|V\|^2$ storage even if the graph has only $O(\|V\|)$ edges. Simply to initialize the adjacency matrix in the straightforward manner requires $O(\|V\|^2)$ time, which would preclude $O(\|V\|)$ algorithms for manipu-

† We use $\|X\|$ for the number of elements in (size or *cardinality* of) set X.

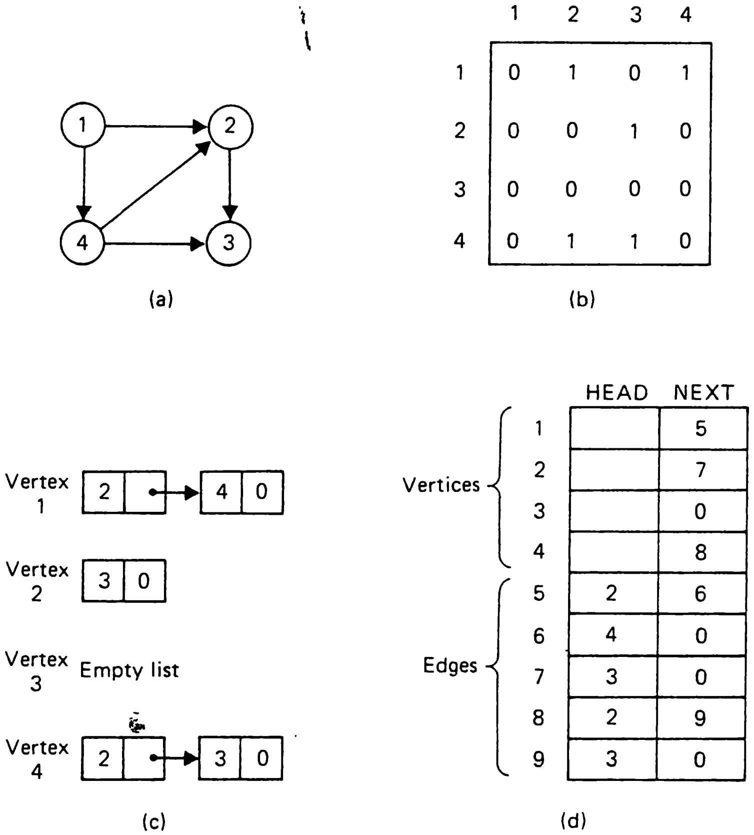
‡ Note that (a, a) may be an edge of a directed graph, but not an undirected one.

lating graphs with only $O(\|V\|)$ edges. Although there exist methods to overcome this difficulty (see Exercise 2.12), other problems almost invariably arise which make $O(\|V\|)$ algorithms based on adjacency matrices rare.

An interesting alternative is to represent the rows and/or columns of an adjacency matrix by bit vectors. Such a representation may introduce considerable efficiency into graph algorithms.

Another possible representation for a graph is by means of lists. The *adjacency list* for a vertex v is a list of all vertices w adjacent to v. A graph can be represented by $\|V\|$ adjacency lists, one for each vertex.

Example 2.2. Figure 2.6(a) illustrates a directed graph with four vertices. Figure 2.6(b) shows the adjacency matrix. Figure 2.6(c) shows the four adjacency lists, one-for each vertex. For example, there are edges from vertex 1



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

		HEAD	NEXT
Vertices	1		5
	2		7
	3		0
	4		8
Edges	5	2	6
	6	4	0
	7	3	0
	8	2	9
	9	3	0

Fig. 2.6 A directed graph and its representations: (a) directed graph; (b) adjacency matrix; (c) adjacency lists; (d) tabular representation of adjacency lists.

to vertices 2 and 4, so the adjacency list for 1 has items 2 and 4 linked together in the format of Fig. 2.1.

A tabular representation of the adjacency lists is shown in Fig. 2.6(d). Each of the first four locations in the array NEXT holds a pointer to the first vertex on an adjacency list, with NEXT[i] pointing to the first vertex on the adjacency list for vertex i. Note that NEXT[3] = 0, since there are no vertices on the adjacency list of vertex 3. The remaining entries of array NEXT represent the edges of the graph. The array HEAD contains the vertices in the adjacency lists. Thus the adjacency list for vertex 1 begins at location 5, since NEXT[1] = 5. HEAD[5] = 2, indicating that there is an edge (1, 2). NEXT[5] = 6, and HEAD[6] = 4 indicating the edge (1, 4). NEXT[6] = 0, indicating that there are no more edges with tail 1. □

Note that the adjacency list representation of a graph requires storage proportional to $\|V\| + \|E\|$. The adjacency list representation is often used when $\|E\| << \|V\|^2$.

If the graph is undirected, then each edge (v, w) is represented twice, once in the adjacency list of v and once in the adjacency list of w. In this case one might add a new array called LINK to correlate both copies of an undirected edge. Thus if i is the location of vertex w in the adjacency list of v, LINK[i] is the location of v in the adjacency list of w.

If we wish to conveniently delete edges from an undirected graph, adjacency lists can be doubly linked. This is usually necessary because even if we always delete an edge (v, w) which is the first edge on the adjacency list for vertex v, the edge in the reverse direction may be in the middle of the adjacency list for vertex w. In order to delete edge (v, w) from the adjacency list of w quickly, we must be able to find the location of the previous edge on this adjacency list quickly.

2.4 TREES

Next, we introduce a very important kind of directed graph, the tree, and we consider the data structures appropriate to represent it.

Definition. A directed graph with no cycles is called a *directed acyclic graph*. A *(directed) tree* (sometimes called a *rooted tree*) is a directed acyclic graph satisfying the following properties:

1. There is exactly one vertex, called the *root*, which no edges enter.
2. Every vertex except the root has exactly one entering edge.
3. There is a path (which is easily shown unique) from the root to each vertex.

A directed graph consisting of a collection of trees is called a *forest*. Forests and trees are special cases of directed acyclic graphs that arise so frequently that we shall develop additional terminology to discuss them.

Definition. Let $F = (V, E)$ be a graph which is a forest. If (v, w) is in E, then v is called the *father* of w, and w is a *son* of v. If there is a path from v to w, then v is an *ancestor* of w and w is a *descendant* of v. Furthermore if $v \neq w$, then v is a *proper ancestor* of w, and w is a *proper descendant* of v. A vertex with no proper descendants is called a *leaf*. A vertex v and all its descendants are called a *subtree* of F. The vertex v is called the *root* of that subtree.

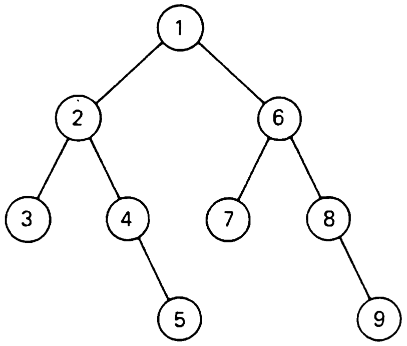
The *depth of a vertex* v in a tree is the length of the path from the root to v. The *height of a vertex* v in a tree is the length of a longest path from v to a leaf. The *height of a tree* is the height of the root. The *level* of a vertex v in a tree is the height of the tree minus the depth of v. In Fig. 2.7(a), for example, vertex 3 is of depth 2, height 0, and level 1.

An *ordered tree* is a tree in which the sons of each vertex are ordered. When drawing an ordered tree, we assume that the sons of each vertex are ordered from left to right. A *binary tree* is an ordered tree such that:

1. each son of a vertex is distinguished either as a *left son* or as a *right son*, and
2. no vertex has more than one left son nor more than one right son.

The subtree T_l (if it exists) whose root is the left son of a vertex v is called the *left subtree* of v. Similarly, the subtree T_r (if it exists) whose root is the right son of v is called the *right subtree* of v. All vertices in T_l are said to be *to the left of* all vertices in T_r.

A binary tree is usually represented by two arrays LEFTSON and RIGHTSON. Let the vertices of a binary tree be denoted by the integers



(a)

	LEFTSON	RIGHTSON
1	2	6
2	3	4
3	0	0
4	0	5
5	0	0
6	7	8
7	0	0
8	0	9
9	0	0

(b)

Fig. 2.7 A binary tree and its representation.

from 1 to n. Then LEFTSON$[i] = j$ if and only if j is the left son of i. If i has no left son, then LEFTSON$[i] = 0$. RIGHTSON$[i]$ is defined analogously.

Example 2.3. A binary tree and its representation are given in Fig. 2.7(a and (b). □

Definition. A binary tree is said to be *complete* if for some integer k every vertex of depth less than k has both a left son and a right son and every vertex of depth k is a leaf. A complete binary tree of height k has exactly $2^{k+1} - 1$ vertices.

A complete binary tree of height k is often represented by a single array. Position 1 in the array contains the root. The left son of the vertex in position i is located at position $2i$ and the right son at position $2i + 1$. Given vertex at position $i > 1$, its father is at position $\lfloor i/2 \rfloor$.

Many algorithms which make use of trees often *traverse* (visit each vertex of) the tree in some order. There are several systematic ways of doing this. Three commonly used traversals are preorder, postorder, and inorder

Definition. Let T be a tree having root r with sons $v_1, \ldots, v_k$, $k \geq 0$. In the case $k = 0$, the tree consists of the single vertex r.

A *preorder traversal* of T is defined recursively as follows:

1. Visit the root r.
2. Visit in preorder the subtrees with roots $v_1, v_2, \ldots, v_k$ in that order

A *postorder traversal* of T is defined recursively as follows:

1. Visit in postorder the subtrees with roots $v_1, v_2, \ldots, v_k$ in that order
2. Visit the root r.

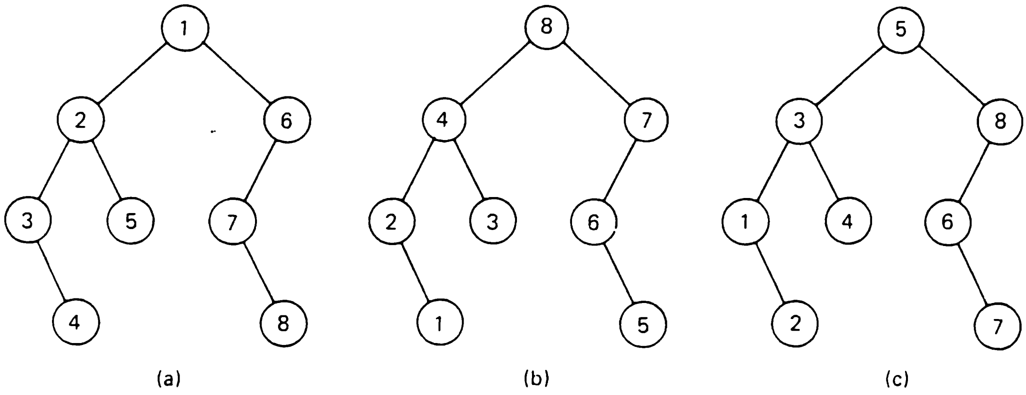


Fig. 2.8 Tree traversals: (a) preorder; (b) postorder; (c) inorder.

For a binary tree, an *inorder traversal* is defined recursively as follows:

1. Visit in inorder the left subtree of the root r (if it exists).
2. Visit r.
3. Visit in inorder the right subtree of r (if it exists).

Example 2.4. Figure 2.8 illustrates a binary tree with the vertices numbered in preorder (Fig. 2.8a), postorder (Fig. 2.8b), and inorder (Fig. 2.8c). □

Once numbers have been assigned by a traversal, it is convenient to refer to vertices by their assigned numbers. Thus v will denote the vertex which has been assigned the number v. If the vertices are numbered in the order visited, then the numberings have some interesting properties.

In preorder all vertices in a subtree with root r have numbers no less than r. More precisely, if D_r is the set of descendants of r, then v is in D_r if and only if $r \leq v < r + \|D_r\|$. By associating with each vertex v both a preorder number and the number of descendants we can easily determine whether a vertex w is a descendant of v. After initially assigning preorder numbers and calculating the number of descendants of each vertex, the question of whether w is a descendant of v can be answered in a fixed amount of time independent of tree size. Postorder numbers have an analogous property.

Inorder numbers of a binary tree have the property that each vertex in the left subtree of a vertex v has a number less than v and each vertex in the right subtree has a number greater than v. Thus to find vertex w, compare w to root r. If $w = r$, then w has been found. If $w < r$, then repeat the process for the left subtree; if $w > r$ repeat the process for the right subtree. Eventually w will be found. Such properties of traversals will be used in later chapters.

One final definition concerning trees should be made.

Definition. An *undirected tree* is an undirected graph which is *connected* (there is a path between any two vertices) and acyclic. A *rooted* undirected tree is an undirected tree in which one vertex is distinguished as the root.

A directed tree can be made into a rooted undirected tree simply by making all edges undirected. We shall use the same terminology and notational conventions for rooted undirected trees as for directed trees. The primary mathematical distinction is that in a directed tree all paths go from ancestors to descendants whereas in a rooted undirected tree paths exist in both directions.

2.5 RECURSION

A procedure that calls itself, directly or indirectly, is said to be *recursive*. The use of recursion often permits more lucid and concise descriptions of algorithms than would be possible without recursion. In this section we shall

```
procedure INORDER(VERTEX):
begin
    if LEFTSON[VERTEX] $\neq$ 0 then
        INORDER(LEFTSON[VERTEX]);
    NUMBER[VERTEX] $\leftarrow$ COUNT;
    COUNT $\leftarrow$ COUNT + 1;
    if RIGHTSON[VERTEX] $\neq$ 0 then
        INORDER(RIGHTSON[VERTEX])
end
```

Fig. 2.9. Recursive procedure for inorder.

give an example of a recursive algorithm and sketch how recursion can be implemented on a RAM.

Consider the definition of inorder traversal of a binary tree given in Section 2.4. In creating an algorithm for assigning morder numbers to the vertices of a binary tree, we would like the algorithm to reflect the definition of inorder traversal. One such algorithm is given below. Note that the algorithm calls itself recursively to number a subtree.

Algorithm 2.1. Inorder numbering of the vertices of a binary tree.

Input. A binary tree represented by arrays LEFTSON and RIGHTSON.

Output. An array called NUMBER such that NUMBER[i] is the inorder number of vertex i.

Method. In addition to LEFTSON, RIGHTSON, and NUMBER, the algorithm makes use of a global variable COUNT which contains the inorder number to be assigned to a vertex. COUNT has initial value 1. The parameter VERTEX is initially the root. The procedure of Fig. 2.9 is used recursively.

The algorithm itself is:

```
begin
    COUNT $\leftarrow$ 1;
    INORDER(ROOT)
end □
```

The use of recursion has several advantages. First, it is often easier to understand recursive programs. Had the above algorithm not been written recursively, we would have had to construct an explicit mechanism for traversing the tree. Following a path down the tree presents no problem, but the ability to return to an ancestor requires storing the ancestors on a stack, and the statements manipulating the stack would complicate the algorithm. The nonrecursive version of the same algorithm might look like the following.

Algorithm 2.2. Nonrecursive version of Algorithm 2.1.

Input. Same as Algorithm 2.1.

Output. Same as Algorithm 2.1.

Method. The tree is traversed by storing on a stack all vertices which are not yet numbered and which are on the path from the root to the vertex currently being searched. In going from vertex v to the left son of v, v is stored on the stack. After a search of the left subtree of v, v is numbered and popped from the stack. Then the right subtree of v is numbered.

In going from v to the right son of v, v is not placed on the stack, since after numbering the right subtree we do not wish to return to v; rather, we wish to return to that ancestor of v which has not yet been numbered (the closest ancestor w of v such that v is in the left subtree of w). The algorithm is shown in Fig. 2.10. □

```
          begin
              COUNT $\leftarrow$ 1;
              VERTEX $\leftarrow$ ROOT;
              STACK $\leftarrow$ empty;
left:         while LEFTSON[VERTEX] $\neq$ 0 do
                  begin
                      push VERTEX onto STACK;
                      VERTEX $\leftarrow$ LEFTSON[VERTEX]
                  end;
center:       NUMBER[VERTEX] $\leftarrow$ COUNT;
              COUNT $\leftarrow$ COUNT + 1;
              if RIGHTSON[VERTEX] $\neq$ 0 then
                  begin
                      VERTEX $\leftarrow$ RIGHTSON[VERTEX];
                      goto left
                  end;
              if STACK not empty then
                  begin
                      VERTEX $\leftarrow$ top element of STACK;
                      pop STACK;
                      goto center
                  end
          end
```

Fig. 2.10. Nonrecursive inorder algorithm.

The recursive version is easily proved correct by induction on the number of vertices in the binary tree. The nonrecursive version can similarly be proved correct, but the induction hypothesis is not as transparent, and there is the additional concern of manipulating the stack and correctly traversing the binary tree. On the other hand there may be a penalty for recursion which affects the constant factor in the time and space complexity.

The question naturally arises how recursive algorithms are to be translated into RAM code. To begin, it is sufficient, in light of Theorem 1.2, to discuss the construction of RASP code, since the RASP can be simulated by a RAM with at most a constant factor of slowdown. Here we shall discuss a rather straightforward technique for implementing recursion. This technique is adequate for all the programs that we use in this book but it does not cover all the cases that can arise.

At the heart of recursive procedure implementation is a stack in which are stored the data used by each call of a procedure which has not yet terminated. That is, all nonglobal data is on the stack. The stack is divided into *stack frames*, which are blocks of consecutive locations (registers). Each call of a procedure uses a stack frame whose length depends on the particular procedure called.

Suppose procedure A is currently in execution. The stack would look like Fig. 2.11. If A calls procedure B, we do the following:

1. A stack frame of the proper size is placed on top of the stack. Into the frame goes, in an order known to B:
   a) Pointers to the actual parameters for this call of B.†
   b) Empty space for the local variables used by B.
   c) The address of the RASP instruction in routine A which should be executed after the call to B terminates (the *return address*).‡ If B is a function that returns a value, a pointer to the location in A's stack frame in which the value of the function is to be placed (the *value address*) is also placed in the stack frame for B.
2. Control passes to the first instruction of B. The address of the value of any parameter or local identifier belonging to B is found by indexing into the stack frame for B.
3. When B terminates, it returns control to A by the following sequence of steps.
   a) The return address is obtained from the top of the stack.

† If an actual parameter is an expression, it is evaluated in the stack frame of A and a pointer to that value is placed in the frame for B. If an actual parameter is a structure such as an array, a single pointer to the first word of the structure suffices.

‡ It is the jumps to return addresses which are awkward (although possible, of course) for the RAM and which motivate us to use the RASP model here.

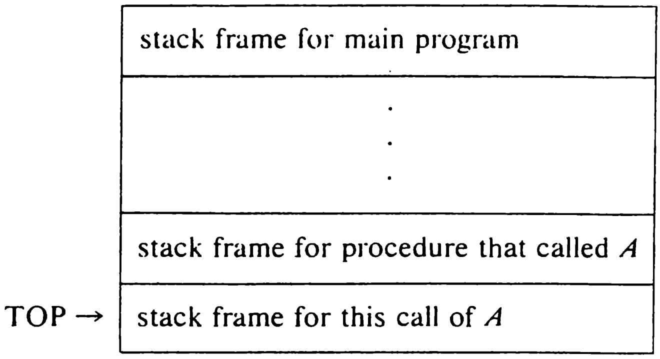

stack frame for main program
.
.
.
stack frame for procedure that called A
TOP →
stack frame for this call of A

Fig. 2.11. Stack for recursive procedure calls.

b) If B is a function, the value denoted by the expression portion of the **return** statement is stored in the location prescribed by the value address on the stack.

c) The stack frame for procedure B is popped from the stack. This leaves the frame for procedure A on top of the stack.

d) Execution of A resumes at the location given by the return address.

Example 2.5. Consider the procedure INORDER from Algorithm 2.1. When, for example, it calls itself with LEFTSON[VERTEX] as an actual parameter, it stores the address of the new value of VERTEX on the stack along with a return address to indicate that on completion of the call, execution continues with line 2. The variable VERTEX is thus effectively replaced by LEFTSON[VERTEX] wherever VERTEX occurs in the procedure definition.

In a sense, the nonrecursive version, Algorithm 2.2, is modeled on the above implementation. However, we have there recognized that the completion of a call to INORDER with actual parameter RIGHTSON[VERTEX] completes execution of the calling procedure also. Thus there is no need to store a return address or to store VERTEX on the stack in the case where the actual parameter is RIGHTSON[VERTEX]. □

The time required for a procedure call is proportional to the time required to evaluate the actual parameters and store pointers to their values on the stack. The time for a return is certainly no greater than this.

In accounting for the time spent by a collection of recursive procedures, it is usually easiest to charge the cost of a call to the procedure doing the calling. Then one can bound, as a function of input size, the time spent by a call of each procedure, exclusive of the time spent by the procedures it calls. Summing this bound over all calls of procedures gives an upper bound on the total time spent.

In order to calculate the time complexity of a recursive algorithm we make use of recurrence equations. A function $T_i(n)$ is associated with the ith procedure and denotes the execution time of the ith procedure as a function of some parameter n of the input. Usually one can express a recurrence equation for $T_i(n)$ in terms of the execution times of procedures called by procedure i. The resulting set of simultaneous recurrence equations is then solved. Often only one procedure is involved, and $T(n)$ depends on values of $T(m)$ for a finite set of m less than n. In the next section we shall study solutions for some frequently encountered recurrences.

Remember that here, and elsewhere, all cost analyses assume the uniform cost function. If we use the logarithmic cost function, the length of the stack used to implement recursive procedures may affect the time complexity analysis.

2.6 DIVIDE-AND-CONQUER

A common approach to solving a problem is to partition the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole. This approach, especially when used recursively, often yields efficient solutions to problems in which the subproblems are smaller versions of the original problem. We illustrate the technique with two examples followed by an analysis of the resulting recurrence equations.

Consider the problem of finding both the maximum and the minimum elements of a set S containing n elements. For simplicity we shall assume that n is a power of 2. One obvious way to find the maximum and minimum elements would be to find each separately. For example, the following procedure finds the maximum element of S in $n - 1$ comparisons between elements of S.

```
begin
    MAX ← any element in $S$:
    for all other elements $x$ in $S$ do
        if $x$ > MAX then MAX ← $x$
end
```

We could similarly find the minimum of the remaining $n - 1$ elements with $n - 2$ comparisons, giving a total of $2n - 3$ comparisons to find the maximum and minimum, assuming $n \geq 2$.

The divide-and-conquer approach would divide the set S into two subsets S_1 and S_2, each with $n/2$ elements. The algorithm would then find the maximum and minimum elements of each of the two halves, by recursive applications of the algorithm. The maximum and minimum elements of S

```
    procedure MAXMIN($S$):
1.  if $\|S\| = 2$ then
        begin
2.          let $S = \{a, b\}$;
3.          return (MAX($a$, $b$), MIN($a$, $b$))
        end
    else
        begin
4.          divide $S$ into two subsets $S_1$ and $S_2$, each with half the elements;
5.          (max1, min1) ← MAXMIN($S_1$);
6.          (max2, min2) ← MAXMIN($S_2$);
7.          return (MAX(max1, max2), MIN(min1, min2))
        end
```

Fig. 2.12. Procedure to find MAX and MIN.

could be calculated from the maximum and minimum elements of S_1 and S_2 by two more comparisons. The algorithm is stated more precisely below.

Algorithm 2.3. Finding the maximum and minimum elements of a set.

Input. A set S with n elements, where n is a power of 2 and $n \geq 2$.

Output. The maximum and minimum elements of S.

Method. The recursive procedure MAXMIN is applied to set S. MAXMIN has one argument† which is a set S with $\|S\| = 2^k$ for some $k \geq 1$, and it returns a pair (a, b), where a is the maximum and b the minimum element in S. Procedure MAXMIN is given in Fig. 2.12. □

Note that the only steps requiring a comparison between elements of S are step 3, where the two elements of S are compared, and step 7, where we must compare max1 with max2 and min1 with min2. Let $T(n)$ be the number of comparisons between elements of S required by MAXMIN to find the maximum and minimum elements in a set of n elements. Clearly, $T(2) = 1$. If $n > 2$, $T(n)$ is the total number of comparisons used in the two calls of MAXMIN (lines 5 and 6) on sets of size $n/2$, plus the two comparisons from line 7. That is,

$$T(n) = \begin{cases} 1, & \text{for } n = 2, \\ 2T(n/2) + 2, & \text{for } n > 2. \end{cases} \qquad (2.2)$$

† Since we are only counting comparisons here, the method of passing arguments is unimportant. However, if the set S is represented by an array, we can arrange to call MAXMIN efficiently by passing pointers to the first and last elements in a subset of S, which will be in consecutive words of the array.

The function $T(n) = \frac{3}{2}n - 2$ is a solution to recurrence (2.2). It clearly satisfies (2.2) for $n = 2$, and if it satisfies (2.2) for $n = m$, where $m \geq 2$, then

$$T(2m) = 2\left(\frac{3m}{2} - 2\right) + 2 = \tfrac{3}{2}(2m) - 2,$$

and thus it satisfies (2.2) for $n = 2m$. Thus, by induction on n, we have shown that $T(n) = \frac{3}{2}n - 2$ satisfies (2.2) whenever n is a power of 2.

We can show that at least $\frac{3}{2}n - 2$ comparisons between elements of S are necessary to find both the maximum and minimum elements in a set of n numbers. Thus Algorithm 2.3 is optimal with respect to the number of comparisons made between elements of S when n is a power of 2.

In the preceding example, the divide-and-conquer approach reduced the number of comparisons by a constant factor. In the next example we shall actually reduce the asymptotic growth rate of an algorithm by using the divide-and-conquer technique.

Consider multiplying two n-bit numbers. The traditional method requires $O(n^2)$ bit operations. The method developed below requires on the order of $n^{\log 3}$ or approximately $n^{1.59}$ bit operations.†

Let x and y be two n-bit numbers. Again for simplicity we assume that n is a power of 2. We partition x and y into two halves as shown in Fig. 2.13. If we treat each half as an $(n/2)$-bit number, then we can express the product as follows:

$$\begin{aligned} xy &= (a2^{n/2} + b)(c2^{n/2} + d) \\ &= ac2^n + (ad + bc)2^{n/2} + bd. \end{aligned} \tag{2.3}$$

Equation (2.3) computes the product of x and y by four multiplications of $(n/2)$-bit numbers plus some additions and shifts (multiplications by powers of 2). The product z of x and y can also be computed by the following program.

```
begin
    $u \leftarrow (a + b) * (c + d)$;
    $v \leftarrow a * c$;
    $w \leftarrow b * d$;                                        (2.4)
    $z \leftarrow v * 2^n + (u - v - w) * 2^{n/2} + w$
end
```

Ignore for the moment the fact that due to a carry, $a + b$ and $c + d$ may be $(n/2 + 1)$-bit numbers and assume that they have only $n/2$ bits. The scheme requires only three multiplications of $(n/2)$-bit numbers, plus some additions and shifts, to multiply two n-bit numbers. One can use the multiplication

† Recall all logarithms are to the base 2 unless otherwise stated.

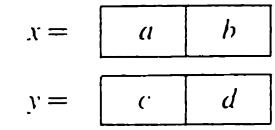


Fig. 2.13. Partition of bit strings.

routine recursively to evaluate the products u, v, and w. The additions and shifts require time $O_B(n)$. Thus the time complexity of multiplying two n-bit numbers is bounded from above by

$$T(n) = \begin{cases} k, & \text{for } n = 1, \\ 3T(n/2) + kn, & \text{for } n > 1, \end{cases} \tag{2.5}$$

where k is a constant reflecting the additions and shifts in the expressions in (2.4). The solution to recurrence (2.5) is bounded from above by

$$3kn^{\log 3} \simeq 3kn^{1.59}.$$

One can actually show that

$$T(n) = 3kn^{\log 3} - 2kn$$

in (2.5). The proof proceeds by induction on n, for n a power of 2. The basis, $n = 1$, is trivial. If $T(n) = 3kn^{\log 3} - 2kn$ satisfies (2.5) for $n = m$, then

$$\begin{aligned} T(2m) &= 3T(m) + 2km \\ &= 3[3km^{\log 3} - 2km] + 2km \\ &= 3k(2m)^{\log 3} - 2k(2m) \end{aligned}$$

for the induction step. Thus $T(n) \leq 3kn^{\log 3}$ follows. Note that attempting to use $3kn^{\log 3}$ rather than $3kn^{\log 3} - 2kn$ in the induction fails to work.

In order for the multiplication algorithm to be complete, we must take care of the fact that $a + b$ and $c + d$ are $(n/2 + 1)$-bit numbers, and thus the product $(a + b)(c + d)$ cannot be directly calculated by a recursive application of the algorithm to a problem of size $n/2$. Instead, we must write $a + b$ as $a_1 2^{n/2} + b_1$, where a_1 is the leading bit of the sum $a + b$ and b_1 is the remaining bits. Similarly, write $c + d$ as $c_1 2^{n/2} + d_1$. The product $(a + b)(c + d)$ can be expressed as

$$a_1c_1 2^n + (a_1d_1 + b_1c_1)2^{n/2} + b_1d_1. \tag{2.6}$$

The term b_1d_1 is computed by a recursive application of the multiplication algorithm on a problem of size $n/2$. The other multiplications in (2.6) can be computed in $O_B(n)$ time, since they involve either one of the single bits a_1 and c_1 or a power of 2 as an argument.

Example 2.6. This asymptotically fast integer-multiplication algorithm can be applied to decimal numbers, as well as binary. The following calculations illustrate the method.

$$\begin{array}{lrr} x = 3141 & a = 31 & c = 59 \\ y = 5927 & b = \underline{41} & d = \underline{27} \\ & a + b = 72 & c + d = 86 \end{array}$$

$$\begin{aligned} u &= (a+b)(c+d) = 72 \times 86 = 6192 \\ v &= ac = 31 \times 59 = 1829 \\ w &= bd = 41 \times 27 = 1107 \\ xy &= 18290000^{\dagger} + (6192 - 1829 - 1107) \times 100 + 1107 \\ &= 18616707 \ \square \end{aligned}$$

Note that the algorithm based on (2.4) has replaced one multiplication by three additions and subtractions [in comparison with (2.3)]. The intuitive reason why this replacement leads to asymptotic efficiency is that multiplication is harder to perform than addition, and for sufficiently large n, any fixed number of n-bit additions requires less time than an n-bit multiplication no matter what (known) algorithm we use. At first it appears that reducing the number of $(n/2)$-bit products from four to three could at best reduce the total time by 25%. However, (2.4) is applied recursively to compute $(n/2)$-bit, $(n/4)$-bit, . . . products. The 25% savings at each level is compounded and accounts for the improvement in the asymptotic time complexity.

The time complexity of a procedure is determined by the number and size of the subproblems and to a lesser extent by the amount of work necessary to divide the problem into subproblems. Since recurrences of the form of (2.2) or (2.5) arise frequently in analyzing recursive divide-and-conquer algorithms we shall consider the solution in the general case.

Theorem 2.1. Let a, b, and c be nonnegative constants. The solution to the recurrence

$$T(n) = \begin{cases} b, & \text{for } n = 1, \\ aT(n/c) + bn, & \text{for } n > 1 \end{cases}$$

for n a power of c is

$$T(n) = \begin{cases} O(n), & \text{if } a < c, \\ O(n \log n), & \text{if } a = c, \\ O(n^{\log_c a}), & \text{if } a > c. \end{cases}$$

Proof. If n is a power of c, then

$$T(n) = bn \sum_{i=0}^{\log_c n} r^i, \qquad \text{where } r = a/c.$$

† v must be shifted four decimal places and $u - v - w$ shifted two.

If $a < c$, then $\sum_{i=0}^{\infty} r^i$ converges, so $T(n)$ is $O(n)$. If $a = c$, then each term in the sum is unity, and there are $O(\log n)$ terms. Thus $T(n)$ is $O(n \log n)$. Finally, if $a > c$ then

$$bn \sum_{i=0}^{\log_c n} r^i = bn \frac{r^{1+\log_c n} - 1}{r - 1},$$

which is $O(a^{\log_c n})$ or equivalently $O(n^{\log_c a})$. □

From Theorem 2.1 we see that dividing a problem (using a linear amount of work) into two subproblems of half size results in an $O(n \log n)$ algorithm. If the number of subproblems were 3, 4, or 8, then the algorithm would be of order $n^{\log 3}$, n^2, or n^3, respectively. On the other hand, dividing the problem into four subproblems of size $n/4$ results in an $O(n \log n)$ algorithm, and 9 and 16 subproblems yield algorithms of order $n^{\log 3}$ and n^2, respectively. Thus an asymptotically faster algorithm for integer multiplication could be obtained if one divided the integers into four pieces and were able to express integer multiplication in terms of eight or fewer smaller multiplications. Other important recurrences arise when the work to divide the problem is not proportional to the size of the problem. Some of these are covered in the exercises.

In the case where n is not a power of c, one can usually embed a problem of size n in a problem of size n', where n' is the smallest power of c equal to or greater than n. Thus the asymptotic growth rates of Theorem 2.1 hold for arbitrary n. In practice, one can often design recursive algorithms which divide problems of any size as nearly into c equal parts as possible. These algorithms are usually more efficient (by a constant factor) than those obtained by pretending the input size is the next-higher power of c.

2.7 BALANCING

Both our examples of the divide-and-conquer technique partitioned a problem into subproblems of equal size. This was not a coincidence. A basic guide to good algorithm design is to maintain balance. To illustrate this principle we shall use an example from sorting and contrast the effect of dividing a problem into unequal-size subproblems as opposed to equal-size subproblems. The reader should not infer from the example that divide-and-conquer is the only technique in which balancing is useful. Chapter 4 contains several examples in which balancing sizes of subtrees or balancing the costs of two operations results in efficient algorithms.

Consider the problem of sorting a sequence of n integers into nondecreasing order. Perhaps the simplest sorting method is to locate the smallest element by scanning the sequence and then to interchange the smallest element with the first. The process is repeated on the last $n - 1$ elements, which results in the second smallest element being placed in the second position.

Repeating the process on the last $n-2, n-3, \ldots, 2$ elements sorts the sequence.

The algorithm gives rise to the recurrence

$$T(n) = \begin{cases} 0, & \text{for } n = 1, \\ T(n-1) + n - 1, & \text{for } n > 1 \end{cases} \tag{2.7}$$

for the number of comparisons made between the elements to be sorted. The solution to (2.7) is $T(n) = n(n-1)/2$, which is $O(n^2)$.

Although this sorting algorithm could be viewed as a recursive application of divide-and-conquer with division into unequal pieces, it is not efficient for large n. In order to design an asymptotically efficient sorting algorithm one should achieve a balance. Instead of dividing a problem of size n into two problems, one of size 1 and one of size $n-1$, one should divide the problem into two subproblems of approximately half the size. This is accomplished by a method known as *merge sorting.*

Consider a sequence of integers $x_1, x_2, \ldots, x_n$. Again for simplicity assume that n is a power of 2. One way to sort the sequence is to divide it into two sequences $x_1, x_2, \ldots, x_{n/2}$ and $x_{(n/2)+1}, \ldots, x_n$, sort each subsequence, and then merge them. By "merging," we mean taking the two sequences which are already sorted and combining them into one sorted sequence.

Algorithm 2.4. Mergesort.

Input. Sequence of numbers $x_1, x_2, \ldots, x_n$, where n is a power of 2.

Output. Sequence $y_1, y_2, \ldots, y_n$, a permutation of the input satisfying $y_1 \leq y_2 \leq \cdots \leq y_n$.

Method. We make use of a procedure MERGE(S, T), which takes two sorted sequences S and T as input and produces as output a sequence consisting of the elements of S and T in sorted order. Since S and T are themselves sorted, MERGE requires at most one fewer comparison than the sum of the lengths of S and T. It works by repeatedly selecting the larger of the largest elements remaining on S and T, then deleting the element selected. Ties may be broken in favor of S.

We also make use of the procedure SORT(i, j) in Fig. 2.14, which sorts the subsequence $x_i, x_{i+1}, \ldots, x_j$ on the assumption that the subsequence has length 2^k for some $k \geq 0$.

To sort the given sequence $x_1, x_2, \ldots, x_n$ we call SORT(1, n). □

In counting comparisons, Algorithm 2.4 gives rise to the recurrence

$$T(n) = \begin{cases} 0, & \text{for } n = 1, \\ 2T(n/2) + n - 1, & \text{for } n > 1 \end{cases}$$

whose solution, by Theorem 2.1, is $T(n) = O(n \log n)$. For large n, bal-

```
procedure SORT($i, j$):
if $i = j$ then return $x_i$
else
    begin
        $m \leftarrow (i+j-1)/2$;
        return MERGE(SORT($i, m$), SORT($m+1, j$))
    end
```

Fig. 2.14. Mergesort.

ancing the size of the subproblems has paid off handsomely. A similar analysis shows that the total time, not only comparisons, spent in procedure SORT is $O(n \log n)$.

2.8 DYNAMIC PROGRAMMING

Recursive techniques are useful if a problem can be divided into subproblems with reasonable effort and the sum of the sizes of the subproblems can be kept small. Recall from Theorem 2.1 that if the sum of the sizes of the subproblems is an, for some constant $a > 1$, the recursive algorithm is likely to be polynomial in time complexity. However, if the obvious division of a problem of size n results in n problems of size $n - 1$, then a recursive algorithm is likely to have exponential growth. In this case a tabular technique called *dynamic programming* often results in a more efficient algorithm.

In essence, dynamic programming calculates the solution to all subproblems. The computation proceeds from the small subproblems to the larger subproblems, storing the answers in a table. The advantage of the method lies in the fact that once a subproblem is solved, the answer is stored and never recalculated. The technique is easily understood from a simple example.

Consider the evaluation of the product of n matrices

$$M = M_1 \times M_2 \times \cdots \times M_n,$$

where each M_i is a matrix with r_{i-1} rows and r_i columns. The order in which the matrices are multiplied together can have a significant effect on the total number of operations required to evaluate M, no matter what matrix multiplication algorithm is used.

Example 2.7. Assume that the multiplication of a $p \times q$ matrix by a $q \times r$ matrix requires pqr operations, as it does in the "usual" algorithm, and consider the product

$$M = \underset{[10 \times 20]}{M_1} \times \underset{[20 \times 50]}{M_2} \times \underset{[50 \times 1]}{M_3} \times \underset{[1 \times 100]}{M_4}\ . \tag{2.8}$$

where the dimensions of each M_i are shown in the brackets. Evaluating M in the order

$$M_1 \times (M_2 \times (M_3 \times M_4))$$

requires 125,000 operations, while evaluating M in the order

$$(M_1 \times (M_2 \times M_3)) \times M_4$$

requires only 2200 operations. □

Trying all possible orderings in which to evaluate the product of n matrices so as to minimize the number of operations is an exponential process (see Exercise 2.31), which is impractical when n is large. However, dynamic programming provides an $O(n^3)$ algorithm. Let m_{ij} be the minimum cost of computing $M_i \times M_{i+1} \times \cdots \times M_j$ for $1 \leq i \leq j \leq n$. Clearly,

$$m_{ij} = \begin{cases} 0, & \text{if } i = j, \\ \underset{i \leq k < j}{\text{MIN}} \; (m_{ik} + m_{k+1,j} + r_{i-1}r_k r_j), & \text{if } j > i. \end{cases} \tag{2.9}$$

The term m_{ik} is the minimum cost of evaluating $M' = M_i \times M_{i+1} \times \cdots \times M_k$. The second term, $m_{k+1,j}$, is the minimum cost of evaluating

$$M'' = M_{k+1} \times M_{k+2} \times \cdots \times M_j.$$

The third term is the cost of multiplying M' by M''. Note that M' is an $r_{i-1} \times r_k$ matrix and M'' is an $r_k \times r_j$ matrix. Equation (2.9) states that $m_{ij}, j > i$, is the minimum, taken over all possible values of k between i and $j - 1$, of the sum of these three terms.

The dynamic programming approach calculates the m_{ij}'s in order of increasing difference in the subscripts. We begin by calculating m_{ii} for all i, then $m_{i,i+1}$ for all i, next $m_{i,i+2}$, and so on. In this way, the terms m_{ik} and $m_{k+1,j}$ in (2.9) will be available when we calculate m_{ij}. This follows since $j - i$ must be strictly greater than either of $k - i$ and $j - (k + 1)$ if k is in the range $i \leq k < j$. The algorithm is given below.

Algorithm 2.5. Dynamic programming algorithm for computing the minimum cost order of multiplying a string of n matrices, $M_1 \times M_2 \times \cdots \times M_n$.

Input. $r_0, r_1, \ldots, r_n$, where r_{i-1} and r_i are the dimensions of matrix M_i.

Output. The minimum cost of multiplying the M_i's, assuming pqr operations are required to multiply a $p \times q$ matrix by a $q \times r$ matrix.

Method. The algorithm is shown in Fig. 2.15. □

Example 2.8. Applying the algorithm to the string of four matrices in (2.8), where $r_0, \ldots, r_4$ are 10, 20, 50, 1, 100, would result in computing the values for the m_{ij}'s shown in Fig. 2.16. Thus the minimum number of operations

```
begin
    for $i \leftarrow 1$ until $n$ do $m_{ii} \leftarrow 0$;
    for $l \leftarrow 1$ until $n - 1$ do
        for $i \leftarrow 1$ until $n - l$ do
            begin
                $j \leftarrow i + l$;
                $m_{ij} \leftarrow \underset{i \leq k < j}{\text{MIN}} (m_{ik} + m_{k+1,j} + r_{i-1} * r_k * r_j)$
            end;
    write $m_{1n}$
end
```

Fig. 2.15. Dynamic programming algorithm for ordering matrix multiplications.

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10{,}000$	$m_{23} = 1000$	$m_{34} = 5000$	
$m_{13} = 1200$	$m_{24} = 3000$		
$m_{14} = 2200$			

Fig. 2.16. Costs of computing products $M_i \times M_{i+1} \times \cdots \times M_j$.

required to evaluate the product is 2200. An order in which the multiplications may be done can be determined by recording, for each table entry, a value of k which gives rise to the minimum seen in (2.9). □

2.9 EPILOGUE

This chapter has touched upon a number of fundamental techniques used in efficient algorithm design. We have seen how high-level data structures such as lists, queues, and stacks allow the algorithm designer to remove himself from such mundane chores as manipulating pointers and permit him to focus on the overall structure of the algorithm itself. We have also seen how the powerful techniques of recursion and dynamic programming often lead to elegant and natural algorithms. We also presented certain general principles such as divide-and-conquer and balancing.

These techniques are certainly not the only tools available but they are among the more important. As we progress through the remainder of this book, we shall encounter a number of other techniques. These will range from selecting an appropriate representation for a problem to performing operations in a judicious order. Perhaps the most important principle for the good algorithm designer is to refuse to be content. The designer should continue to examine a problem from a number of viewpoints until he is convinced that he has the most suitable algorithm for his needs.

EXERCISES

2.1 Select an implementation for a doubly linked list. Write Pidgin ALGOL algorithms for inserting and deleting an item. Make sure your programs work when deleting the first and/or the last item, and when the list is empty.

2.2 Write an algorithm to reverse the order of items on a list. Prove that your algorithm works correctly.

2.3 Write algorithms to implement the operations PUSH, POP, ENQUEUE, and DEQUEUE mentioned in Section 2.1. Do not forget to check whether a pointer has reached the end of the array reserved for the stack or queue.

2.4 Write the conditions for testing a queue for emptiness. Assume the array NAME used in Section 2.1 is of size k. How many elements may be stored in the queue? Draw pictures illustrating the queue and typical positions for the pointers FRONT and REAR when the queue (a) is empty, (b) contains one element, and (c) is full.

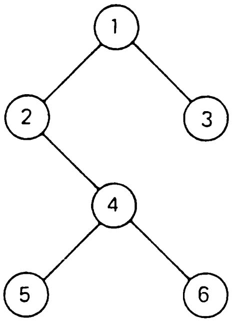
2.5 Write an algorithm to delete the first edge (v, w) on the adjacency list for v in an undirected graph. Assume that adjacency lists are doubly linked and that LINK locates v on the adjacency list of w, as described in Section 2.3.

2.6 Write an algorithm to construct the adjacency lists for an undirected graph. Each edge (v, w) is to be represented twice, once in the adjacency list of v and once in the adjacency list of w. The two copies of each edge should be linked together so that when one is deleted the other can also be deleted easily. Assume the input is a list of edges.

***2.7** (*Topological sort.*) Let $G = (V, E)$ be a directed acyclic graph. Write an algorithm to assign integers to the vertices of G such that if there is a directed edge from vertex i to vertex j, then i is less than j. [*Hint:* An acyclic graph must have a vertex with no edge coming into it. Why? One solution to the problem is to search for a vertex with no incoming edge, assign this vertex the lowest number, and delete it from the graph, along with all outgoing edges. Repeat the process on the resulting graph, assigning the next lowest number, and so on. To make the above algorithm efficient, i.e., $O(\|E\| + \|V\|)$, one must avoid searching each new graph for a vertex with no incoming edge.]

***2.8** Let $G = (V, E)$ be a directed acyclic graph with two designated vertices, the start vertex and the destination vertex. Write an algorithm to find a set of paths from the start vertex to the destination vertex such that

1) no vertex other than the start or destination vertex is common to two paths,
2) no additional path can be added to the set and still satisfy condition (1).



	LEFTSON	RIGHTSON
1	2	3
2	0	4
3	0	0
4	5	6
5	0	0
6	0	0

Fig. 2.17 A binary tree.

Note that there may be many sets of paths satisfying the above conditions. You are not required to find the set with the most paths but any set satisfying the above conditions. Your algorithm should have an $O(\|E\| + \|V\|)$ execution time.

*2.9 (*Stable marriage problem.*) Let B be a set of n boys and G be a set of n girls. Each boy ranks the girls from 1 to n and each girl ranks the boys from 1 to n. A *pairing* is a one-one correspondence of boys to girls. A pairing is *stable* if for each two boys b_1 and b_2 and their paired girls g_1 and g_2, the following two conditions are both satisfied:

1) either b_1 ranks g_1 higher than g_2, or g_2 ranks b_2 higher than b_1,
2) either b_2 ranks g_2 higher than g_1, or g_1 ranks b_1 higher than b_2.

Prove that a stable pairing always exists and write an algorithm to find one such pairing.

2.10 Consider a binary tree with names attached to the vertices. Write an algorithm to print the names in (a) preorder, (b) postorder, and (c) inorder.

*2.11 Write an algorithm to evaluate (a) prefix Polish, (b) infix, and (c) postfix Polish arithmetic expressions with operators + and ×.

*2.12 Develop a technique to initialize an entry of a matrix to zero the first time it is accessed, thereby eliminating the $O(\|V\|^2)$ time to initialize an adjacency matrix. [*Hint:* Maintain a pointer in each initialized entry to a back pointer on a stack. Each time an entry is accessed, verify that the contents are not random by making sure the pointer in that entry points to the active region on the stack and that the back pointer points to the entry.]

2.13 Simulate Algorithms 2.1 and 2.2 on the binary tree in Fig. 2.17.

2.14 Prove that Algorithm 2.2 is correct.

2.15 (*Towers of Hanoi.*) The Towers of Hanoi problem consists of three pegs A, B, and C, and n squares of varying size. Initially the squares are stacked on peg A in order of decreasing size, the largest square on the bottom. The problem is to move the squares from peg A to peg B one at a time in such a way that no square is ever placed on a smaller square. Peg C may be used for temporary storage of squares. Write a recursive algorithm to solve this problem.

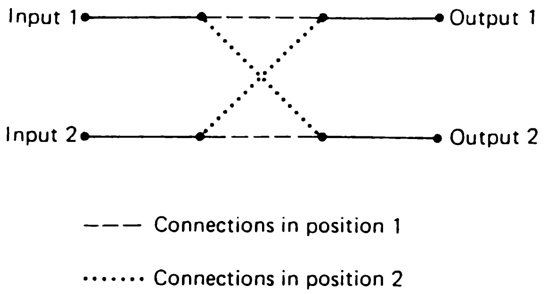

Input 1
Output 1
Input 2
Output 2
——— Connections in position 1
······· Connections in position 2

Fig. 2.18 A two-position switch.

What is the execution time of your algorithm in terms of the number of times ; square is moved?

**2.16 Solve Exercise 2.15 with a nonrecursive algorithm. Which algorithm is easie to understand and prove correct?

*2.17 Prove that $2^n - 1$ moves are both necessary and sufficient for the solution tc Exercise 2.15.

2.18 Write an algorithm to generate all permutations of the integers 1 to n. [*Hint* The set of permutations of the integers 1 to n can be obtained from the set o: permutations of the integers 1 to $n - 1$ by inserting n in each possible positior of each permutation.]

2.19 Write an algorithm to find the height of a binary tree. Assume that the tree is represented as in Fig. 2.7(b).

2.20 Write an algorithm for calculating the number of descendants of each vertex ir a tree.

**2.21 Consider a two-position switch with two inputs and two outputs, as shown in Fig. 2.18. In one position inputs 1 and 2 are connected to outputs 1 and 2, respectively. In the other position inputs 1 and 2 are connected to outputs 2 anc 1, respectively. Using these switches, design a network with n inputs and n outputs which is capable of achieving any of the $n!$ possible permutations of the inputs. Your network should use no more than $O(n \log n)$ switches. [*Hint* Make use of the divide-and-conquer approach.]

2.22 Write a RASP program to simulate the following program computing $\binom{n}{m}$.

```
procedure COMB($n$, $m$):
if $m = 0$ or $n = m$ then return 1
else return (COMB($n - 1, m$) + COMB($n - 1, m - 1$))
```

Use a stack to store current values of n and m and the return and value addresses when calls are made.

*2.23 In a number of situations a problem of size n is advantageously'divided into $\sqrt{n}$ subproblems of size about $\sqrt{n}$. Recurrence equations of the form

$$T\left(\frac{n^2}{2^r}\right) = nT(n) + bn^2$$

result, where r is an integer, $r \geq 1$. Show that the solution to the recurrence equation is $O(n(\log n)^r \operatorname{loglog} n)$.

2.24 Evaluate the sums:

a) $\sum_{i=1}^{n} i$ b) $\sum_{i=1}^{n} a^i$ c) $\sum_{i=1}^{n} ia^i$

d) $\sum_{i=1}^{k} 2^{k-i}i^2$ e) $\sum_{i=0}^{n} \binom{n}{i}$ f) $\sum_{i=1}^{n} i\binom{n}{i}$

*2.25 Solve the following recurrences, given $T(1) = 1$:
a) $T(n) = aT(n-1) + bn$ b) $T(n) = T(n/2) + bn \log n$
c) $T(n) = aT(n-1) + bn^c$ d) $T(n) = aT(n/2) + bn^c$

*2.26 Modify Algorithm 2.3 for finding the maximum and minimum elements of a set by allowing the recursion to go down to level $\|S\| = 1$. What is the asymptotic growth rate of the number of comparisons?

*2.27 Show that $\lceil \frac{3}{2}n - 2 \rceil$ comparisons are necessary and sufficient for finding both the largest and smallest elements in a set of n elements.

*2.28 Modify the integer-multiplication algorithm to divide each integer into (a) three, and (b) four pieces. What are the complexities of your algorithms?

*2.29 Let A be an array of positive or negative integers of size n, where $A[1] < A[2] < \cdots < A[n]$. Write an algorithm to find an i such that $A[i] = i$ provided such an i exists. What is the order of execution time of your algorithm? Prove that $O_C(\log n)$ is the best possible.

*2.30 If n is not a power of 2 in Algorithm 2.4, we can obtain a valid merge sorting algorithm by replacing the statement $m \leftarrow (i+j-1)/2$ in Fig. 2.14 by $m \leftarrow \lfloor (i+j)/2 \rfloor$. Let $T(n)$ be the number of comparisons to sort n elements by this method.
a) Show that

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \end{aligned}$$

b) Show that the solution to this recurrence is

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

2.31 Show that the solution to the recurrence

$$\begin{aligned} X(1) &= 1, \\ X(n) &= \sum_{i=1}^{n-1} X(i)X(n-i), \qquad \text{for } n > 1, \end{aligned}$$

is

$$X(n+1) = \frac{1}{n+1}\binom{2n}{n}.$$

$X(n)$ is the number of ways to fully parenthesize a string of n symbols. The $X(n)$'s are called the Catalan numbers. Show that $X(n) \geq 2^{n-2}$.

2.32 Modify Algorithm 2.5 to write out an order in which the matrices should be multiplied so as to minimize the number of scalar multiplications.

2.33 Write an efficient algorithm to determine an order of evaluating the matrix product $M_1 \times M_2 \times \cdots \times M_n$ so as to minimize the number of scalar multiplications in the case where each M is of dimension 1×1, $1 \times d$, $d \times 1$, or $d \times d$ for some fixed d.

Definition. A *context-free grammar* in Chomsky normal form G is a four-tuple (N, Σ, P, S) where (1) N is a finite set of *nonterminal symbols*, (2) Σ is a finite set of *terminal symbols*, (3) P is a finite set of pairs, called *productions*, of the form $A \rightarrow BC$ or $A \rightarrow a$ where A, B, C are in N and a is in Σ, and (4) S is a distinguished symbol in N. We write $\alpha A\gamma \Longrightarrow \alpha\beta\gamma$ if α, β, γ are strings of nonterminals and terminals and $A \rightarrow \beta$ is in P. $L(G)$, the *language generated by* G, is the set of terminal strings $\{w | S \overset{*}{\Longrightarrow} w\}$ where $\overset{*}{\Longrightarrow}$ is the reflexive and transitive closure of $\Longrightarrow$.

*2.34 Write an $O(n^3)$ algorithm to determine whether a given string $w = a_1a_2 \cdots a_n$ is in $L(G)$, where $G = (N, \Sigma, P, S)$ is a context-free grammar in Chomsky normal form. [*Hint:* Let $m_{ij} = \{A | A \in N \text{ and } A \overset{*}{\Longrightarrow} a_ia_{i+1} \ldots a_j\}$. $w \in L(G)$ if and only if $S \in m_{1n}$. Use dynamic programming to compute the m_{ij}'s.]

*2.35 Let x and y be strings of symbols from some alphabet. Consider the operations of deleting a symbol from x, inserting a symbol into x, and replacing a symbol of x by another symbol. Describe an algorithm to find the minimum number of such operations needed to transform x into y.

BIBLIOGRAPHIC NOTES

More information on data structures and their implementation can be found in Knuth [1968] or Stone [1972]. Pratt [1975] contains a description of recursion implementation in ALGOL-like languages. Berge [1958] and Harary [1969] discuss graph theory. Knuth [1968] is a source for trees and tree traversals. Burkhard [1973] is an additional source on tree traversal algorithms.

The optimality of Algorithm 2.3 (finding the minimum and maximum) was shown by Pohl [1972]. The $O(n^{1.59})$ integer multiplication algorithm from Section 2.6 is by Karatsuba and Ofman [1962]. Winograd [1973] considers such speed-ups from a more general point of view.

The notion of dynamic programming was popularized by Bellman [1957], and Algorithm 2.5 is a well-known application reported by Godbole [1973] and Muraoka and Kuck [1973]. The application of dynamic programming to context-free language recognition (Exercise 2.34) is the independent work of J. Cocke, Kasami [1965] and Younger [1967]. Wagner and Fischer [1974] contains a solution to Exercise 2.35.

For more information on the solution of recurrence equations, see Liu [1968] or Sloane [1973].

SORTING AND ORDER STATISTICS

CHAPTER 3

In this chapter we consider two important related problems—sorting a sequence of elements and selecting the kth smallest element in a sequence. By sorting a sequence we mean rearranging the elements in the sequence so that the elements appear in nonincreasing or nondecreasing order. We can find the kth smallest element in a sequence by sorting the sequence into nondecreasing order and selecting the kth element from the resulting sequence. However, we shall see that there are faster methods than this for selecting the kth smallest element.

Sorting is both a practically important and theoretically interesting problem. Since a significant portion of commercial data processing involves sorting large quantities of data, efficient sorting algorithms are of considerable economic importance. Even in algorithm design, the process of sorting a sequence of elements is an essential part of many algorithms.

In this chapter we consider two classes of sorting algorithms. The first class of algorithms makes use of the structure of the elements to be sorted. For example, if the elements to be sorted are integers in a fixed range 0 to $m - 1$, then we can sort a sequence of n elements in $O(n + m)$ time; if the elements to be sorted are strings over a fixed alphabet, then a sequence of strings can be sorted in time linearly proportional to the sum of the lengths of the strings.

The second class of algorithms assumes no structure on the elements to be sorted. The basic operation is a comparison between a pair of elements. With algorithms of this nature we shall see that at least $n \log n$ comparisons are needed to sort a sequence of n elements. We give two $O_C(n \log n)$ sorting algorithms—Heapsort, which is $O_C(n \log n)$ in the worst case, and Quicksort, which is $O_C(n \log n)$ in the expected case.

3.1 THE SORTING PROBLEM

Definition. A *partial order* on a set S is a relation R such that for each a, b, and c in S:

1. aRa is true (R is *reflexive*),
2. aRb and bRc imply aRc (R is *transitive*), and
3. aRb and bRa imply $a = b$ (R is *antisymmetric*).

The relation $\leq$ on integers and the relation $\subseteq$ (set inclusion) are two examples of partial orders.

A *linear order* or *total order* on a set S is a partial order R on S such that for every pair of elements a, b either aRb or bRa. The relation $\leq$ on integers is a linear order;† $\subseteq$ on sets is not.

† For a linear order $\leq$, we use $a < b$ to denote $a \leq b$ but $a \neq b$, as one would expect. Also, $b > a$ is synonymous with $a < b$ and $b \geq a$ is the same as $a \leq b$.

The *sorting problem* can be formulated as follows. We are given a sequence of n elements $a_1, a_2, \ldots, a_n$ drawn from a set having a linear order, which we shall usually denote $\leq$. We are to find a permutation π of these n elements that will map the given sequence into a nondecreasing sequence $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ such that $a_{\pi(i)} \leq a_{\pi(i+1)}$ for $1 \leq i < n$. Usually we shall produce the sorted sequence itself rather than the sorting permutation π.

Sorting methods are classified as being *internal* (where the data resides in random access memory) or *external* (where the data is predominantly outside the random access memory). External sorting is an integral part of such applications as account processing, which usually involve far more elements than can be stored in random access memory at one time. Thus external sorting methods for data which are on secondary storage devices (such as a disk memory or a magnetic tape) have great commercial importance.

Internal sorting is important in algorithm design as well as commercial applications. In those cases where sorting arises as part of another algorithm, the number of items to be sorted is usually small enough to fit in random access memory. However, we assume that the number of items to be sorted is moderately large. If one is going to sort only a handful of items, a simple strategy such as the $O(n^2)$ "bubble sort" (see Exercise 3.5) is far more expedient.

There are numerous sorting algorithms. We make no attempt to survey all the important ones; rather we limit ourselves to methods which we have found to be of use in algorithm design. We first consider the case in which the elements to be sorted are integers or (almost equivalently) strings over a finite alphabet. Here, we see that sorting can be performed in linear time. Then we consider the problem of sorting without making use of the special properties of integers or strings, in which case we are forced to make program branches depend only on comparisons between the elements to be sorted. Under these conditions we shall see that $O(n \log n)$ comparisons are necessary, as well as sufficient, to sort a sequence of n elements.

3.2 RADIX SORTING

To begin our study of integer sorting, let $a_1, a_2, \ldots, a_n$ be a sequence of integers in the range 0 to $m - 1$. If m is not too large, the sequence can easily be sorted in the following manner.

1. Initialize m empty queues, one for each integer in the range 0 to $m - 1$. Each queue is called a *bucket*.
2. Scan the sequence $a_1, a_2, \ldots, a_n$ from left to right, placing element a_i in the a_i-th queue.
3. Concatenate the queues (the contents of queue $i + 1$ are appended to the end of queue i) to obtain the sorted sequence.

Since an element can be inserted into the ith queue in constant time, the n elements can be inserted into the queues in time $O(n)$. Concatenating the m queues requires $O(m)$ time. If m is $O(n)$, then the algorithm sorts n integers in time $O(n)$. We call this algorithm a *bucket sort.*

The bucket sort can be extended to sort a sequence of tuples (i.e., lists) of integers into lexicographic order. Let $\leq$ be a linear order on set S. The relation $\leq$ when extended to tuples whose components are from S is a *lexicographic order* if $(s_1, s_2, \ldots, s_p) \leq (t_1, t_2, \ldots, t_q)$ exactly when either:

1. there exists an integer j such that $s_j < t_j$ and for all $i < j$, $s_i = t_i$, or
2. $p \leq q$ and $s_i = t_i$ for $1 \leq i \leq p$.

For example, if one treats strings of letters (under the natural alphabetic ordering) as tuples, then the words in a dictionary are in lexicographic order.

We first generalize the bucket sort to sequences consisting of k-tuples whose components are integers in the range 0 to $m - 1$. The sorting is done by making k passes over the sequence using the bucket sort on each pass. On the first pass the k-tuples are sorted according to their kth components. On the second pass the resulting sequence is sorted according to the $(k - 1)$st components. On the third pass the sequence resulting from the second pass is sorted according to the $(k - 2)$nd components, and so on. On the kth (and final) pass the sequence resulting from the $(k - 1)$st pass is sorted according to the first components.† The sequence is now in lexicographic order. A precise description of the algorithm is given below.

Algorithm 3.1. Lexicographic sort.

Input. A sequence $A_1, A_2, \ldots, A_n$ where each A_i is a k-tuple

$$(a_{i1}, a_{i2}, \ldots, a_{ik})$$

with a_{ij} an integer in the range 0 to $m - 1$. (A convenient data structure for this sequence of k-tuples is an $n \times k$ array.)

Output. A sequence $B_1, B_2, \ldots, B_n$ which is a permutation of $A_1, A_2, \ldots, A_n$ such that $B_i \leq B_{i+1}$ for $1 \leq i < n$.

Method. In transferring a k-tuple A_i to some bucket, only a pointer to A_i is actually moved. Thus A_i can be added to a bucket in fixed time rather than time bounded by k. We use a queue called QUEUE to hold the "current" sequence of elements. An array Q of m buckets is also used, where bucket $Q[i]$ is intended to hold those k-tuples that have the integer i in the component currently under consideration. The algorithm is shown in Fig. 3.1. □

† In many practical situations it is sufficient to bucket sort the strings only on the basis of their first components. If the number of elements that get placed into each bucket is small, then we can sort the strings in each bucket with some straightforward sorting algorithm such as bubble sort.

```
begin
    place $A_1, A_2, \ldots, A_n$ in QUEUE;
    for $j \leftarrow k$ step $-1$ until $1$ do
        begin
            for $l \leftarrow 0$ until $m - 1$ do make $Q[l]$ empty;
            while QUEUE not empty do
                begin
                    let $A_i$ be the first element in QUEUE;
                    move $A_i$ from QUEUE to bucket $Q[a_{ij}]$
                end;
            for $l \leftarrow 0$ until $m - 1$ do
                concatenate contents of $Q[l]$ to the end of QUEUE
        end
end
```

Fig. 3.1. Lexicographic sort algorithm.

Theorem 3.1. Algorithm 3.1 lexicographically sorts a length n sequence of k-tuples, where each component of a k-tuple is an integer between 0 and $m - 1$, in time $O((m + n)k)$.

Proof. The proof that Algorithm 3.1 works correctly is by induction on the number of executions of the outer loop. The induction hypothesis is that after r executions, the k-tuples in QUEUE will be lexicographically sorted according to their r rightmost components. The result is easily established once it is observed that the $(r + 1)$st execution sorts k-tuples by their $(r + 1)$st component from the right, and that if two k-tuples are placed in the same bucket, the first k-tuple precedes the second in the lexicographic order determined by the r rightmost components.

One pass of the outer loop of Algorithm 3.1 requires $O(m + n)$ time. The loop is repeated k times to give a time complexity of $O((m + n)k)$. □

Algorithm 3.1 has a variety of applications. It has been used in punched card sorting machines for a long time. It can also be used to sort $O(n)$ integers in the range 0 to $n^k - 1$ in time $O(kn)$, since such an integer can be thought of as a k-tuple of digits in the range 0 to $n - 1$ (i.e., the representation of the integer in base n notation).

Our final generalization of the bucket sort will be to tuples of varying sizes, which we shall call strings. If the longest string is of length k, we could pad out every string with a special symbol to make all strings be of length k and then use Algorithm 3.1. However, if there are only a few long strings, then this approach is unnecessarily inefficient for two reasons. First, on each pass, every string is examined, and second, every bucket $Q[i]$ is examined even if almost all of these buckets are empty. We shall describe an algorithm that sorts a sequence of n strings of varying length, whose components are in the range 0 to $m-1$, in time $O(m+l_{\text{total}})$, where l_i is the length of the ith string, and $l_{\text{total}}=\sum_{i=1}^{n} l_i$. The algorithm is useful in the situation where m and l_{total} are both $O(n)$.

The essence of the algorithm is to first arrange the strings in order of decreasing length. Let $l_{\max}$ be the length of the longest string. Then, as in Algorithm 3.1, $l_{\max}$ passes of the bucket sort are used. However, the first pass sorts (by rightmost component) only those strings of length $l_{\max}$. The second pass sorts [according to the $(l_{\max}-1)$st component] those strings of length at least $l_{\max}-1$, and so on.

For example, suppose *bab, abc,* and *a* are three strings to be sorted. (We have assumed that the components of tuples are integers, but for notational convenience we shall often use letters instead. This should cause no difficulty because we can always substitute 0, 1, and 2 for a, b, and c, if we like.) Here $l_{\max}=3$, so on the first pass we would sort only the first two strings on the basis of their third components. In sorting these two strings we would put *bab* into the *b*-bucket and *abc* into the *c*-bucket. The *a*-bucket would remain empty. In the second pass we would sort these same two strings on the second component. Now the *a*-bucket and *b*-bucket would be occupied, but the *c*-bucket would remain empty. On the third and final pass we would sort all three strings on the basis of their first component. This time the *a*- and *b*-buckets would be occupied and the *c*-bucket would be empty.

We can see that in general on a given pass many buckets can be empty. Thus a preprocessing step that determines which buckets will be nonempty on a given pass is beneficial. The list of nonempty buckets for each pass is determined in increasing order of bucket number. This allows us to concatenate the nonempty buckets in time proportional to the number of nonempty buckets.

Algorithm 3.2. Lexicographic sort of strings of varying length.

Input. A sequence of strings (tuples), $A_1, A_2, \ldots, A_n$, whose components are integers in the range 0 to $m-1$. Let l_i be the length of $A_i=(a_{i1}, a_{i2}, \ldots, a_{il_i})$, and let $l_{\max}$ be the largest of the l_i's.

Output. A permutation $B_1, B_2, \ldots, B_n$ of the A_i's such that

$$B_1 \le B_2 \le \cdots \le B_n.$$

Method

1. We begin by making lists, one for each l, $1 \le l \le l_{\max}$, of those symbols that appear in the lth component of one or more of the strings. We do so by first creating, for each component a_{il}, $1 \le i \le n$, $1 \le l \le l_i$ of each string A_i, a pair (l, a_{il}). Such a pair indicates that the lth component of some string contains the integer a_{il}. These pairs are then sorted lex-

```
         begin
1.           make QUEUE empty;
2.           for $j \leftarrow 0$ until $m-1$ do make $Q[j]$ empty;
3.           for $l \leftarrow l_{\max}$ step $-1$ until 1 do
                 begin
4.                   concatenate LENGTH[$l$] to the beginning of
                       QUEUE;†
5.                   while QUEUE not empty do
                         begin
6.                           let $A_i$ be the first string on QUEUE;
7.                           move $A_i$ from QUEUE to bucket $Q[a_{il}]$
                         end;
8.                   for each $j$ on NONEMPTY[$l$] do
                         begin
9.                           concatenate $Q[j]$ to the end of QUEUE;
10.                          make $Q[j]$ empty
                         end
                 end
         end
```

† Technically, we should only concatenate to the end of a queue, but concatenation to the beginning should present no conceptual difficulty. The most efficient approach here would be to select A_i's from LENGTH[l] first at line 6, then select them from QUEUE, without ever concatenating LENGTH[l] and QUEUE at all.

Fig. 3.2. Lexicographic sort of strings of varying length.

icographically by an obvious generalization of Algorithm 3.1.‡ Then, by scanning the sorted list from left to right it is easy to create $l_{\max}$ sorted lists NONEMPTY[l], for $1 \leq l \leq l_{\max}$, such that NONEMPTY[l] contains exactly those symbols that appear in the lth component of some string. That is, NONEMPTY[l] contains, in sorted order, all those integers j such that $a_{il} = j$ for some i.

2. We determine the length of each string. We then make lists LENGTH[l], for $1 \leq l \leq l_{\max}$, where LENGTH[l] consists of all strings of length l. (Although we speak of moving a string, we are only moving a pointer to a string. Thus each string can be added to LENGTH[l] in a fixed number of steps.)
3. We now sort the strings by components as in Algorithm 3.1, beginning with the components in position $l_{\max}$. However, after the ith pass

‡ In Algorithm 3.1, the assumption was made that components were chosen from the same alphabet. Here, the second component ranges from 0 to $m-1$, while the first ranges from 1 to $l_{\max}$.

QUEUE contains only those strings of length $l_{\max} - i + 1$ or greater, and these strings will already be sorted according to components $l_{\max} - i + 1$ through $l_{\max}$. The NONEMPTY lists computed in step 1 are used to determine which buckets are occupied at each pass of the bucket sort. This information is used to help speed up the concatenation of the buckets. This part of the algorithm is given in Pidgin ALGOL in Fig. 3.2 on p. 81. □

Example 3.1. Let us sort the strings *a*, *bab*, and *abc* using Algorithm 3.2. One possible representation for these strings is the data structure shown in Fig. 3.3. STRING is an array such that STRING[i] is a pointer to the representation of the ith string whose length and components are stored in the array DATA. The cell in DATA pointed to by STRING[i] gives the number j of symbols in the ith string. The next j cells of DATA contain these symbols.

The lists of strings used by Algorithm 3.2 are really lists of pointers such as those in the array STRING. For notational convenience, in the remainder of this example we shall write the actual strings, rather than the pointers to the strings, on lists. Bear in mind, however, that it is the pointers rather than the strings themselves that are being stored in the queues.

In part 1 of Algorithm 3.2 we create the pair $(1, a)$ from the first string, the pairs $(1, b)$, $(2, a)$, $(3, b)$ from the second, and the pairs $(1, a)$, $(2, b)$, $(3, c)$ from the third. The sorted list of these pairs is:

$$(1, a)\ (1, a)\ (1, b)\ (2, a)\ (2, b)\ (3, b)\ (3, c).$$

By scanning this sorted list from left to right we deduce that

$$\begin{aligned}\text{NONEMPTY}[1] &= a, b\\ \text{NONEMPTY}[2] &= a, b\\ \text{NONEMPTY}[3] &= b, c\end{aligned}$$

In part 2 of Algorithm 3.2 we compute $l_1 = 1$, $l_2 = 3$, and $l_3 = 3$. Thus LENGTH[1] = *a*, LENGTH[2] is empty, and LENGTH[3] = *bab*, *abc*. We therefore begin part 3 by setting QUEUE = *bab*, *abc*, and sorting these strings by their third component. The fact that NONEMPTY[3] = *b*, *c* assures us that when we form the sorted list in lines 8–10 of Fig. 3.2, $Q[a]$ need not be concatenated to the end of QUEUE. We thus have QUEUE = *bab*, *abc* after the first pass of the loop of lines 3–10 in Fig. 3.2.

In the second pass, QUEUE does not change, since LENGTH[2] is empty and the sort by second component does not change the order. In the third pass, we set QUEUE to *a*, *bab*, *abc* at line 4. The sort by first components gives QUEUE = *a*, *abc*, *bab*, which is the correct order. Note that in the third pass, $Q[c]$ remains empty, and since c is not on NONEMPTY[1], we do not concatenate $Q[c]$ to the end of QUEUE. □

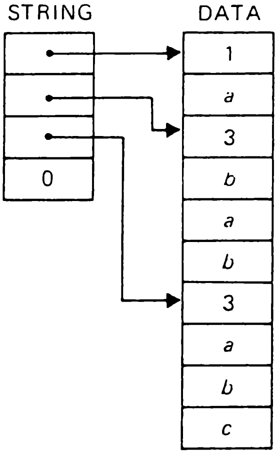


Fig. 3.3 Data structure for strings.

Theorem 3.2. Algorithm 3.2 sorts its input in time $O(l_{\text{total}} + m)$, where

$$l_{\text{total}} = \sum_{i=1}^{n} l_i.$$

Proof. An easy induction on the number of passes through the outer loop in Fig. 3.2 proves that after i passes, QUEUE contains those strings of length $l_{\max} - i + 1$ or greater, and that they are sorted according to components $l_{\max} - i + 1$ through $l_{\max}$. Thus the algorithm lexicographically sorts its input.

For the timing result, part 1 of the algorithm uses $O(l_{\text{total}})$ time to create the pairs and $O(m + l_{\text{total}})$ time to sort them. Similarly, part 2 requires no more than $O(l_{\text{total}})$ time.

We must now direct our attention to part 3 and the program of Fig. 3.2. Let n_i be the number of strings having an ith component. Let m_i be the number of different symbols appearing in the ith components of the strings (i.e. m_i is the length of NONEMPTY[i]).

Consider a fixed value of l in line 3 of Fig. 3.2. The loop of lines 5–7 requires $O(n_l)$ time and the loop of lines 8–10 requires $O(m_l)$ time. Step 4 requires constant time, so one pass through the loop of lines 3–10 requires $O(m_l + n_l)$ time. Thus the entire loop takes

$$O\left(\sum_{l=1}^{l_{\max}} (m_l + n_l)\right)$$

time. Since

$$\sum_{l=1}^{l_{\max}} m_l \leq l_{\text{total}} \quad \text{and} \quad \sum_{l=1}^{l_{\max}} n_l = l_{\text{total}}.$$

we see that lines 3–10 require time $O(l_{\text{total}})$. Then, as line 1 requires constant time and line 2 requires $O(m)$ time, we have the desired result. □

We offer one example where string sorting arises in the design of an algorithm.

Example 3.2. Two trees are said to be *isomorphic* if we can map one tree into the other by permuting the order of the sons of vertices. Consider the problem of determining whether two trees T_1 and T_2 are isomorphic. The following algorithm works in time linearly proportional to the number of vertices. The algorithm assigns integers to the vertices of the two trees, starting with vertices at level 0 and working up towards the roots, in such a way that the trees are isomorphic if and only if their roots are assigned the same integer. The algorithm proceeds as follows.

1. Assign to all leaves of T_1 and T_2 the integer 0.
2. Inductively, assume that all vertices of T_1 and T_2 at level $i-1$ have been assigned integers. Assume L_1 is a list of the vertices of T_1 at level $i-1$ sorted by nondecreasing value of the assigned integers. Assume L_2 is the corresponding list for T_2.†
3. Assign to the nonleaves of T_1 at level i a tuple of integers, by scanning the list L_1 from left to right and performing the following actions: For each vertex v on list L_1 take the integer assigned to v to be the next component of the tuple associated with the father of v. On completion of this step, each nonleaf w of T_1 at level i will have a tuple $(i_1, i_2, \ldots, i_k)$ associated with it, where $i_1, i_2, \ldots, i_k$ are the integers, in nondecreasing order, associated with the sons of w. Let S_1 be the sequence of tuples created for the vertices of T_1 on level i.
4. Repeat step 3 for T_2 and let S_2 be the sequence of tuples created for the vertices of T_2 on level i.
5. Sort S_1 and S_2 using Algorithm 3.2. Let S_1' and S_2', respectively, be the sorted sequences of tuples.
6. If S_1' and S_2' are not identical, then halt; the trees are not isomorphic. Otherwise, assign the integer 1 to those vertices of T_1 on level i represented by the first distinct tuple on S_1', assign the integer 2 to the vertices represented by the second distinct tuple, and so on. As these integers are assigned to the vertices of T_1 on level i, make a list L_1 of the vertices so assigned. Append to the front of L_1 all leaves of T_1 on level i. Let L_2 be the corresponding list of vertices of T_2. These two lists can now be used for the assignment of tuples to vertices at level $i+1$ by returning to step 3.

† You should convince yourself that level numbers can be assigned in $O(n)$ steps by a preorder traversal of the tree.

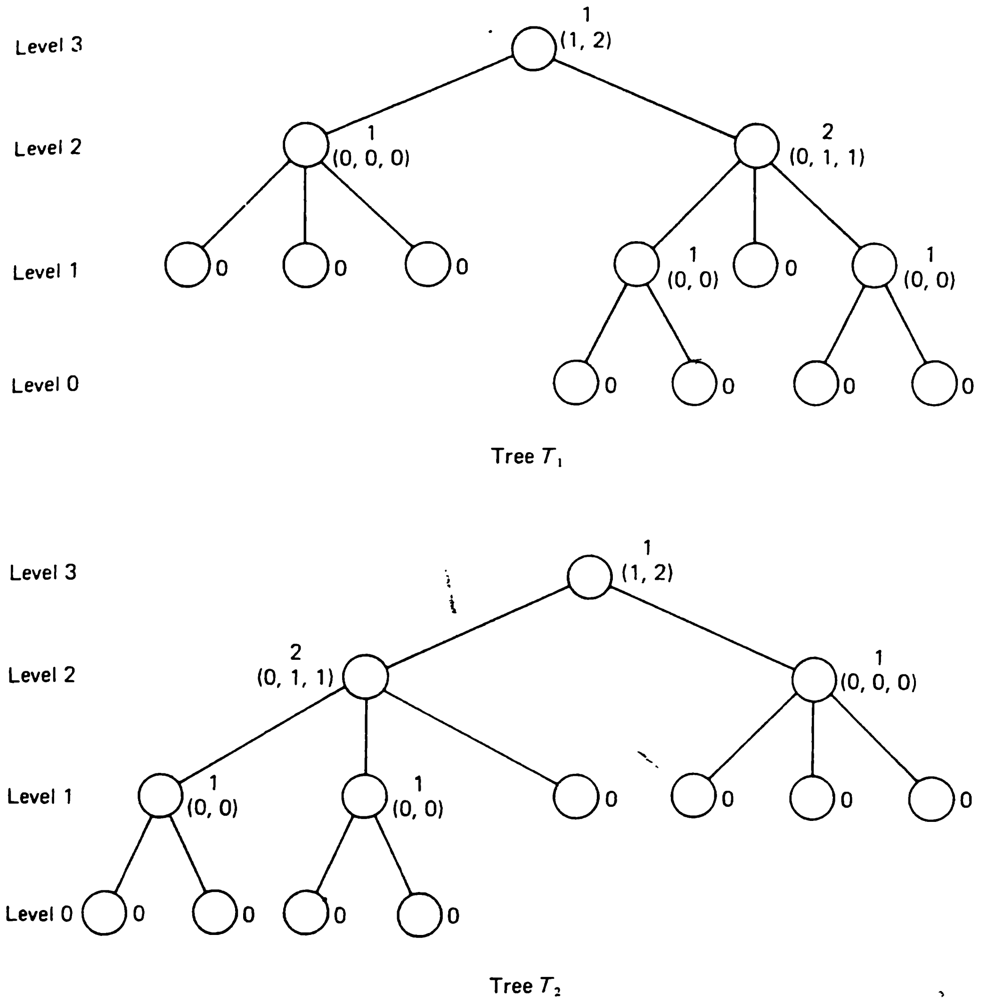


Fig. 3.4 Numbers assigned by tree isomorphism algorithm.

7. If the roots of T_1 and T_2 are assigned the same integer, T_1 and T_2 are isomorphic. □

Figure 3.4 illustrates the assignment of integers and tuples to the vertices of two isomorphic trees.

Theorem 3.3. We can determine whether two n-vertex trees are isomorphic in $O(n)$ time.

Proof. The theorem follows from a formalization of the algorithm of Example 3.2. The proof of correctness of the algorithm will be omitted. The analysis of the running time is obtained by observing that the work of assigning integers to vertices at level i, other than leaves, is proportional to the number of vertices at level $i-1$. Summing over all levels results in $O(n)$ time. The work in assigning integers to leaves is also proportional to n and thus the algorithm takes $O(n)$ time. □

A *labeled tree* is a tree in which labels are attached to the vertices. Suppose the vertex labels are integers in the range 1 to n. Then we can determine whether two labeled trees with n vertices are isomorphic in linear time if we include the label of each vertex as the first component of the tuple assigned to that vertex in the algorithm above. Thus we have the following corollary.

Corollary. Determining the isomorphism of two n-vertex labeled trees with labels in the range 1 to n takes $O(n)$ time.

3.3 SORTING BY COMPARISONS

In this section we consider the problem of sorting a sequence of n elements drawn from a linearly ordered set S whose elements have no known structure. The only operation that can be used to gain information about the sequence is the comparison of two elements. We first show that any algorithm which sorts by comparisons must on some sequence of length n use at least $O(n \log n)$ comparisons.

Assume that there are n distinct elements $a_1, a_2, \ldots, a_n$ which are to be sorted. An algorithm that sorts by comparisons can be represented by a decision tree as described in Section 1.5. Figure 1.18 (p. 25) showed a decision tree that sorts the sequence a, b, c. In what follows, we assume that if element a is compared with element b at some vertex v of a decision tree, then we branch to the left son of v if $a < b$, and to the right son if $a \geq b$.

Normally, sorting algorithms that use comparisons for branching restrict themselves tc comparing two input elements at a time. In fact, an algorithm that works for an arbitrary linearly ordered set may not combine input data in any way, since operations on data do not "make sense" in the completely general setting. In any event, we can prove a strong result about the height of any decision tree which sorts n elements.

Lemma 3.1. A binary tree of height h has at most 2^h leaves.

Proof. An elementary induction on h. One need only observe that a binary tree of height h is composed of a root and at most two subtrees, each of height at most $h-1$. □

Theorem 3.4. Any decision tree that sorts n distinct elements has height at least $\log n!$

Proof. Since the result of sorting n elements can be any one of the $n!$ permutations of the input, there must be at least $n!$ leaves in the decision tree. By Lemma 3.1, the height must be at least $\log n!$. □

Corollary. Any algorithm for sorting by comparisons requires at least $cn \log n$ comparisons to sort n elements for some $c > 0$ and sufficiently large n.

Proof. We note that for $n > 1$

$$n! \geq n(n-1)(n-2)\cdots\left(\left\lceil\frac{n}{2}\right\rceil\right) \geq \left(\frac{n}{2}\right)^{n/2},$$

so $\log n! \geq (n/2)\log(n/2) \geq (n/4)\log n$ for $n \geq 4$. □

From Stirling's approximation a more accurate estimate of $n!$ is $(n/e)^n$, so $n(\log n - \log e) = n \log n - 1.44n$ is a good approximate lower bound on the number of comparisons needed to sort n elements.

3.4 HEAPSORT—AN $O(n \log n)$ COMPARISON SORT

Since every sorting algorithm which sorts by comparisons requires essentially $n \log n$ comparisons to sort at least one sequence of length n, it is natural to ask whether there exist sorting algorithms that use only $O(n \log n)$ comparisons to sort all sequences of length n. In fact, we have already seen one such algorithm, the merge sort of Section 2.7. Another such algorithm is Heapsort. In addition to being a useful sorting algorithm, Heapsort uses an interesting data structure which has other applications.

Heapsort is best understood in terms of a binary tree as in Fig. 3.5 in which every leaf is of depth d or $d-1$. We label the vertices of the tree with the elements of the sequence to be sorted. Heapsort then rearranges the elements on the tree until the element associated with each vertex is greater than or equal to the elements associated with its sons. Such a labeled tree is called a *heap*.

Example 3.3. Figure 3.5 illustrates a heap. Note that the sequence of elements on the path from each leaf to the root is linearly ordered and that the largest element in a subtree is always at the root of that subtree. □

The next step in Heapsort is to remove from the heap the largest element, which is at the root. The label of some leaf is moved to the root and that leaf is deleted. The tree is then remade into a heap and the process is repeated. The sequence of elements removed from the heap is the sorted sequence (in descending order).

A convenient data structure for a heap is an array A, where $A[1]$ is the element stored at the root, and $A[2i]$ and $A[2i+1]$ are the elements stored at

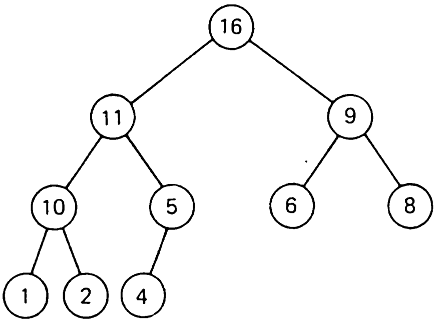


Fig. 3.5 A heap.

the left and right sons (if they exist) of the vertex at which element $A[i]$ is stored. For example, the heap in Fig. 3.5 would be represented by the following array.

16	11	9	10	5	6	8	1	2	4

Observe that the vertices of smallest depth appear first in the array, and vertices of equal depth appear in left-to-right order.

Not every heap can be so represented. In terms of the tree representation, the leaves at the lowest level must be as far left as possible, as in Fig. 3.5.

If we use an array to represent the heap, several operations in the Heapsort algorithm are easy to perform. For example, in the algorithm we must remove the element at the root, store this element somewhere, then remake the remaining tree into a heap, and remove the unlabeled leaf. We can remove the largest element from the heap and store it by interchanging $A[1]$ and $A[n]$, and then considering location n of the array no longer part of the heap. We treat location n as the leaf deleted from the tree. To remake the tree in locations $1, 2, \ldots, n-1$ into a heap, we take the new element $A[1]$ and percolate it as far down a path in the tree as necessary. We can then repeat this process interchanging $A[1]$ and $A[n-1]$ and considering the tree to occupy locations $1, 2, \ldots, n-2$, and so on.

Example 3.4. Consider, in terms of the heap of Fig. 3.5, what happens when we interchange the first and last elements in the array representing the heap. The resulting array corresponds to the labeled tree in Fig. 3.6(a).

4	11	9	10	5	6	8	1	2	16

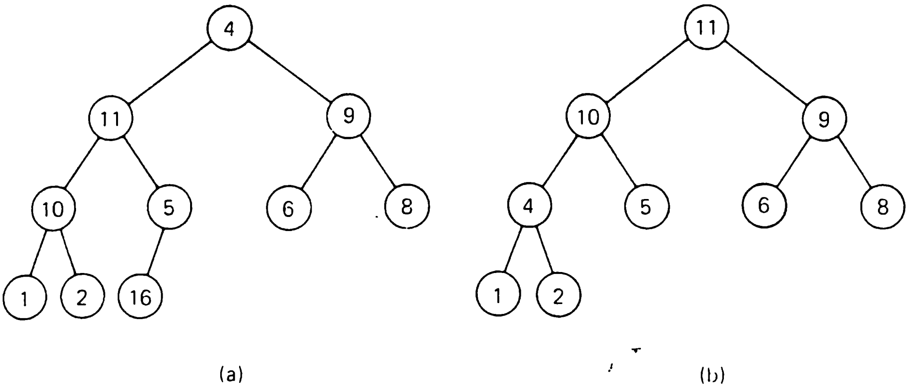


Fig. 3.6 (a) Result of interchanging 4 and 16 in the heap of Fig. 3.5; (b) result after reconstructing heap and excluding element 16.

We exclude element 16 from further consideration. To convert the resulting tree into a heap, we exchange element 4 with 11, the larger of its two sons 11 and 9.

In its new position, 4 has sons 10 and 5. Since these are larger than 4, we interchange 4 with 10, the larger son. Then the sons of 4 in its new position are 1 and 2. Since 4 exceeds each of these, no further interchanges are needed. The resulting heap is shown in Fig. 3.6(b). Note that although element 16 has been removed from the heap it is still present at the end of the array A. □

We now begin a formal description of the Heapsort algorithm. Let $a_1, a_2, \ldots, a_n$ be the sequence of elements to be sorted. Assume that these elements are initially in the array A in this order, i.e., $A[i] = a_i$, $1 \leq i \leq n$. The first step is to build the heap. That is, the elements in A are rearranged to satisfy the *heap property:* $A[i] \geq A[2i]$ for $1 \leq i \leq n/2$, and $A[i] \geq A[2i+1]$ for $1 \leq i < n/2$. This is done by starting with the leaves and building larger and larger heaps. Each subtree consisting of a leaf already forms a heap. A subtree of height h is made into a heap by interchanging the element at the root with the larger of the elements at the sons of the root, if it is smaller than either of them. So doing may destroy a heap of height $h - 1$, which must then be remade into a heap. The algorithm is made precise below.

Algorithm 3.3. Construction of a heap.

Input. Array of elements $A[i]$, $1 \leq i \leq n$.

Output. Elements of A arranged into a heap such that $A[i] \leq A[\lfloor i/2 \rfloor]$ for $1 < i \leq n$.

Method. The heart of the algorithm is the recursive procedure HEAPIFY. The parameters i and j give the range of locations in the array A having the heap property, and i is the root of the heap to be made.

```
         procedure HEAPIFY($i, j$):
1.       if $i$ is not a leaf and if a son of $i$ contains a larger element than $i$
           does then
             begin
2.               let $k$ be a son of $i$ with the largest element;
3.               interchange $A[i]$ and $A[k]$;
4.               HEAPIFY($k, j$)
             end
```

The parameter j is used to determine whether i is a leaf and whether i has one or two sons. If $i > j/2$, then i is a leaf and HEAPIFY(i, j) need not do anything, since $A[i]$ is a heap by itself.

The algorithm to give all of A the heap property is simply:

```
procedure BUILDHEAP:
for $i \leftarrow n$[†] step $-1$ until 1 do HEAPIFY($i, n$) □
```

We next show that Algorithm 3.3 makes a heap out of A in linear time.

Lemma 3.2. If the vertices $i+1, i+2, \ldots, n$ are the roots of heaps, then after a call to HEAPIFY(i, n), all of $i, i+1, \ldots, n$ will be the roots of heaps.

Proof. The proof proceeds by backwards induction on i. The basis, $i = n$, is trivial, since vertex n must be a leaf, and the test of line 1 assures that HEAPIFY(n, n) does nothing.

For the inductive step, note that if vertex i is a leaf or if i has no son with a larger element, then there is nothing to prove, by the above argument. However, if vertex i has one son (i.e., if $2i = n$) and if $A[i] < A[2i]$, then line 3 of HEAPIFY interchanges $A[i]$ and $A[2i]$. At line 4, HEAPIFY($2i, n$) is called, so the inductive hypothesis implies that the tree with root at vertex $2i$ is remade into a heap. Vertices $i+1, i+2, \ldots, 2i-1$ never cease being the roots of heaps. Since in the new permutation of array A we have $A[i] > A[2i]$, the tree with root i is likewise a heap.

Similarly, if vertex i has two sons (i.e., $2i + 1 \leq n$) and if the larger of $A[2i]$ and $A[2i+1]$ is larger than $A[i]$, we can argue as above to show that after the call of HEAPIFY(i, n), all of $i, i+1, \ldots, n$ will be roots of heaps. □

Theorem 3.5. Algorithm 3.3 makes A into a heap in linear time.

[†] In practice, we would begin at $\lfloor n/2 \rfloor$.

Proof. Using Lemma 3.2, we may show by an easy backwards induction on i that vertex i becomes the root of a heap for all i, $1 \le i \le n$.

Let $T(h)$ be the time required to execute HEAPIFY on a vertex of height h. Then $T(h) \le T(h-1) + c$ for some constant c, which implies that $T(h)$ is $O(h)$.

Algorithm 3.3 calls HEAPIFY, exclusive of recursive calls, once for each vertex. Thus the time spent by BUILDHEAP is on the order of the sum, over all vertices, of the heights of the vertices. But at most $\lceil n/2^{i+1} \rceil$ vertices are of height i. Therefore, the total time spent by BUILDHEAP is on the order of

$$\sum_{i=1}^{h} i \frac{n}{2^i},$$

which is $O(n)$. □

We can now complete the specification of Heapsort. Once the elements of A have been arranged into a heap, elements are removed one at a time from the root. This is done by interchanging $A[1]$ and $A[n]$ and rearranging $A[1], A[2], \ldots, A[n-1]$ into a heap. Next $A[1]$ and $A[n-1]$ are interchanged and $A[1], A[2], \ldots, A[n-2]$ are rearranged into a heap, and so on until the heap consists of one element. At that point $A[1], A[2], \ldots, A[n]$ is the sorted sequence.

Algorithm 3.4. Heapsort.

Input. Array of elements $A[i]$, $1 \le i \le n$.

Output. Elements of A sorted into nondecreasing order.

Method. We make use of the procedure BUILDHEAP, which is Algorithm 3.3. The algorithm is as follows.

```
begin
    BUILDHEAP;
    for $i \leftarrow n$ step $-1$ until 2 do
        begin
            interchange $A[1]$ and $A[i]$;
            HEAPIFY($1, i-1$)
        end
end □
```

Theorem 3.6. Algorithm 3.4 sorts n elements in time $O(n \log n)$.

Proof. The proof of correctness is by induction on the number of times m that the main loop has been executed. The induction hypothesis is that after m iterations $A[n-m+1], \ldots, A[n]$ contains the m largest elements in sorted (smallest first) order, and $A[1], \ldots, A[n-m]$ forms a heap. The details are left for an exercise. The time to execute HEAPIFY(1, i) is $O(\log i)$. Hence Algorithm 3.4 is of complexity $O(n \log n)$. □

Corollary. Heapsort is $O_C(n \log n)$ in time complexity.

3.5 QUICKSORT—AN $O(n \log n)$ EXPECTED TIME SORT

So far we have considered only the worst-case behavior of sorting algorithms. For many applications a more realistic measure of the time complexity of an algorithm is its expected running time. When we consider sorting, we find that no comparison sorting algorithm can have an expected time complexity significantly lower than $n \log n$ under the decision tree model. However, we do find that there are sorting algorithms whose worst-case running times are cn^2 for some constant c but whose expected running times are among the best of known sorting algorithms. Quicksort, the algorithm to be discussed in this section, is an example of such an algorithm.

Before we can talk about the expected running time of an algorithm, we must agree on what the probability distribution of the inputs is. For sorting, a natural assumption, and the one we shall make, is that every permutation of the sequence to be sorted is equally likely to appear as an input. Under this assumption, we can readily bound from below the expected number of comparisons needed to sort a sequence of n elements.

The general method is to associate with each leaf v of a decision tree the probability that v will be reached on a given input. If we know the probability distribution of the inputs, the probabilities associated with the leaves can be determined. Thus we can calculate the expected number of comparisons made by a particular sorting algorithm by evaluating, over all leaves of the decision tree for that algorithm, the sum $\Sigma_i p_i d_i$, where p_i is the probability of reaching the ith leaf and d_i is its depth. This figure is called the *expected depth* of the decision tree. We are thus led to the following generalization of Theorem 3.4.

Theorem 3.7. On the assumption that all permutations of a sequence of n elements are equally likely to appear as input, any decision tree that sorts n elements has an expected depth of at least $\log n!$.

Proof. Let $D(T)$ be the sum of the depths of the leaves of a binary tree T. Let $D(m)$ be the smallest value of $D(T)$ taken over all binary trees T with m leaves. We shall show, by induction on m, that $D(m) \geq m \log m$.

The basis, $m = 1$, is trivial. Now, assume the inductive hypothesis is true for all values of m less than k. Consider a decision tree T with k leaves. T consists of a root having a left subtree T_i with i leaves and a right subtree T_{k-i} with $k - i$ leaves for some i, $1 \leq i < k$. Clearly,

$$D(T) = i + D(T_i) + (k - i) + D(T_{k-i}).$$

Therefore, the minimum sum is given by

$$D(k) = \min_{1 \le i \le k} [k + D(i) + D(k-i)]. \tag{3.1}$$

Invoking the inductive hypothesis, we obtain from (3.1)

$$D(k) \ge k + \min_{1 \le i \le k} [i \log i + (k-i) \log (k-i)]. \tag{3.2}$$

It is easy to show that the minimum occurs at $i = k/2$. Thus

$$D(k) \ge k + k \log \frac{k}{2} = k \log k.$$

We conclude that $D(m) \ge m \log m$ for all $m \ge 1$.

Now we claim that a decision tree T sorting n random elements has at least $n!$ leaves. Moreover, exactly $n!$ leaves will have probability $1/n!$ each, and.the remaining leaves will have probability zero. We may remove from T all vertices that are ancestors only of leaves of probability zero, without changing the expected depth of T. We are thus left with a tree T' having $n!$ leaves each of probability $1/n!$. Since $D(T') \ge n! \log n!$, the expected depth of T' (and hence of T) is at least $(1/n!)\ n! \log n! = \log n!$. □

Corollary. Every comparison sort makes at least $cn \log n$ comparisons on average for some constant $c > 0$.

There is an efficient algorithm, called Quicksort, which is worth mentioning because its expected running time, while bounded below by $cn \log n$ for some constant c, as any comparison sort must be, is a fraction of the running time of other known algorithms when implemented on most real machines. The fact that Quicksort has a worst-case running time which is quadratic is not important in many applications.

Algorithm 3.5. Quicksort.

Input. Sequence S of n elements, $a_1, a_2, \ldots, a_n$.

Output. The elements of S in sorted order.

Method. We define the recursive procedure QUICKSORT in Fig. 3.7. The algorithm consists of a call to QUICKSORT(S). □

Theorem 3.8. Algorithm 3.5 sorts a sequence of n elements in $O(n \log n)$ expected time.

Proof. The correctness of Algorithm 3.5 follows by a straightforward induction on the size of S. For simplicity in the timing analysis assume that all elements of S are distinct. This assumption will maximize the sizes of S_1 and S_3

```
     procedure QUICKSORT($S$):
1.   if $S$ contains at most one element then return $S$
     else
         begin
2.           choose an element $a$ randomly from $S$;
3.           let $S_1$, $S_2$, and $S_3$ be the sequences of elements in $S$ less
               than, equal to, and greater than $a$, respectively;
4.           return (QUICKSORT($S_1$) followed by $S_2$ followed by
               QUICKSORT($S_3$))
         end
```

Fig. 3.7. Quicksort program.

constructed at line 3, and therefore maximize the average time spent in the recursive calls at line 4. Let $T(n)$ be the expected time required by QUICKSORT to sort a sequence of n elements. Clearly, $T(0) = T(1) = b$ for some constant b.

Suppose that element a chosen at line 2 is the ith smallest element of the n elements in sequence S. Then the two recursive calls of QUICKSORT at line 4 have an expected time of $T(i-1)$ and $T(n-i)$, respectively. Since i is equally likely to take on any value between 1 and n, and the balance of QUICKSORT(S) clearly requires time cn for some constant c, we have the relationship:

$$T(n) \leq cn + \frac{1}{n}\sum_{i=1}^{n} [T(i-1) + T(n-i)], \qquad \text{for} \quad n \geq 2. \tag{3.3}$$

Algebraic manipulation of (3.3) yields

$$T(n) \leq cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \tag{3.4}$$

We shall show that for $n \geq 2$, $T(n) \leq kn \log_e n$, where $k = 2c + 2b$ and $b = T(0) = T(1)$. For the basis $n = 2$, $T(2) \leq 2c + 2b$ follows immediately from (3 4). For the induction step, write (3.4) as

$$T(n) \leq cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki \log_e i. \tag{3.5}$$

Since $i \log_e i$ is concave upwards, it is easy to show that

$$\sum_{i=2}^{n-1} i \log_e i \leq \int_2^n x \log_e x \, dx \leq \frac{n^2 \log_e n}{2} - \frac{n^2}{4}. \tag{3.6}$$

Substituting (3.6) in (3.5) yields

$$T(n) \leq cn + \frac{4b}{n} + kn \log_e n - \frac{kn}{2}. \tag{3.7}$$

Since $n \geq 2$ and $k = 2c + 2b$, it follows that $cn + 4b/n \leq kn/2$. Thus $T(n) \leq kn \log_e n$ follows from (3.7). □

As a practical matter, we should consider two details. First is the method of "randomly" selecting element a in line 2 of QUICKSORT. The implementer might be tempted to take the easy way out by always choosing, say, the first element of sequence S. This choice could cause the performance of QUICKSORT to be considerably worse than is implied by Eq. (3.3). Frequently, the sequence passed to a sorting routine is "somewhat" sorted already, so the first element has a higher than average probability of being small. As an extreme case, the reader can check that if QUICKSORT is set to work on an already sorted sequence with no duplicates, and if the first element of S is always chosen to be the element a at line 2, then sequence S_3 will always contain one fewer element than S. In this case QUICKSORT would take a quadratic number of steps.

A better technique for choosing the partition element a at line 2 would be to use a random number generator to generate an integer i, $1 \leq i \leq |S|$,† and then select the ith element in S as a. A somewhat simpler approach would be to choose a sample of elements from S and then use the median of the sample as the partition element. For example, the median of the first, middle, and last elements of S could be used as the partition element.

The second matter is how to efficiently partition S into the three sequences S_1, S_2, and S_3. It is possible and desirable to have all n original elements in an array A. As QUICKSORT calls itself recursively, its argument S will always be in consecutive array entries, say $A[f], A[f+1], \ldots, A[l]$ for some $1 \leq f \leq l \leq n$. Having selected the "random" element a, we can arrange to partition S in place. That is, we can move S_1 to $A[f], A[f+1], \ldots, A[k]$, and $S_2 \cup S_3$ to $A[k+1]$, $A[k+2], \ldots, A[l]$, for some $k, f \leq k \leq l$. Then, $S_2 \cup S_3$ can be split up if desired, but it is usually more efficient to simply call QUICKSORT recursively on S_1 and $S_2 \cup S_3$, unless one of these sets is empty.

Perhaps the easiest way to partition S in place is to use two pointers to the array, i and j. Initially, $i = f$, and at all times, $A[f]$ through $A[i-1]$ will contain elements of S_1. Similarly, $j = l$ initially, and at all times $A[j+1]$ through $A[l]$ will hold elements of $S_2 \cup S_3$. The routine in Fig. 3.8 will perform the partition.

After the partition, we can call QUICKSORT on the array $A[f]$ through $A[i-1]$, which is S_1, and on the array $A[j+1]$ through $A[l]$, which is $S_2 \cup S_3$. However, if $i = f$, in which case $S_1 = \emptyset$, we must first remove at least one instance of a, from $S_2 \cup S_3$. It is convenient to remove the element on which we partitioned. It should also be noted that if this array represent-

† We use $|S|$ to denote the length of a sequence S.

```
begin
    $i \leftarrow f$;
    $j \leftarrow l$;
    while $i \leq j$ do
        begin
            while $A[j] \geq a$ and $j \geq f$ do $j \leftarrow j - 1$;
            while $A[i] < a$ and $i \leq l$ do $i \leftarrow i + 1$;
            if $i < j$ then
                begin
                    interchange $A[i]$ and $A[j]$;
                    $i \leftarrow i + 1$;
                    $j \leftarrow j - 1$;
                end
        end
end
```

Fig. 3.8. Partitioning S into S_1 and $S_2 \cup S_3$, in place.

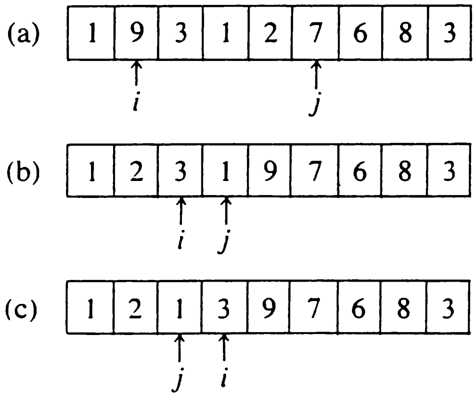


Fig. 3.9. Partitioning an array.

ation is used for sequences, we can pass arguments to QUICKSORT simply by passing pointers to the first and last locations of the portion of the array being used.

Example 3.5. Let us partition the array A

1	2	3	4	5	6	7	8	9
6	9	3	1	2	7	1	8	3

about the element $a = 3$. The **while** statement of line 4 results in j being decreased from 9, its initial value, to 7, since $A[9] = 3$ and $A[8] = 8$ are both equal to or greater than a, but $A[7] = 1 < a$. Line 5 does not increase i from its initial value of 1, since $A[1] = 6 \geq a$. Thus we interchange $A[1]$ and $A[7]$, set i to 2 and j to 6, leaving the array of Fig. 3.9(a). The results after the next two passes through the loop of lines 3–9 are shown in Fig. 3.9(b) and (c). At this point $i > j$, and the execution of the **while** statement of line 3 is complete. □

3.6 ORDER STATISTICS

A problem closely related to sorting is that of selecting the kth smallest element in a sequence of n elements.† One obvious solution is to sort the sequence into nondecreasing order and then locate the kth element. As we have seen, this would require $n \log n$ comparisons. By a careful application of the divide-and-conquer strategy, we can find the kth smallest element in $O(n)$ steps. An important special case occurs when $k = \lceil n/2 \rceil$, in which case we are finding the median of a sequence in linear time.

Algorithm 3.6. Finding the kth smallest element.

Input. A sequence S of n elements drawn from a linearly ordered set and an integer k, $1 \leq k \leq n$.

Output. The kth smallest element in S.

Method. We use the recursive procedure SELECT in Fig. 3.10. □

Let us examine Algorithm 3.6 intuitively to see why it works. The basic idea is to partition the given sequence about some element m into three subsequences S_1, S_2, S_3 such that S_1 contains all elements less than m, S_2 all

† Strictly speaking, the kth *smallest* of a sequence $a_1, a_2, \ldots, a_n$ is an element b in the sequence such that there are at most $k - 1$ values for i for which $a_i < b$ and at least k values of i for which $a_i \leq b$. For example, 4 is the second and third smallest element of the sequence 7, 4, 2, 4.

```
procedure SELECT($k$, $S$):
    if $|S| < 50$ then
        begin
            sort $S$;
            return $k$th smallest element in $S$
        end
    else
        begin
            divide $S$ into $\lfloor |S|/5 \rfloor$ sequences of 5 elements each
            with up to four leftover elements;
            sort each 5-element sequence;
            let $M$ be the sequence of medians of the 5-element sets;
            $m \leftarrow$ SELECT($\lceil |M|/2 \rceil$, $M$);
            let $S_1$, $S_2$, and $S_3$ be the sequences of elements in $S$ less
              than, equal to, and greater than $m$, respectively;
            if $|S_1| \geq k$ then return SELECT($k$, $S_1$)
            else
                if ($|S_1| + |S_2| \geq k$) then return $m$
                else return SELECT($k - |S_1| - |S_2|$, $S_3$)
        end
```

Fig. 3.10. Algorithm to select kth smallest element.

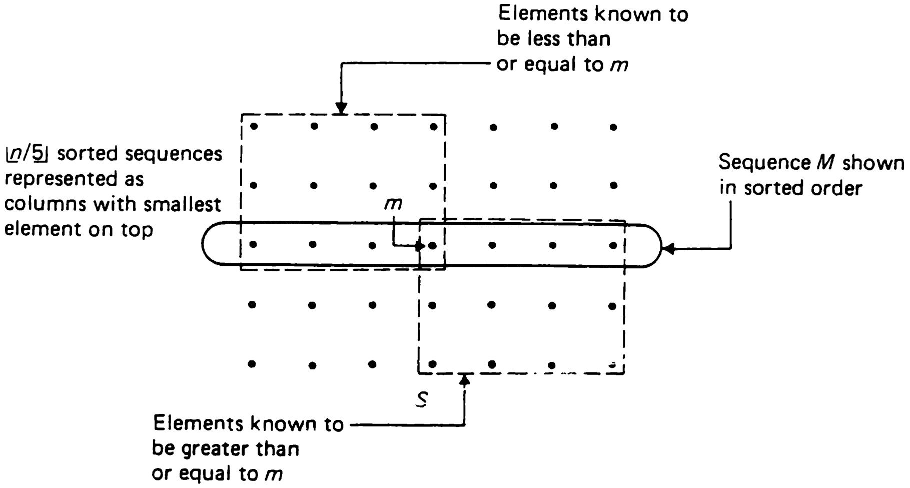

Elements known to be less than or equal to m
$\lfloor n/5 \rfloor$ sorted sequences represented as columns with smallest element on top
Sequence M shown in sorted order
m
S
Elements known to be greater than or equal to m

Fig. 3.11 Partitioning of S by Algorithm 3.6.

elements equal to m, and S_3 all elements greater than m. By counting the number of elements in S_1 and in S_2 we can determine in which of S_1, S_2, or S_3 the kth smallest element lies. In this manner we can replace the given problem by a smaller problem.

In order to obtain a linear algorithm we must be able to find a partition element in linear time such that the sizes of the subsequences S_1 and S_3 are each no more than a fixed fraction of the size of S. The trick is in how the partition element m is chosen. The sequence S is partitioned into subsequences of five elements each. Each subsequence is sorted and the median is selected from each subsequence to form the sequence M. Now M contains only $\lfloor n/5 \rfloor$ elements, and we can find its median five times faster than that of a sequence of n elements.

Furthermore at least one-fourth of the elements of S are less than or equal to m and at least one-fourth of the elements are greater than or equal to m. This is illustrated in Fig. 3.11. The question arises, why the "magic number" 5? The answer is that there are two recursive calls of SELECT, each on a sequence a fraction of the size of S. The lengths of the two sequences must sum to less than $|S|$ to make the algorithm work in linear time. Numbers other than 5 will work, but for certain numbers sorting the subsequences will become expensive. We leave it as an exercise to determine which numbers are appropriate in place of 5.

Theorem 3.9. Algorithm 3.6 finds the kth smallest element in a sequence S of n elements in time $O(n)$.

Proof. The correctness of the algorithm is a straightforward induction on the size of S, and this part of the proof is left for an exercise. Let $T(n)$ be the time required to select the kth smallest element from a sequence of size n. The sequence of medians M is of size at most $n/5$ and thus the recursive call

$$\text{SELECT}(\lceil |M|/2 \rceil, M)$$

requires at most $T(n/5)$ time.

Sequences S_1 and S_3 are each of size at most $3n/4$. To see this note that at least $\lfloor n/10 \rfloor$ elements of M are greater than or equal to m, and for each of these elements there are two distinct elements of S which are at least as large. Thus S_1 is of size at most $n - 3\lfloor n/10 \rfloor$, which for $n \geq 50$ is less than $3n/4$. A similar argument applies to S_3. Thus the recursive call at line 10 or 12 requires at most $T(3n/4)$ time. All other statements require at most $O(n)$ time. Thus, for some constant c, we have

$$\begin{aligned} T(n) &\leq cn, && \text{for } n \leq 49, \\ T(n) &\leq T(n/5) + T(3n/4) + cn, && \text{for } n \geq 50. \end{aligned} \tag{3.8}$$

From (3.8) we can prove by induction on n that $T(n) \leq 20cn$. □

3.7 EXPECTED TIME FOR ORDER STATISTICS

In this section we shall consider expected time results for selecting the kth smallest element in a sequence of n elements. We shall see that at least $n - 1$ comparisons are needed to find the kth smallest element both in the worst case and in the expected time case. Thus the selection algorithm given in the previous section is optimal under the decision tree model to within a constant factor. In this section we shall present another selection algorithm, one whose worst-case behavior is quadratic but whose expected time behavior is a fraction of that of Algorithm 3.6.

Let $S = \{a_1, a_2, \ldots, a_n\}$ be a set of n distinct elements. Let T be the decision tree of an algorithm for finding the kth smallest element in S. Every path p in T defines a relation R_p on S such that $a_iR_pa_j$ if two distinct elements a_i and a_j are compared at some vertex on p and the outcome of that comparison is either $a_i < a_j$ or $a_i \leq a_j$.† Let R_p^+ be the transitive closure of the relation R_p.‡ Intuitively, if $a_iR_p^+a_j$, then the sequence of comparisons represented by path p determines that $a_i < a_j$ since no element is compared to itself.

Lemma 3.3. If path p determines that element a_m is the kth smallest in S, then for each $i \neq m$, $1 \leq i \leq n$, either $a_iR_p^+a_m$ or $a_mR_p^+a_i$.

Proof. Suppose some element a_u is unrelated to a_m by the relation R_p^+. We shall show that by placing a_u either before a_m or after a_m in the linear ordering on S, we can contradict the assumption that path p has correctly determined that a_m is the kth smallest element in S. Let $S_1 = \{a_j | a_jR_p^+a_u\}$ and $S_2 = \{a_j | a_uR_p^+a_j\}$. Let S_3 be the remaining elements in S. By hypothesis, a_u and a_m are in S_3.

If a_j is any element in S_1 (respectively, S_2) and $a_lR_p^+a_j$ (respectively, $a_jR_p^+a_l$), then by transitivity, a_l is also in S_1 (respectively, S_2). Thus we may construct a linear order R consistent with R_p^+ such that all elements in S_1 precede all those in S_3 which, in turn, precede all those in S_2.

By hypothesis a_u is unrelated by R_p^+ to any element in S_3. Suppose that a_u precedes a_m in this linear order R, i.e., a_uRa_m. Then we can find a new linear order R' which is the same as R except that a_u is moved immediately after a_m. R' is also consistent with R_p^+. For each of R and R' we can find distinct integer values for the a's that will satisfy either R or R', respectively. But a_m cannot be the kth element in both cases, since a_m is preceded by one fewer element in R' than in R. We may therefore conclude that if some element in S is not related by R_p^+ to a_m, then T does not correctly select the kth element in the set S. The case a_mRa_u is handled symmetrically. □

† Recall that we assume each comparison a vs. b has outcome $a < b$ or $b \leq a$. If $a_i < a_j$, the comparison was a_i vs. a_j with outcome $a_i < a_j$. If $a_i \leq a_j$, the comparison was a_j vs. a_i with outcome $a_i \leq a_j$.

‡ The *transitive closure* of a relation R is the relation R^+ defined by cR^+d if and only if there is a sequence $e_1Re_2, e_2Re_3, \ldots, e_{m-1}Re_m$, where $m \geq 2$, $c = e_1$, and $d = e_m$.

Theorem 3.10. If T is a decision tree that selects the kth smallest element in a set S, $\|S\| = n$, then every leaf of T has depth at least $n - 1$.

Proof. Consider a path p in T from the root to a leaf. By Lemma 3.3, either $a_iR_p^+a_m$ or $a_mR_p^+a_i$ for each $i \neq m$, where a_m is the element selected as kth smallest. For element a_i, $i \neq m$, define the *key* comparison for a_i to be the first comparison on p involving a_i such that either:

1. a_i is compared with a_m,
2. a_i is compared with a_j, $a_iR_pa_j$, and $a_jR_p^+a_m$, or
3. a_i is compared with a_j, $a_jR_pa_i$, and $a_mR_p^+a_j$.

Intuitively, the key comparison for a_i is the first comparison from which we can eventually determine whether a_i precedes or follows a_m.

Clearly, every element a_i except a_m has a key comparison, else we would have neither $a_iR_p^+a_m$ nor $a_mR_p^+a_i$. Furthermore, it is easy to see that no comparison may be the key comparison for both elements being compared. Since there are $n - 1$ elements that must be involved in key comparisons, the path p must have length at least $n - 1$. □

Corollary. Finding the kth smallest element in S requires at least $n - 1$ comparisons in either the expected or worst-case sense.

In fact, a stronger result than Theorem 3.10 can be proven for all k except 1 or n. See Exercises 3.21–3.23.

When it comes to finding a good expected-time algorithm for computing the kth smallest element in S, a strategy similar to Quicksort works well.

Algorithm 3.7. Finding the kth smallest element.

Input. A sequence S of n elements drawn from a set with linear order $\leq$, and an integer k, $1 \leq k \leq n$.

Output. The kth smallest element in S.

Method. We apply the recursive procedure SELECT given in Fig. 3.12. □

Theorem 3.11. Algorithm 3.7 has a linear expected running time.

Proof. Let $T(n)$ be the expected running time of SELECT on a sequence of n elements. For simplicity, assume that all elements in S are distinct. (The results do not change if there are repeated elements.)

Suppose the element a chosen at line 2 is the ith smallest element in S. Then i may be any of $1, 2, \ldots, n$ with equal probability. If $i > k$, we call SELECT on a sequence of $i - 1$ elements, and if $i < k$, we call it on a sequence of $n - i$ elements. Thus the expected cost of the recursion at line 4 or 6 is

$$\frac{1}{n}\left[\sum_{i=1}^{k-1} T(n-i) + \sum_{i=k+1}^{n} T(i-1)\right] = \frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i)\right].$$

```
      procedure SELECT($k$, $S$):
1.    if $|S| = 1$ then return the single element in $S$
      else
          begin
2.            choose an element $a$ randomly from $S$;
3.            let $S_1$, $S_2$, and $S_3$ be the sequences of elements in $S$ less
                  than, equal to, and greater than $a$, respectively;
4.            if $|S_1| \geq k$ then return SELECT($k$, $S_1$)
              else
5.                if $|S_1| + |S_2| \geq k$ then return $a$
6.                else return SELECT($k - |S_1| - |S_2|$, $S_3$)
          end
```

Fig. 3.12. Selection algorithm.

The rest of procedure SELECT requires cn time for some constant c, so we have the following inequality for $n \geq 2$:

$$T(n) \leq cn + \operatorname*{MAX}_{k} \left\{ \frac{1}{n} \left[\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i) \right] \right\}. \tag{3.9}$$

We leave it as an inductive exercise to show that if $T(1) \leq c$, then for all $n \geq 2$, $T(n) \leq 4cn$. □

EXERCISES

3.1 Use Algorithm 3.1 to sort the strings *abc, acb, bca, bbc, acc, bac, baa*.

3.2 Use Algorithm 3.2 to sort the strings *a, bc, aab, baca, cbc, cc*.

3.3 Test whether the two trees in Fig. 3.13 are isomorphic in the sense of Example 3.2.

3.4 Sort the list 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7 using (a) Heapsort, (b) Quicksort, (c) Mergesort (Algorithm 2.4). In each case, how many comparisons are made?

3.5 Consider the following algorithm to sort a sequence of elements $a_1, a_2, \ldots, a_n$ stored in array A. That is, $A[i] = a_i$ for $1 \leq i \leq n$.

```
procedure BUBBLESORT($A$):
for $j = n - 1$ step $-1$ until 1 do
    for $i = 1$ step 1 until $j$ do
        if $A[i+1] < A[i]$ then interchange $A[i]$ and $A[i+1]$
```

a) Prove that BUBBLESORT sorts the elements in A into nondecreasing order.

b) Determine the worst-case and expected running times of BUBBLESORT.

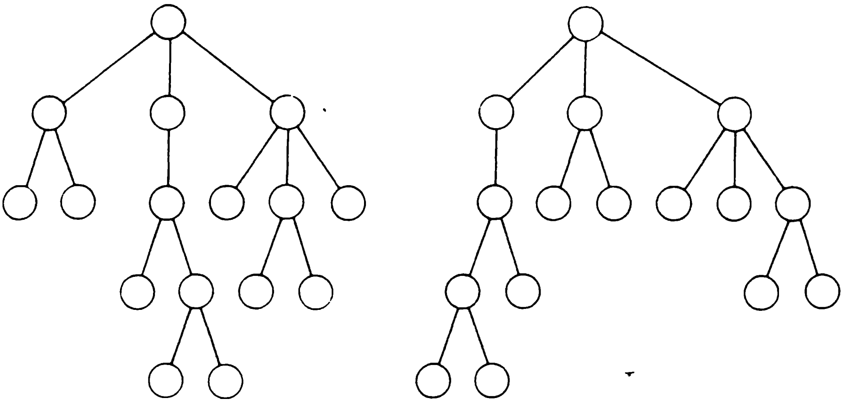


Fig. 3.13 Two trees.

3.6 Complete the proof that a heap can be constructed in linear time (Theorem 3.5).

3.7 Complete the proof that Heapsort requires $O(n \log n)$ time (Theorem 3.6).

3.8 Show that the worst-case running time of Quicksort is $O(n^2)$.

3.9 What is the worst-case running time of Algorithm 3.7 for finding the kth smallest element?

3.10 Prove that the expected time to sort n elements is bounded below by $cn \log n$ for some constant c by completing the proof of Theorem 3.7 and solving Eq. (3.2), on p. 93.

3.11 Complete the proof that Algorithm 3.6 finds the kth smallest element in time $O(n)$ (Theorem 3.9) by solving Eq. (3.8), on p. 99.

3.12 Prove the correctness of the partitioning routine in Fig. 3.8 (p. 96) and analyze its running time.

3.13 Complete the proof that Algorithm 3.7 for finding the kth smallest element has expected time $O(n)$ (Theorem 3.11) by solving Eq. (3.9), on p. 102.

3.14 Show that the expected number of comparisons used by Algorithm 3.7 is at most $4n$. Can you improve this bound if you know the value of k for which the algorithm will be used?

3.15 Let S be a sequence of elements with m_i copies of the ith element for $1 \le i \le k$. Let $n = \sum_{i=1}^{k} m_i$. Prove that

$$O\left(n + \log\left(\frac{n!}{m_1!\, m_2! \cdots m_k!}\right)\right)$$

comparisons are necessary and sufficient to sort S by a comparison sort.

3.16 Let $S_1, S_2, \ldots, S_k$ be sets of integers in the range 1 to n, where the sum of the cardinalities of the S_i's is n. Describe an $O(n)$ algorithm to sort all of the S_i's.

3.17 Given a sequence $a_1, a_2, \ldots, a_n$ and a permutation $\pi(1), \pi(2), \ldots, \pi(n)$, write a Pidgin ALGOL algorithm to rearrange the sequence in place into the order

$a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$. What are the worst-case and the expected running times of your algorithm?

*3.18 In building a heap of size $2^k - 1$, we built two heaps of size $2^{k-1} - 1$, then combined them by adding a root and pushing the element at the root down to its proper place. One could just as easily have built a heap by adding one element at a time as a new leaf and pushing the new element up the tree. Write an algorithm for building a heap by adding one leaf at a time and compare the asymptotic growth rate of your algorithm to that of Algorithm 3.3.

3.19 Consider a rectangular array. Sort the elements in each row into increasing order. Next sort the elements in each column into increasing order. Prove that the elements in each row remain sorted.

*3.20 Let $a_1, a_2, \ldots, a_n$ be a sequence of elements and let p and q be positive integers. Consider the subsequences formed by selecting every pth element. Sort these subsequences. Repeat the process for q. Prove that the subsequences of distance p remain sorted.

**3.21 Consider finding both the largest and second largest elements from a set of n elements by means of comparisons. Prove that $n + \lceil \log n \rceil - 2$ comparisons are necessary and sufficient.

**3.22 Show that the expected number of comparisons needed to find the kth smallest element in a sequence of n elements is at least $(1 + .75\alpha(1 - \alpha))n$, where $\alpha = k/n$, and k and n are sufficiently large.

**3.23 Show that in the worst case, $n + \text{MIN}(k, n - k + 1) - 2$ comparisons are necessary to find the kth smallest element in a set of n elements.

**3.24 Let S be a set of n integers. Assume you can perform only addition of elements of S and comparisons between sums. Under these conditions how many comparisons are required to find the maximum element of S?

*3.25 Algorithm 3.6 divides its argument into subsequences of size 5. Does the algorithm work for other sizes such as 3, 7, or 9? Select that size which minimizes the total number of comparisons. Figure 3.14 indicates the fewest known number of comparisons to sort various size sets. For $n \leq 12$, the number of comparisons shown is known to be optimal.

*3.26 Algorithm 3.5 divides a sequence into subsequences of length 5, finds the median of each subsequence, and then finds the median of the medians. Instead of finding the median of the medians would it be more efficient to find some other element such as the $\lfloor k/5 \rfloor$th?

Size n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Comparisons	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50

Fig. 3.14. Fewest known comparisons to sort n elements.

*3.27 Consider the following sorting method. Start with a sequence consisting of one element. Insert the remaining elements into the sequence one at a time by binary search. Devise a data structure which will allow you to perform binary search and insert elements quickly. Can you implement this sort in $O(n \log n)$ time?

**3.28 Instead of selecting one random element to partition a set of size n, as we did in Quicksort or Algorithm 3.7, we could choose a small sample, say of size s, find its median, and use that median to partition the entire set. Show how to choose s as a function of n to minimize the expected number of comparisons needed to sort.

**3.29 Extend the idea of Exercise 3.28 to minimize the number of comparisons needed to find order statistics. [*Hint:* Choose two elements from the sample set that with high probability straddle the desired element.]

3.30 A sorting method is *stable* if equal elements remain in the same relative order in the sorted sequence as they were in originally. Which of the following sorting algorithms are stable?

a) BUBBLESORT (Exercise 3.5)
b) Mergesort (Algorithm 2.4)
c) Heapsort (Algorithm 3.4)
d) Quicksort (Algorithm 3.5)

Research Problem

3.31 There are several open problems concerning the number of comparisons needed in certain situations. For example, one might wish to find the k smallest elements out of a set of n. The case $k = 3$ is discussed by Pratt and Yao [1973]. It is not known, for $n \geq 13$, whether the numbers given in Fig. 3.14 are optimal for sorting n elements. For small n, the sorting algorithm of Ford and Johnson [1959] is optimal in terms of the number of comparisons.

BIBLIOGRAPHIC NOTES

Knuth [1973a] is an encyclopedic compendium of sorting methods. Heapsort originated with Williams [1964] and was improved by Floyd [1964]. Quicksort is due to Hoare [1962]. Improvements to Quicksort along the lines of Exercise 3.28 were suggested by Singleton [1969] and Frazer and McKellar [1970]. Algorithm 3.6, the linear worst-case algorithm for finding order statistics, is by Blum, Floyd, Pratt, Rivest, and Tarjan [1972]. Hadian and Sobel [1969] and Pratt and Yao [1973] discuss the number of comparisons for finding certain order statistics.

The result in Exercise 3.21 is due to Kislitsyn [1964]. Exercises 3.22, 3.23 and 3.29 are from Floyd and Rivest [1973], which also contains a stronger lower bound than stated in Exercise 3.22. Exercises 3.19, 3.20, and some generalizations are discussed in Gale and Karp [1970] and Liu [1972]. An interesting application of sorting to finding the convex hull of a set of points in the plane is given by Graham [1972]. Stable sorting has been treated by Horvath [1974].

DATA STRUCTURES FOR SET MANIPULATION PROBLEMS

CHAPTER 4

A good way to approach the design of an efficient algorithm for a given problem is to examine the fundamental nature of the problem. Often, a problem can be formulated in terms of basic mathematical objects such as sets, and an algorithm for the problem can be outlined in terms of fundamental operations on these basic objects. An advantage of this point of view is that we can examine several alternative data structures in order to select the one that is best suited for the problem as a whole. Thus good data structure design goes hand-in-hand with good algorithm design.

In this chapter we shall study seven fundamental operations on sets, which are characteristic of many searching and information retrieval problems. We present a variety of data structures for representing sets and consider the suitability of each structure when a sequence of various types of operations is to be performed.

4.1 FUNDAMENTAL OPERATIONS ON SETS

We shall consider the following basic operations on sets.

1. MEMBER(a, S). Determine whether a is a member of S; if so, print "yes," otherwise, print "no."
2. INSERT(a, S). Replace set S by $S \cup \{a\}$.
3. DELETE(a, S). Replace set S by $S - \{a\}$.
4. UNION(S_1, S_2, S_3). Replace sets S_1 and S_2 by $S_3 = S_1 \cup S_2$. We shall assume that S_1 and S_2 are disjoint when this operation is performed, in order to avoid the necessity of deleting duplicate copies of an element in $S_1 \cup S_2$.
5. FIND(a). Print the name of the set of which a is currently a member. If a is in more than one set, the instruction is undefined.
6. SPLIT(a, S). Here we assume S is a set with a linear order $\leq$ on its elements. This operation partitions S into two sets S_1 and S_2 such that $S_1 = \{b|b \leq a \text{ and } b \in S\}$ and $S_2 = \{b|b > a \text{ and } b \in S\}$.
7. MIN(S). Print the smallest (with respect to $\leq$) element of the set S.

Many problems encountered in practice can be reduced to one or more subproblems, where each subproblem can be abstractly formulated as a sequence of basic instructions to be performed on some data base (universal set of elements). In this chapter we shall consider sequences of instructions σ in which the instructions in σ are drawn from a subset of these seven set operations.

For example, processing sequences of MEMBER, INSERT, and DELETE operations is an integral part of many searching problems. A data structure that can be used to process a sequence of MEMBER, INSERT, and DELETE operations will be called a *dictionary*. In this chapter we shall study several data structures, such as hash tables, binary search trees, and 2–3 trees, which can be used to implement dictionaries.

There are a number of questions of interest. We shall be primarily interested in the time complexity of σ, that is, the amount of time required to execute the instructions in σ measured as a function of the length of σ and the size of the data base. We shall consider both average and worst-case time complexity and shall make a further distinction between on-line and off-line complexity.

Definition. The *on-line* execution of σ requires that the instructions in σ be executed from left to right, executing the ith instruction in σ without looking at any following instructions. The *off-line* execution of σ permits all of σ to be scanned before any answers need to be produced.

Clearly, any on-line algorithm can be used as an off-line algorithm, but the converse is not necessarily true. We shall see situations in which an off-line algorithm is faster than any known on-line algorithm. However, in many applications we are restricted to considering only on-line algorithms.

Given a sequence of instructions to execute, the most basic question is what data structure we should use to represent the underlying data base. Often, a problem will require a careful balance between two conflicting desires. In the typical situation, the sequence will specify several operations which are to be repeatedly performed, often in an unknown order. There may be several data structures, each of which makes one operation very easy to perform but other operations very hard. In many cases, the best solution is a compromise. We often use some data structure that makes no operation as easy as it could be, but one that makes the overall performance better than that of any obvious approach.

We now give an example that illustrates how the spanning tree problem for graphs can be formulated in terms of a sequence of set operations.

Definition. Let $G = (V, E)$ be an undirected graph. A *spanning tree* of G is an undirected tree $S = (V, T)$. A *spanning forest for* $G = (V, E)$ is a set of undirected trees $\{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\}$ such that the V_i's form a partition† of V and each T_i is a (possibly empty) subset of E.

A *cost function* c for a graph $G = (V, E)$ is a mapping from E to real numbers. $c(G')$, the cost of a subgraph $G' = (V', E')$ of G, is $\Sigma_{e \in E'} c(e)$.

Example 4.1. Consider the algorithm shown in Fig. 4.1 for finding $S = (V, T)$, a minimum-cost spanning tree of a given graph $G = (V, E)$. This minimum-cost spanning tree algorithm is discussed in detail in Section 5.1, and its application is illustrated in Example 5.1.

The algorithm of Fig. 4.1 uses three sets, E, T, and VS. Set E contains the edges of the given graph G. Set T is used to collect the edges of the final spanning tree. The algorithm works by transforming a spanning forest for G

† That is, $V_1 \cup V_2 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$.

```
    begin
1.      $T \leftarrow \emptyset$;
2.      $VS \leftarrow \emptyset$;
3.      for each vertex $v \in V$ do add the singleton set $\{v\}$ to $VS$;
4.      while $\|VS\| > 1$ do
            begin
5.              choose $(v, w)$, an edge in $E$ of lowest cost;
6.              delete $(v, w)$ from $E$;
7.              if $v$ and $w$ are in different sets $W_1$ and $W_2$ in $VS$ then
                    begin
8.                      replace $W_1$ and $W_2$ in $VS$ by $W_1 \cup W_2$;
9.                      add $(v, w)$ to $T$
                    end
            end
    end
```

Fig. 4.1. Minimum-cost spanning tree algorithm.

into a single spanning tree. The set VS contains the vertex sets of the trees in the spanning forest. Initially, VS contains each vertex of G in a set by itself.

We can view the algorithm as a sequence of operations that manipulate the three sets E, T, and VS. Line 1 initializes the set T. Lines 2 and 3 initialize VS; the elements of VS are sets themselves. Line 3 adds the initial singleton sets to VS. Line 4, which controls the main loop of this algorithm, requires maintaining a count of the number of sets of vertices in the set VS. In line 7 we determine whether edge (v, w) connects two trees in the spanning forest. If so, the two trees are merged in line 8 and the edge (v, w) is added to the final spanning tree in line 9.

Line 7 requires that we be able to find the name of the set in VS that contains a particular vertex. (The actual names used for the sets in VS are not important, so arbitrary names can be used.) Basically, line 7 requires that we be able to handle the FIND primitive efficiently. Similarly, line 8 requires that we be able to execute the UNION operation on disjoint sets of vertices.

Finding a data structure to handle either the UNION operation by itself or the FIND operation by itself is relatively easy. However, here the data structure should be one in which both UNION and FIND are easy to implement. Furthermore, since execution of the UNION operation in line 8 depends on the outcome of the FIND operations in line 7, the execution of the required sequence of UNION and FIND instructions must be on-line. We shall study two such structures in Sections 4.6 and 4.7.

Consider the sequence of operations performed on the set of edges E. At line 5 we need the MIN interrogation primitive and in line 6 we need the DELETE primitive. We have already encountered a good data structure for these two primitives, the heap of Section 3.4. (Although there the heap was used to find the largest element, it should be obvious that the heap can be used just as easily to find the smallest element.)

Finally, the set T of edges for the spanning tree requires only the operation INSERT at line 9 to add a new edge to T. A simple list suffices here. □

4.2 HASHING

Besides the question of which instructions appear in a given sequence of instructions σ, another important issue in the selection of a suitable data structure to process σ is the size of the data base (universal set) being manipulated by the operations in σ. For example, we saw in Chapter 3 that the problem of sorting a sequence of n elements could be done in linear time using a bucket sort if the elements were integers in some suitable range. However, if the elements were drawn from an arbitrary linearly ordered set, then $O(n \log n)$ time was the best we could do.

In this section we shall consider the problem of maintaining a changing set S of elements. New elements will be added to S, old elements will be removed from S, and from time to time we shall have to answer the question, "Is element x currently in S?" This problem is modeled naturally by a dictionary; we need a data structure that will permit sequences of MEMBER, INSERT, and DELETE instructions to be processed conveniently. We shall assume that the elements that can appear in S are chosen from an extremely large universal set, so that representing S as a bit vector is impractical.

Example 4.2. A compiler or an assembler keeps track in a "symbol table" of all the identifiers it has seen in the program it is translating. For most programming languages the set of all possible identifiers is extremely large. For example, in FORTRAN there are about 1.62×10^9 possible identifiers. Thus it is infeasible to represent a symbol table by an array with one entry for each possible identifier, independent of whether that identifier actually appears in the program.

The operations which a compiler performs on a symbol table are of two types. First, new identifiers must be added to the table as they are encountered. This job involves setting up a location in the table into which the particular identifier is stored and into which data about the identifier (e.g., is it real or integer?) can be stored. Second, from time to time the compiler may request information about an identifier (e.g., is the identifier of type integer?).

Thus a data structure that can handle the operations INSERT and MEMBER is likely to be satisfactory for implementing a symbol table. In fact, the data structure discussed in this section is often used to implement a symbol table. □

We shall consider *hashing*, a technique which handles not only INSERT and MEMBER instructions, as needed in symbol table construction, but the

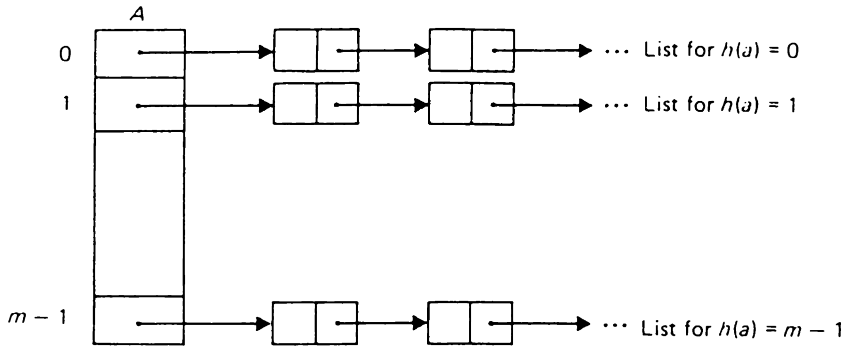


Fig. 4.2 ·A hashing scheme.

DELETE instruction as well. There are many variations on the theme of hashing, and we shall consider only the basic idea here.

A hashing scheme is represented by Fig. 4.2. There is a *hashing function* h, which maps elements of the universal set (e.g., in the case of a symbol table, the set of all possible identifiers) to the integers 0 through $m - 1$. We assume throughout that, for all elements a, $h(a)$ can be computed in a constant amount of time. There is a size m array A whose entries are pointers to lists of members of the set S. The list pointed to by $A[i]$ consists of all those elements a in S such that $h(a) = i$.

To execute the instruction INSERT(a, S), we compute $h(a)$ and then search the list pointed to by $A[h(a)]$. If a is not on this list, it is appended to the end of this list. To execute the instruction DELETE(a, S), we again search the list $A[h(a)]$ and delete a if it appears on this list. Similarly, MEMBER(a, S) is answered by scanning the list $A[h(a)]$.

The computational complexity of this hashing scheme is easy to analyze. From a worst-case standpoint it is not very good. For example, suppose we have a sequence σ of n distinct INSERT operations. It is possible that h applied to each element to be inserted yields the same number, with the result that all elements appear on the same list. In this situation, it requires time proportional to i to process the ith instruction in σ. Thus hashing can require time proportional to n^2 to add all n elements to the set S.

However, in an expected time sense hashing looks much better. If we assume that $h(a)$ is equally likely to be any value between 0 and $m - 1$, and that $n \leq m$ elements are to be inserted, then on inserting the ith element, the expected length of the list on which it is placed is $(i - 1)/m$, which is always less than 1. Thus the expected time needed to insert n elements is $O(n)$. If $O(n)$ DELETE and MEMBER operations are executed in conjunction with the INSERT operations, the total expected cost is still $O(n)$.

Bear in mind that this analysis presumes m, the size of the hash table, to be equal to or greater than n, the maximum size of the set S. However, n is usually not known in advance. A reasonable way to proceed when n is unknown is to be prepared to construct a sequence of hash tables T_0, T_1, T_2, . . .

We choose some suitable value for the size m of the initial hash table T_0. Then, once the number of elements inserted in T_0 exceeds m, we create a new hash table T_1 of size $2m$ and, by rehashing,† move all elements currently in T_0 into T_1. The old hash table T_0 can now be discarded. We can then continue inserting more elements in T_1 until the number of elements exceeds $2m$. At this point we create a new hash table T_2 of size $4m$, and rehash the elements from T_1 to T_2. In general, we create a table T_k of size $2^k m$ as soon as table T_{k-1} contains $2^{k-1}m$ elements. We continue in this way until we have inserted all elements.

Consider the expected time required to insert 2^k elements into a hash table, using this scheme and assuming $m = 1$. We see that this process is modeled by the recurrence

$$T(1) = 1,$$

$$T(2^k) = T(2^{k-1}) + 2^k,$$

whose solution is clearly $T(2^k) = 2^{k+1} - 1$.

We conclude that a sequence of n INSERT, MEMBER, and DELETE instructions can be processed in $O(n)$ expected time by hashing.

The choice of the hashing function h is important. If the elements to be added to S are integers uniformly distributed in some range 0 to r, where $r >> n$, then $h(a)$ can be taken to be a modulo m, where m is the size of the current hash table. Other examples of hashing functions can be found in some of the references cited at the end of the chapter.

4.3 BINARY SEARCH

In this section we shall compare three different solutions to a simple searching problem. We are given a set S with n elements drawn from some large universal set. We are to process a sequence σ consisting only of MEMBER instructions.

The most straightforward solution is to store the elements of S in a list. Each MEMBER(a, S) instruction is processed by sequentially searching the list until the given element a is found or until all of the elements in the list have been examined. This solution requires time proportional to $n \times |\sigma|$ to process all instructions in σ both in the worst case and in the expected case. The main advantage of this scheme is that it requires very little preprocessing time.

† A new hashing function, one which will give values from 0 through $2m - 1$, must be used.

```
procedure SEARCH($a, f, l$):
if $f > l$ then return "no"
else
    if $a = A[\lfloor (f+l)/2 \rfloor]$ then return "yes"
    else
        if $a < A[\lfloor (f+l)/2 \rfloor]$ then
            return SEARCH($a, f, \lfloor (f+l)/2 \rfloor - 1$)
        else return SEARCH($a, \lfloor (f+l)/2 \rfloor + 1, l$)
```

Fig. 4.3. Binary search algorithm.

Another solution is to enter the elements of S in a hash table of size $\|S\|$. An instruction MEMBER(a, S) is executed by searching the list $h(a)$. If a good hashing function h can be found, then this solution requires $O(|\sigma|)$ expected and $O(n|\sigma|)$ worst-case time to process σ. The primary difficulty is finding a hashing function that will uniformly distribute the elements of S throughout the hash table.

If there is a linear order $\leq$ on S, a third solution is to use binary search. Here we store the elements of S in an array A. Next we sort the.array so that $A[1] < A[2] < \cdots < A[n]$. Then to determine whether an element a is in S, we compare a with the element b stored in location $\lfloor (1+n)/2 \rfloor$. If $a = b$, we halt and answer "yes." Otherwise, we repeat this procedure on the first half of the array if $a < b$, or on the last half if $a > b$. By repeatedly splitting the search area in half, we need never make more than $\lceil \log (n+1) \rceil$ comparisons to find a or to determine that it is not in S.

The recursive procedure SEARCH(a, f, l) given in Fig. 4.3 looks for element a in locations $f, f+1, f+2, \ldots, l$ of the array A. In order to determine whether a is in S, we call SEARCH(a, 1, n).

To understand why this procedure works, we can imagine that the array A represents a binary tree. The root is at location $\lfloor (1+n)/2 \rfloor$, and the left and right sons of the root are located at $\lfloor (1+n)/4 \rfloor$ and $\lfloor 3(1+n)/4 \rfloor$, and so on. This interpretation of binary search will become clearer in the next section.

It can be easily shown that SEARCH makes at most $\lceil \log (n+1) \rceil$ comparisons when looking for any element in A, since no path in the underlying tree is longer than $\lceil \log (n+1) \rceil$. If all elements are equally likely to be targets for a search, then it can also be shown (Exercise 4.4) that SEARCH gives the optimal expected number of comparisons (namely, $|\sigma| \times \log n$) to process the MEMBER instructions in the sequence σ.†

† Of course, hashing does not work by comparisons alone, so it is possible that hashing is "better" than binary search, and in many cases, it is.

4.4 BINARY SEARCH TREES

Consider the following problem. We have a set S in which elements are being inserted, and from which elements are being deleted. From time to time we may want to know whether a given element is in S or we may want to know what is the smallest element currently in S. We assume that the elements being added to S come from a large universal set that is linearly ordered by a relation $\leq$. This problem can be abstracted as processing a sequence of INSERT, DELETE, MEMBER, and MIN instructions.

We have seen that a hash table is a good data structure for processing sequences of INSERT, DELETE, and MEMBER instructions. However, it is not possible to find the smallest element in a hash table without searching the entire table. A data structure that is suited for all four instructions is the binary search tree.

Definition. A *binary search tree* for a set S is a labeled binary tree in which each vertex v is labeled by an element $l(v) \in S$ such that

1. for each vertex u in the left subtree of v, $l(u) < l(v)$,
2. for each vertex u in the right subtree of v, $l(u) > l(v)$, and
3. for each element $a \in S$, there is exactly one vertex v such that $l(v) = a$.

Note that conditions 1 and 2 imply that the labels of the tree are in inorder. Also, condition 3 follows from 1 and 2.

Example 4.3. Figure 4.4 shows one possible binary search tree for the ALGOL keywords **begin, else, end, if, then.** The linear order here is lexicographic order. □

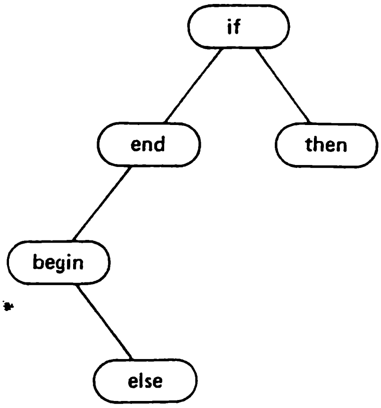


Fig. 4.4 A binary search tree.

To determine whether an element a is in a set S represented by a binary search tree, we compare a with the label of the root. If the label of the root is a, then a is clearly in S. If a is less than the label of the root, we then search the left subtree of the root (if one exists). If a is greater than the label of the root, we search the right subtree of the root. If a is in the tree, it will eventually be located. Otherwise, the process terminates when we would have to search a nonexistent left or right subtree.

Algorithm 4.1. Searching a binary search tree.

Input. A binary search tree T for a set S and an element a.

Output. "Yes" if $a \in S$, "no" otherwise.

Method. If T is empty,† return "no." Otherwise let r be the root of T. The algorithm then consists of a single procedure call, SEARCH(a, r), of the recursive procedure SEARCH defined in Fig. 4.5. □

Algorithm 4.1 clearly suffices to execute the instruction MEMBER(a, S). Moreover, we can easily modify it to execute the instruction INSERT(a, S). If the tree is empty, we create a root labeled a. If the tree is not empty and the element to be inserted is not found on the tree, then the procedure SEARCH fails to find a son either at line 3 or at line 5. Instead of returning "no" at line 4 or 6, respectively, a new vertex is created for the element and attached where the missing son belongs.

Binary search trees are convenient for executing MIN and DELETE instructions as well. The smallest element in a binary search tree T is found by following the path $v_0, v_1, \ldots, v_p$, where v_0 is the root of T, v_i is the left son of v_{i-1} for $1 \leq i \leq p$, and v_p has no left son. The label on v_p is the smallest element in T. In certain problems it might be convenient to maintain a pointer to v_p to provide a constant access time to the smallest element.

Implementation of the instruction DELETE(a, S) is a little harder. Suppose that the element a to be deleted is found at vertex v. Three cases can occur.

1. Vertex v is a leaf. In this case remove vertex v from the tree.
2. Vertex v has exactly one son. In this case make the father of v the father of the son, effectively removing v from the tree. (If v is the root, then make the son of v the new root.)
3. Vertex v has two sons. Find the largest element b in the left subtree of v. Recursively, remove the vertex containing b from the subtree and then set the label of v to b. Note that b will be the largest element that is smaller than a in the entire tree.

† Although our definition of a tree requires a tree to have at least one vertex, the root, in many algorithms we shall treat the empty tree (the tree with no vertices) as a binary tree.

```
procedure SEARCH($a$, $v$):
if $a = l(v)$ then return "yes"
else
    if $a < l(v)$ then
        if $v$ has a left son $w$ then return SEARCH($a$, $w$)
        else return "no"
    else
        if $v$ has a right son $w$ then return SEARCH($a$, $w$)
        else return "no"
```

Fig. 4.5. Searching a binary search tree.

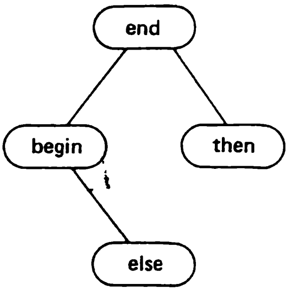


Fig. 4.6 Binary search tree after DELETE.

We leave a Pidgin ALGOL specification of the DELETE operation as an exercise. Note that a single MEMBER, INSERT, DELETE, or MIN instruction can use $O(n)$ time.

Example 4.4. Suppose we wish to remove the word **if** from the binary search tree in Fig. 4.4. The word **if** is located at the root, which has two sons. The largest word less than **if** (lexicographically) in the left subtree of the root is **end.** We remove the vertex labeled **end** from the tree and replace **if** by **end** at the root. Then, since **end** had one son (**begin**), we make **begin** be a son of the root, leaving the tree of Fig. 4.6. □

Consider the time complexity of a sequence of n INSERT instructions, when a binary search tree is used to represent the underlying set. The time required to insert an element a into a binary search tree is bounded by a constant times the number of comparisons made between a and the elements already in the tree. Thus we can measure time in terms of the number of comparisons made.

In the worst case, adding n elements to a tree could require quadratic time. For example, suppose the sequence of elements to be added happens to be sorted (in increasing order). In this case the search tree would consist of a single chain of right sons. However, if n random elements are inserted, then the required insertion time is $O(n \log n)$, as the following theorem shows.

Theorem 4.1. The expected number of comparisons needed to insert n random elements into an initially empty binary search tree is $O(n \log n)$, for $n \geq 1$.

Proof. Let $T(n)$ be the number of comparisons between sequence elements needed to create a binary search tree from the sequence $a_1, a_2, \ldots, a_n$. We assume $T(0) = 0$. Let $b_1, b_2, \ldots, b_n$ be this sequence sorted in ascending order.

If $a_1, \ldots, a_n$ is a random sequence of elements, then a_1 is equally likely to be b_j for any j, $1 \leq j \leq n$. Element a_1 becomes the root of the binary search tree, and in the final tree the $j-1$ elements $b_1, b_2, \ldots, b_{j-1}$ will be in the left subtree of the root and the $n-j$ elements $b_{j+1}, b_{j+2}, \ldots, b_n$ will be in the right subtree.

Let us count the expected number of comparisons made when inserting $b_1, b_2, \ldots, b_{j-1}$ into the tree. Each of these elements is compared once with the root, giving a total of $j-1$ comparisons with the root. Then inductively, $T(j-1)$ more comparisons are necessary to insert $b_1, b_2, \ldots, b_{j-1}$ into the left subtree of the tree. All told, $j-1+T(j-1)$ comparisons are made to insert $b_1, b_2, \ldots, b_{j-1}$ into the binary search tree. Similarly, $n-j+T(n-j)$ comparisons are made to insert $b_{j+1}, b_{j+2}, \ldots, b_n$ into the tree.

Since j is equally likely to have any value between 1 and n, we have

$$T(n) = \frac{1}{n} \sum_{j=1}^{n} (n-1+T(j-1)+T(n-j)). \tag{4.1}$$

Simple algebraic manipulation of (4.1) yields

$$T(n) = n-1+\frac{2}{n} \sum_{j=0}^{n-1} T(j). \tag{4.2}$$

Using the techniques of Section 3.5 we can show that

$$T(n) \leqslant kn \log n$$

where $k = \log_e 4 = 1.39$. Thus the expected number of comparisons to insert n elements into a binary search tree is $O(n \log n)$. □

In summary, using the techniques of this section we can process a random sequence of n INSERT, DELETE, MEMBER, and MIN instructions in $O(n \log n)$ expected time. The worst-case performance is quadratic. However, even this can be improved to $O(n \log n)$ by one of the balanced tree schemes (2–3 trees, AVL trees or trees of bounded balance) discussed in Section 4.9 and Exercises 4.30–4.37.

4.5 OPTIMAL BINARY SEARCH TREES

In Section 4.3 we were given a set $S = \{a_1, a_2, \ldots, a_n\}$, that is, a subset of some large universal set U, and we were asked to design a data structure that would allow us to process efficiently a sequence σ consisting only of MEMBER instructions. Let us reconsider this problem, but this time let us assume that, in addition to being given the set S, we are given the probability that the instruction MEMBER(a, S) will appear in σ for all elements a in the universal set U. We would now like to design a binary search tree for S such that a sequence σ of MEMBER instructions can be processed on-line with the smallest expected number of comparisons.

Let $a_1, a_2, \ldots, a_n$ be the elements in the set S in increasing order, and let p_i be the probability of the instruction MEMBER(a_i, S) in σ. Let q_0 be the probability that an instruction of the form MEMBER(a, S), for some $a < a_1$, appears in σ. Let q_i be the probability that an instruction of the form MEMBER(a, S), for some $a_i < a < a_{i+1}$ appears in σ. Finally, let q_n be the probability that an instruction of the form MEMBER(a, S), for some $a > a_n$, appears in σ. To define the cost of a binary search tree it is convenient to add $n + 1$ fictitious leaves to the binary tree to reflect the elements in $U - S$. We shall call these leaves 0, 1, . . . , n.

Figure 4.7 shows the binary search tree of Fig. 4.4 with these fictitious leaves. For example, the leaf labeled 3 represents those elements a such that **end** $< a <$ **if**.

We need to define the cost of a binary search tree. If element a is the label $l(v)$ of some vertex v, then the number of vertices visited when we

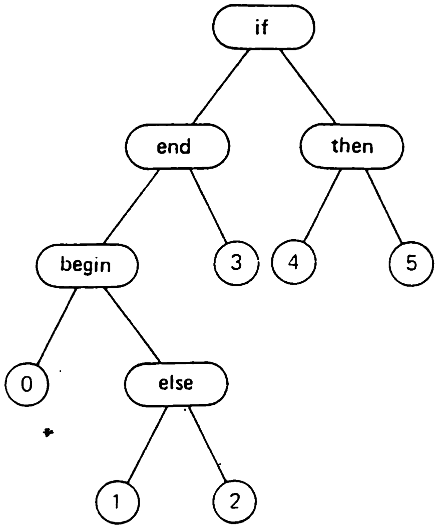


Fig. 4.7 Binary search tree with added leaves.

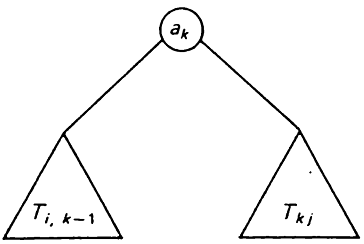


Fig. 4.8 Subtree T_{ij}.

process the instruction MEMBER(a, S) is one more than the depth of vertex v. If $a \notin S$, and $a_i < a < a_{i+1}$, then the number of vertices visited to process the instruction MEMBER(a, S) is equal to the depth of the fictitious leaf i. Thus the *cost* of a binary search tree can be defined as

$$\sum_{i=1}^{n} p_i \times (\text{DEPTH}(a_i) + 1) + \sum_{i=0}^{n} q_i \times \text{DEPTH}(i).$$

Once we have a minimum-cost binary search tree T, we can execute a sequence of MEMBER instructions with the smallest expected number of vertex visits, simply by using Algorithm 4.1 on T to process each MEMBER instruction.

Given the p_i's and q_i's, how do we find a minimum-cost tree? The divide-and-conquer approach suggests determining the element a_i that belongs at the root. This would divide the problem into two subproblems: constructing the left subtree and constructing the right subtree. However, there seems to be no easy way to determine the root, short of solving the entire problem. This suggests we consider $2n$ subproblems, two for each possible root. This naturally leads to a dynamic programming solution.

For $0 \leq i < j \leq n$, let T_{ij} be a minimum-cost tree for the subset of elements $\{a_{i+1}, a_{i+2}, \ldots, a_j\}$. Let c_{ij} be the cost of T_{ij} and r_{ij} the root of T_{ij}. The *weight* w_{ij} of T_{ij} is defined to be $q_i + (p_{i+1} + q_{i+1}) + \cdots + (p_j + q_j)$.

A tree T_{ij} consists of a root a_k, plus a left subtree $T_{i,k-1}$ which is a minimum-cost tree for $\{a_{i+1}, a_{i+2}, \ldots, a_{k-1}\}$, plus a right subtree T_{kj} which is a minimum-cost tree for $\{a_{k+1}, a_{k+2}, \ldots, a_j\}$, as shown in Fig. 4.8. If $i = k - 1$, there is no left subtree and if $k = j$, there is no right subtree. For notational convenience we shall treat T_{ii} as the empty tree. The weight w_{ii} of T_{ii} is q_i and its cost c_{ii} is 0.

In T_{ij}, $i < j$, the depth of every vertex in the left and right subtrees has increased by one from what the depths were in $T_{i,k-1}$ and T_{kj}. Thus c_{ij}, the cost of T_{ij}, can be expressed as

$$\begin{aligned} c_{ij} &= w_{i,k-1} + p_k + w_{kj} + c_{i,k-1} + c_{kj} \\ &= w_{ij} + c_{i,k-1} + c_{kj}. \end{aligned}$$

The value of k to use is that which minimizes the sum $c_{i,k-1} + c_{kj}$. Thus to find an optimal tree T_{ij} we compute the cost for each k, $i < k \leq j$, of the tree with root a_k, left subtree $T_{i,k-1}$, and right subtree T_{kj}, and then select a tree of minimum cost. The following algorithm contains the details.

Algorithm 4.2. Construction of an optimal binary search tree.

Input. A set of elements $S = \{a_1, a_2, \ldots, a_n\}$. We assume $a_1 < a_2 < \cdots < a_n$. We are also given probabilities $q_0, q_1, \ldots, q_n$ and $p_1, p_2, \ldots, p_n$ such that for $1 \leq i < n$, q_i denotes the probability of executing an instruction of the form MEMBER(a, S) taken over all a such that $a_i < a < a_{i+1}$, q_0 denotes the probability of executing MEMBER(a, S) for $a < a_1$, q_n denotes the probability of executing MEMBER(a, S) for $a > a_n$, and for $1 \leq i \leq n$ p_i denotes the probability of executing MEMBER(a_i, S).

Output. A minimum cost binary search tree for S.

Method

1. For $0 \leq i < j \leq n$, we compute r_{ij} and c_{ij} in order of increasing value of $j - i$, using the dynamic programming algorithm of Fig. 4.9.
2. Having computed the r_{ij}'s we call BUILDTREE(0, n) to recursively construct an optimal tree for T_{0n}. The procedure BUILDTREE is given in Fig. 4.10. □

```
begin
    for $i \leftarrow 0$ until $n$ do
        begin
            $w_{ii} \leftarrow q_i$;
            $c_{ii} \leftarrow 0$
        end;
    for $l \leftarrow 1$ until $n$ do
        for $i \leftarrow 0$ until $n - l$ do
            begin
                $j \leftarrow i + l$;
                $w_{ij} \leftarrow w_{i,j-1} + p_j + q_j$;
                let $m$ be a value of $k$, $i < k \leq j$, for which $c_{i,k-1} + c_{kj}$
                    is minimum;
                $c_{ij} \leftarrow w_{ij} + c_{i,m-1} + c_{mj}$;
                $r_{ij} \leftarrow a_m$
            end
end
```

Fig. 4.9. Algorithm to compute roots of optimal subtrees.

```
procedure BUILDTREE($i, j$):
begin
    create vertex $v_{ij}$, the root of $T_{ij}$;
    label $v_{ij}$ by $r_{ij}$;
    let $m$ be the subscript of $r_{ij}$ (i.e., $r_{ij} = a_m$);
    if $i < m - 1$ then make BUILDTREE($i$, $m - 1$) the left subtree of $v_{ij}$;
    if $m < j$ then make BUILDTREE($m, j$) the right subtree of $v_{ij}$
end
```

Fig. 4.10. Procedure to construct optimal binary search tree.

Example 4.5. Consider the four elements $a_1 < a_2 < a_3 < a_4$ with $q_0 = \frac{1}{8}$, $q_1 = \frac{3}{16}$, $q_2 = q_3 = q_4 = \frac{1}{16}$, and $p_1 = \frac{1}{4}$, $p_2 = \frac{1}{8}$, $p_3 = p_4 = \frac{1}{16}$. Figure 4.11 shows the values of w_{ij}, r_{ij}, and c_{ij} computed by the algorithm given in Fig. 4.9. For notational convenience, the values of w_{ij} and c_{ij} in this table have all been multiplied by 16.

$l = j - i$ ↓ \ i →	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$
1	$w_{01} = 9$ $c_{01} = 9$ $r_{01} = a_1$	$w_{12} = 6$ $c_{12} = 6$ $r_{12} = a_2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = a_3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = a_4$	
2	$w_{02} = 12$ $c_{02} = 18$ $r_{02} = a_1$	$w_{13} = 8$ $c_{13} = 11$ $r_{13} = a_2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = a_4$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = a_1$	$w_{14} = 10$ $c_{14} = 18$ $r_{14} = a_2$			
4	$w_{04} = 16$ $c_{04} = 33$ $r_{04} = a_2$				

Fig. 4.11. Values of w_{ij}, c_{ij}, and r_{ij}.

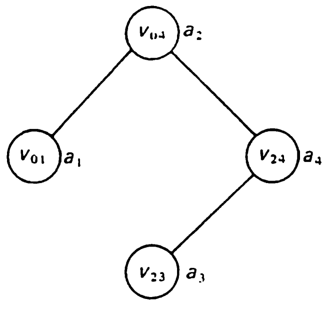


Fig. 4.12 A minimum-cost tree.

For example, to compute r_{14} we must compare the values of $c_{11} + c_{24}$, $c_{12} + c_{34}$, and $c_{13} + c_{44}$, which (multiplied by 16) are, respectively, 8, 9, and 11. Thus in line 8 of Fig. 4.9, $k = 2$ achieves the minimum, so $r_{14} = a_2$.

Having computed the table of Fig. 4.11, we construct the tree T_{04} by calling BUILDTREE(0, 4). The resulting binary search tree is shown in Fig. 4.12. This tree has cost 33/16. □

Theorem 4.2. Algorithm 4.2 requires $O(n^3)$ time to construct an optimal binary search tree.

Proof. Once we have computed the table of r_{ij}'s, we construct an optimal tree from this table in $O(n)$ time with the procedure BUILDTREE. There are only n calls of the procedure and each call takes constant time.

The costliest part is the dynamic programming algorithm of Fig. 4.9. Line 8 requires $O(j - i)$ time to find a value of k that minimizes $c_{i,k-1} + c_{kj}$. The other steps in the loop of lines 5–10 require constant time. The outer loop at line 4 is executed n times. The inner loop is executed at most n times for each iteration of the outer loop. Thus the total cost is $O(n^3)$.

For the correctness of the algorithm, a simple induction on $l = j - i$ shows that r_{ij} and c_{ij} are correctly computed at lines 9 and 10.

To show that an optimal tree is correctly constructed by BUILDTREE, we observe that if a vertex v_{ij} is the root of a subtree for $\{a_{i+1}, a_{i+2}, \ldots, a_j\}$, then its left son will be the root of an optimal tree for $\{a_{i+1}, a_{i+2}, \ldots, a_{m-1}\}$, where $r_{ij} = a_m$ and its right son will be the root of an optimal tree for $\{a_{m+1}, a_{m+2}, \ldots, a_j\}$. An inductive proof that BUILDTREE(i, j) correctly constructs an optimal tree for $\{a_{i+1}, a_{i+2}, \ldots, a_j\}$ should thus be evident. □

In Algorithm 4.2 we may restrict our search for m in line 8 of Fig. 4.9 to the range between the positions of the roots of $T_{i,j-1}$ and $T_{i+1,j}$ and still be guaranteed to find a minimum. With this modification Algorithm 4.2 can be made to find an optimal tree in $O(n^2)$ time.

4.6 A SIMPLE DISJOINT-SET UNION ALGORITHM

Consider the handling of vertices in the spanning tree algorithm of Example 4.1. The set-processing problem that arose had the following three characteristics.

1. Whenever two sets were merged, the two sets were disjoint.
2. The elements of the sets could be treated as integers in the range 1 through n.
3. The operations were UNION and FIND.

In this section and the next we shall consider data structures for problems of this nature. Assume we are given n elements, which we shall take to be the integers $1, 2, \ldots, n$. Initially, each element is assumed to be in a set by itself. A sequence of UNION and FIND instructions is to be executed. A UNION instruction, we recall, is of the form UNION(A, B, C), indicating that two disjoint sets named A and B are to be replaced by their union, and their union is to be named C. In applications it is often unimportant what we choose to name a set, so we shall assume that sets can be named by integers in the range 1 to n. Moreover, we shall assume no two sets are ever given the same name.

There are several interesting data structures to handle this problem. In this section we shall present a data structure that is capable of processing a sequence containing up to $n - 1$ UNION instructions and $O(n \log n)$ FIND instructions in time $O(n \log n)$. In the next section we shall describe a data structure that will handle a sequence of $O(n)$ UNION and FIND instructions with a worst-case time that is almost linear in n. These data structures are also capable of handling sequences of INSERT, DELETE, and MEMBER instructions with the same computational complexity.

Note that the searching algorithms we considered in Sections 4.2–4.5 assumed that the elements were drawn from a universal set that was much larger than the number of instructions to be executed. In this section, we are assuming that the universal set is approximately the same size as the length of the sequence of instructions to be executed.

Perhaps the simplest data structure for the UNION–FIND problem is to use an array to represent the collection of sets present at any one time. Let R be an array of size n such that $R[i]$ is the name of the set containing element i. Since the names of sets are unimportant, we may initially take $R[i] = i$, $1 \le i \le n$, to denote the fact that at the start the collection of sets is $\{\{1\}, \{2\}, \ldots, \{n\}\}$ and set $\{i\}$ is named i.

The instruction FIND(i) is executed by printing the current value of $R[i]$. Thus the cost of executing a FIND instruction is a constant, which is the best we could hope for.

To execute the instruction UNION(A, B, C), we sequentially scan the array R, and each entry containing either A or B is set to C. Thus the cost of executing a UNION instruction is $O(n)$. A sequence of n UNION instructions could require $O(n^2)$ time, which is undesirable.

This simple-minded algorithm can be improved in several ways. One improvement would take advantage of linked lists. Another would recognize that it is more efficient always to merge a smaller set into a larger one. To do so, we need to distinguish between "internal names," which are used to identify sets in the R array, and "external names," the ones mentioned in the UNION instructions. Both are presumably integers between 1 and n, but not necessarily the same.

Let us consider the following data structure for this problem. As before, we use an array R such that $R[i]$ contains the "internal" name of the set containing the element i. But now, for each set A we construct a linked list LIST[A] containing the elements of the set. Two arrays, LIST and NEXT, are used to implement this linked list. LIST[A] is an integer j indicating that j is the first element in the set whose internal name is A. NEXT[j] gives the next element in A, NEXT[NEXT[j]] the next element, and so on.

In addition, we shall use an array called SIZE, where SIZE[A] is the number of elements in set A. Also, sets are renamed internally. Two arrays INTERNAL_NAME and EXTERNAL_NAME associate internal and external names. That is, EXTERNAL_NAME[A] gives the real name (the name dictated by the UNION instructions) of the set with internal name A. INTERNAL_NAME[j] is the internal name of the set with external name j. The internal names are the ones used in the array R.

Example 4.6. Suppose $n = 8$ and we have the collection of three sets $\{1, 3, 5, 7\}$, $\{2, 4, 8\}$, and $\{6\}$ with external names 1, 2, and 3, respectively. The data structures for these three sets are shown in Fig. 4.13, where we assume 1, 2, and 3 have the internal names 2, 3, and 1, respectively. □

The instruction FIND(i) is executed as before, by consulting $R[i]$ to determine the internal name of the set currently containing element i. Then EXTERNAL_NAME[$R[i]$] gives the real name of the set containing i.

A union instruction of the form UNION(I, J, K) is executed as follows. (The line numbers refer to Fig. 4.14.)

1. We determine the internal names for sets I and J (lines 1–2).
2. We compare the relative sizes of sets I and J by consulting the array SIZE (lines 3–4).
3. We traverse the list of elements of the smaller set and change the corresponding entries in the array R to the internal name of the larger set (lines 5–9).

	R	NEXT
1	2	3
2	3	4
3	2	5
4	3	8
5	2	7
6	1	0
7	2	0
8	3	0

	LIST	SIZE	EXTERNAL_NAME
1	6	1	3
2	1	4	1
3	2	3	2

Sets (with external names)

1 = {1, 3, 5, 7}

2 = {2, 4, 8}

3 = {6}

	INTERNAL_NAME
1	2
2	3
3	1

Fig. 4.13. Data structures for UNION–FIND algorithm.

```
     procedure UNION(I, J, K):
     begin
1.      A ← INTERNAL_NAME[I];
2.      B ← INTERNAL_NAME[J];
3.      wlg assume SIZE[A] ≤ SIZE[B]
4.        otherwise interchange roles of A and B in
          begin
5.            ELEMENT ← LIST[A];
6.            while ELEMENT ≠ 0 do
                 begin
7.                  R[ELEMENT] ← B;
8.                  LAST ← ELEMENT;
9.                  ELEMENT ← NEXT[ELEMENT]
                 end;
10.           NEXT[LAST] ← LIST[B];
11.           LIST[B] ← LIST[A];
12.           SIZE[B] ← SIZE[A] + SIZE[B];
13.           INTERNAL_NAME[K] ← B;
14.           EXTERNAL_NAME[B] ← K
          end
     end
```

Fig. 4.14. Implementation of UNION instruction.

4. We merge the smaller set into the larger by appending the list of elements of the smaller set to the beginning of the list for the larger set (lines 10–12).
5. We give the combined set the external name K (lines 13–14).

By merging the smaller set into the larger, the UNION instruction can be executed in time proportional to the cardinality of the smaller set. The complete details are given in the procedure in Fig. 4.14.

Example 4.7. After execution of the instruction UNION(1, 2, 4), the data structures in Fig. 4.13 would become as shown in Fig. 4.15. □

Theorem 4.3. Using the algorithm of Fig. 4.14 we can execute $n - 1$ UNION operations (the maximum possible number) in $O(n \log n)$ steps.

Proof. Since the cost of each UNION is proportional to the number of elements moved, apportioning the cost of each UNION instruction uniformly among the elements moved results in a fixed charge to an element each time it is moved. The key observation is that each time an element is moved from a list, it finds itself on a list which is at least twice as long as before. Thus no element can be moved more than $\log n$ times and hence the total cost charged

	R	NEXT
1	2	3
2	2	4
3	2	5
4	2	8
5	2	7
6	1	0
7	2	0
8	2	1

	LIST	SIZE	EXTERNAL_NAME
1	6	1	3
2	2	7	4
3	–	–	–
4	–	–	–

Sets (with external names)

$3 = \{6\}$

$4 = \{1, 2, 3, 4, 5, 7, 8\}$

	INTERNAL_NAME
1	–
2	–
3	1
4	2

Fig. 4.15. Data structures after UNION instruction.

to any element is $O(\log n)$. The total cost is obtained by summing the costs charged to the elements. Thus the total cost is $O(n \log n)$. □

It follows from Theorem 4.3 that if m FIND and up to $n - 1$ UNION instructions are executed, then the total time spent is $O(\mathrm{MAX}(m, n \log n))$. If m is on the order of $n \log n$ or greater, then this algorithm is actually optimal to within a constant factor. However, in many situations we shall find that m is $O(n)$, and in this case, we can do better than $O(\mathrm{MAX}(m, n \log n))$, as we shall see in the next section.

4.7 TREE STRUCTURES FOR THE UNION-FIND PROBLEM

In the last section we presented a data structure for the UNION-FIND problem that would allow the processing of $n - 1$ UNION instructions and $O(n \log n)$ FIND instructions in time $O(n \log n)$. In this section we shall present a data structure consisting of a forest of trees to represent the collection of sets. This data structure will allow the processing of $O(n)$ UNION and FIND instructions in almost linear time.

Suppose we represent each set A by a rooted undirected tree T_A, where the elements of A correspond to the vertices of T_A. The name of the set is attached to the root of the tree. An instruction of the form UNION(A, B, C) can be executed by making the root of T_A a son of the root of T_B and changing the name at the root of T_B to C. An instruction of the form FIND(i) can be executed by locating the vertex representing element i in some tree T in the forest, and traversing the path from this vertex to the root of T, where we find the name of the set containing i.

With such a scheme, the cost of merging two trees is a constant. However, the cost of a FIND(i) instruction is on the order of the length of the path from vertex i to its root. This path could have length $n - 1$. Thus the cost of executing $n - 1$ UNION instructions followed by n FIND instructions could be as high as $O(n^2)$. For example, consider the cost of the following sequence:

```
UNION(1, 2, 2)
UNION(2, 3, 3)
  .
  .
  .
UNION(n - 1, n, n)
FIND(1)
FIND(2)
  .
  .
  .
FIND(n)
```

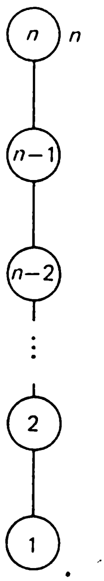


Fig. 4.16 Tree after UNION instructions.

The $n - 1$ UNION instructions result in the tree shown in Fig. 4.16. The cost of the n FIND instructions is proportional to

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

However, the cost can be reduced if the trees can be kept balanced. One way to accomplish this is to keep count of the number of vertices in each tree and, when merging two sets, always to attach the smaller tree to the root of the larger. This technique is analogous to the technique of merging smaller sets into larger, which we used in the last section.

Lemma 4.1. If in executing each UNION instruction the root of the tree with fewer vertices (ties are broken arbitrarily) is made a son of the root of the larger, then no tree in the forest will have height greater than or equal to h unless it has at least 2^h vertices.

Proof. The proof is by induction on h. For $h = 0$, the hypothesis is true since every tree has at least one vertex. Assume the induction hypothesis true for all values less than $h \geq 1$. Let T be a tree of height h with fewest vertices. Then T must have been obtained by merging two trees T_1 and T_2, where T_1 has height $h - 1$ and T_1 has no more vertices than T_2. By the induction hypothesis T_1 has at least 2^{h-1} vertices and hence T_2 has at least 2^{h-1} vertices, implying that T has at least 2^h vertices. □

Consider the worst-case execution time for a sequence of n UNION and FIND instructions using the forest data structure, with the modification that

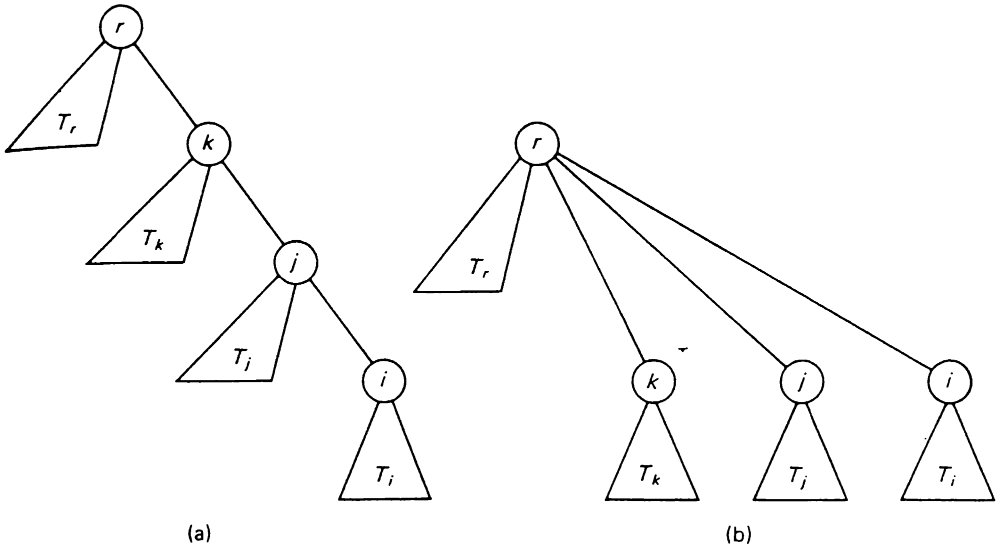


Fig. 4.17 Effect of path compression.

in a UNION the root of the smaller tree becomes a son of the root of the larger tree. No tree can have height greater than log n. Hence the execution of $O(n)$ UNION and FIND instructions costs at most $O(n \log n)$ units of time. This bound is tight, in that there are sequences of n instructions that will take time proportional to $n \log n$.

We now introduce another modification to this algorithm, called *path compression*. Since the cost of the FIND's appears to dominate the total cost, we shall try to reduce the cost of the FIND's. Each time a FIND(i) instruction is executed we traverse the path from vertex i to its root r. Let i, $v_1, v_2, \ldots, v_n, r$ be the vertices on this path. We then make each of $i, v_1, v_2, \ldots, v_{n-1}$ a son of the root. Figure 4.17(b) illustrates the effect of the instruction FIND(i) on the tree of Fig. 4.17(a).

The complete tree-merging algorithm for the UNION–FIND problem, including path compression, is expressed by the following algorithm.

Algorithm 4.3. Fast disjoint-set union algorithm.

Input. A sequence σ of UNION and FIND instructions on a collection of sets whose elements consist of integers from 1 through n. The set names are also assumed to be integers from 1 to n, and initially, element i is by itself in a set named i.

Output. The sequence of responses to the FIND instructions in σ. The response to each FIND instruction is to be produced before looking at the next instruction in σ.

Method. We describe the algorithm in three parts—the initialization, the response to a FIND, and the response to a UNION.

1. *Initialization.* For each element i, $1 \le i \le n$, we create a vertex v_i. We set COUNT[v_i] = 1, NAME[v_i] = i, and FATHER[v_i] = 0. Initially, each vertex v_i is a tree by itself. In order to locate the root of set i, we create an array ROOT with ROOT[i] pointing to v_i. To locate the vertex for element i, we create an array ELEMENT, initially with ELEMENT[i] = v_i.
2. *Executing* FIND(i). The program is shown in Fig. 4.18. Starting at vertex ELEMENT[i] we follow the path to the root of the tree, making

```
begin
    make LIST empty;
    v ← ELEMENT[i];
    while FATHER[v] ≠ 0 do
        begin
            add v to LIST;
            v ← FATHER[v]
        end;
    comment v is now the root;
    print NAME[v];
    for each w on LIST do FATHER[w] ← v
end
```

Fig. 4.18. Executing instruction FIND(i).

```
begin
  wlg assume COUNT[ROOT[i]] ≤ COUNT[ROOT[j]]
    otherwise interchange i and j in
    begin
        LARGE ← ROOT[j];
        SMALL ← ROOT[i];
        FATHER[SMALL] ← LARGE;
        COUNT[LARGE] ← COUNT[LARGE] + COUNT[SMALL];
        NAME[LARGE] ← k;
        ROOT[k] ← LARGE
    end
end
```

Fig. 4.19. Executing instruction UNION(i, j, k).

a list of vertices encountered. At the root, the name of the set is printed, and each vertex on the path traversed is made a son of the root.

3. *Executing* UNION(i, j, k). Via the array ROOT, we find the roots of the trees representing sets i and j. We then make the root of the smaller tree a son of the root of the larger. See Fig. 4.19. □

We shall show that path compression speeds up the algorithm considerably. To calculate the improvement we introduce two functions F and G. Let

$$F(0) = 1,$$

$$F(i) = 2^{F(i-1)}, \qquad \text{for } i > 0.$$

The function F grows extremely fast, as the table in Fig. 4.20 shows. The function $G(n)$ is defined to be smallest integer k such that $F(k) \geq n$. The function G grows extremely slowly. In fact, $G(n) \leq 5$ for all "practical" values of n, i.e., for all $n \leq 2^{65536}$.

We shall now prove that Algorithm 4.3 will execute a sequence σ of cn UNION and FIND instructions in at most $c'nG(n)$ time, where c and c' are constants, c' depending on c. For simplicity, we assume the execution of a UNION instruction takes one "time unit" and the execution of the instruction FIND(i) takes a number of time units proportional to the number of vertices on the path from the vertex labeled i to the root of the tree containing this vertex.†

n	$F(n)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65536}

Fig. 4.20. Some values of F.

† Thus one "time unit" in the sense used here requires some constant number of steps on a RAM. Since we neglect constant factors, order-of-magnitude results can as well be expressed in terms of "time units."

Definition. It is convenient to define the *rank* of a vertex with respect to the sequence σ of UNION and FIND instructions as follows:

1. Delete the FIND instructions from σ.
2. Execute the resulting sequence σ' of UNION instructions.
3. The rank of a vertex v is the height of v in the resulting forest.

We shall now derive some important properties of the rank of a vertex.

Lemma 4.2. There are at most $n/2^r$ vertices of rank r.

Proof. By Lemma 4.1 each vertex of rank r has at least 2^r descendants in the forest which results from executing σ'. Since the sets of descendants of any two distinct vertices of the same height in a forest are disjoint and since there are at most $n/2^r$ disjoint sets of 2^r or more vertices, there can be at most $n/2^r$ vertices of rank r. □

Corollary. No vertex has rank greater than $\log n$.

Lemma 4.3. If at some time during the execution of σ, w is a proper descendant of v, then the rank of w is less than the rank of v.

Proof. If at some time during the execution of σ, w is made a descendant of v, then w will be a descendant of v in the forest resulting from the execution of the sequence σ'. Thus the height of w must be less than the height of v, so the rank of w is less than the rank of v. □

We now partition the ranks into *groups*. We put rank r in group $G(r)$. For example, ranks 0 and 1 are in group 0, rank 2 is in group 1, ranks 3 and 4 are in group 2, ranks 5 through 16 are in group 3. For $n > 1$, the largest possible rank, $\lfloor \log n \rfloor$, is in rank group $G(\lfloor \log n \rfloor) \leq G(n) - 1$.

Consider the cost of executing a sequence σ of cn UNION and FIND instructions. Since each UNION instruction can be executed at the cost of one time unit, all UNION instructions in σ can be executed in $O(n)$ time. In order to bound the cost of executing all FIND instructions we use an important "bookkeeping" trick. The cost of executing a single FIND is apportioned between the FIND instruction itself and certain vertices on the path in the forest data structure which are actually moved. The total cost is computed by summing over all FIND instructions the cost apportioned to them, and then summing the cost assigned to the vertices, over all vertices in the forest.

We charge for the instruction FIND(i) as follows. Let v be a vertex on the path from the vertex representing i to the root of tree containing i.

1. If v is the root, or if FATHER[v] is in a different rank group from v, then charge one time unit to the FIND instruction itself.
2. If both v and its father are in the same rank group, then charge one time unit to v.

By Lemma 4.3 the vertices going up a path are monotonically increasing in rank, and since there are at most $G(n)$ different rank groups, no FIND instruction is charged more than $G(n)$ time units under rule 1. If rule 2 applies, vertex v will be moved and made the son of a vertex of higher rank than its previous father. If vertex v is in rank group $g > 0$, then v can be moved and charged at most $F(g) - F(g - 1)$ times before it acquires a father in a higher rank group. In rank group 0, a vertex can be moved at most once before obtaining a father in a higher group. From then on, the cost of moving v will be charged to the FIND instructions by rule 1.

To obtain an upper bound on the charges made to the vertices themselves, we multiply the maximum possible charge to any vertex in a rank group by the number of vertices in that rank group, and sum over all rank groups. Let $N(g)$ be the number of vertices in rank group $g > 0$. Then by Lemma 4.2:

$$\begin{aligned} N(g) &\le \sum_{r=F(g-1)+1}^{F(g)} n/2^r \\ &\le (n/2^{F(g-1)+1})[1 + \tfrac{1}{2} + \tfrac{1}{4} + \cdots] \\ &\le n/2^{F(g-1)} \\ &\le n/F(g). \end{aligned}$$

The maximum charge to any vertex in rank group $g > 0$ is less than or equal to $F(g) - F(g - 1)$. Thus the maximum charge to all vertices in rank group g is bounded by n. The same statement clearly applies for $g = 0$ as well. Since there are at most $G(n)$ rank groups, the maximum charge to all vertices is $nG(n)$. Therefore, the total amount of time required to process cn FIND instructions is at most $cnG(n)$ charged to the FIND's and at most $nG(n)$ charged to the vertices. Thus we have the following theorem.

Theorem 4.4. Let c be any constant. Then there exists another constant c' depending on c such that Algorithm 4.3 will execute a sequence σ of cn UNION and FIND instructions on n elements in at most $c'nG(n)$ time units.

Proof. By the above discussion. □

It is left as an exercise to show that if the primitive operations INSERT and DELETE, as well as UNION and FIND, are permitted in the sequence σ, then σ can still be executed in $O(nG(n))$ time.

It is not known whether Theorem 4.4 provides a tight bound on the running time of Algorithm 4.3. However, as a matter of theoretical interest, in the remainder of this section we shall prove that the running time of Algorithm 4.3 is not linear in n. To do this, we shall construct a particular sequence of UNION and FIND instructions, which Algorithm 4.3 takes more than linear time to process.

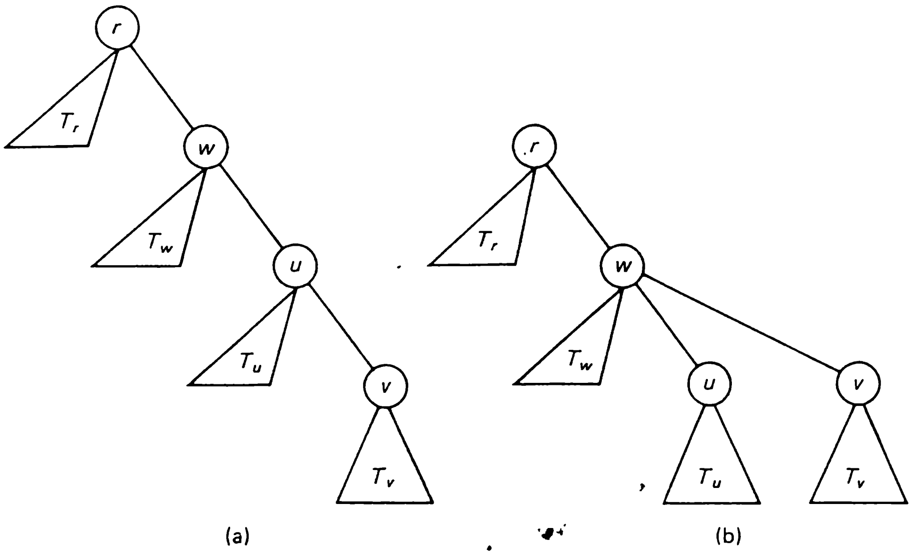


Fig. 4.21 Effect of partial FIND operation.

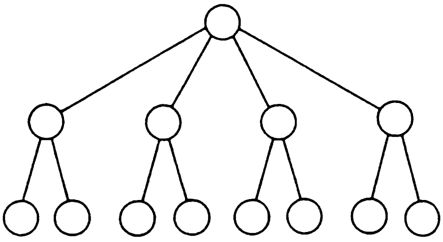


Fig. 4.22 The tree $T(2)$.

It is convenient to introduce a new operation on trees which we shall call *partial FIND*, or PF for short. Let T be a tree in which $v, v_1, v_2, \ldots, v_m, w$ is a path from a vertex v to an ancestor w. (w is not necessarily the root.) The operation PF(v, w) makes each of $v, v_1, v_2, \ldots, v_{m-1}$ sons of vertex w. We say this partial FIND is of *length* $m + 1$ (if $v = w$, the length is 0). Figure 4.21(b) illustrates the effect of PF(v, w) on the tree of Fig. 4.21(a).

Suppose we are given a sequence σ of UNION and FIND instructions. When we execute a given FIND instruction in σ we locate a vertex v in some tree T and follow the path from v to the root w of T. Now suppose we execute only the UNION instructions in σ, ignoring the FIND's. This will result in a forest F of trees. We can still capture the effect of a given FIND instruction in σ by locating in F the vertices v and w used by the original FIND instruction and then executing PF(v, w). Note that the vertex w may no longer be a root in F.

In deriving a lower bound on the running time of Algorithm 4.3, we consider the behavior of the algorithm on a sequence of UNION's followed by PF's which can be replaced by a sequence of UNION's and FIND's whose execution time is the same. From the following special trees we shall derive particular sequences of UNION's and PF's on which Algorithm 4.3 takes more than linear time.

Definition. For $k \geq 0$, let $T(k)$ be the tree such that

1. each leaf of $T(k)$ has depth k.
2. each vertex of height h has 2^h sons, $h \geq 1$.

Thus the root of $T(k)$ has 2^k sons, each of which is a root of a copy of $T(k-1)$. Figure 4.22 shows $T(2)$.

Lemma 4.4. With a sequence of UNION instructions we can create, for any $k \geq 0$, a tree $T'(k)$ that contains as a subgraph the tree $T(k)$. Furthermore, at least one-quarter of the vertices in $T'(k)$ are leaves of $T(k)$.

Proof. The proof proceeds by induction on k. The lemma is trivial for $k = 0$, since $T(0)$ consists of a single vertex. To construct $T'(k)$ for $k > 0$, first construct $2^k + 1$ copies of $T'(k-1)$. Form the tree $T'(k)$ by selecting one copy of $T'(k-1)$ and then merging into it, one by one, each of the remaining copies. The root of the resulting tree has (among others) 2^k sons, each of which is a root of $T'(k-1)$.

Let $N'(k)$ be the total number of vertices in $T'(k)$ and let $L(k)$ be the number of leaves in $T(k)$. Then

$$N'(0) = 1$$
$$N'(k) = (2^k + 1)N'(k-1), \qquad \text{for } k \geq 1,$$

and

$$L(0) = 1$$
$$L(k) = 2^k L(k-1), \qquad \text{for } k \geq 1;$$

so

$$\frac{L(k)}{N'(k)} = \frac{\prod_{i=1}^{k} 2^i}{\prod_{i=1}^{k} (2^i + 1)} = \frac{2}{3} \prod_{i=2}^{k} \frac{1}{1 + 2^{-i}}, \qquad \text{for } k \geq 1. \tag{4.3}$$

We note that for $i \geq 2$, $\log_e (1 + 2^{-i}) < 2^{-i}$, so

$$\log_e \left(\prod_{i=2}^{k} \frac{1}{1 + 2^{-i}} \right) \geq -\sum_{i=2}^{k} 2^{-i} \geq -\tfrac{1}{2}. \tag{4.4}$$

Using (4.3) and (4.4) together we have

$$\frac{L(k)}{N'(k)} \geq \tfrac{2}{3} e^{-1/2} \geq \tfrac{1}{4},$$

thus proving the lemma. □

We shall construct a sequence of UNION and PF instructions that will first build the tree $T'(k)$ and then perform PF's on the leaves of the subgraph $T(k)$. We shall now show that for every $l > 0$, there exists a k such that we can perform a PF of length l in succession on every leaf of $T(k)$.

> **Definition.** Let $D(c, l, h)$ be the smallest value of k such that if we replace every subtree in $T(k)$ whose root has height h by any tree having l leaves and height at least 1, then we may perform a PF of length c on each leaf in the resulting tree.

> **Lemma 4.5.** $D(c, l, h)$ is defined (i.e., finite) for all c, l, and h greater than zero.

Proof. The proof involves a double induction. We wish to prove the result by induction on c. But in order to prove the result for c given the result for $c - 1$, we must also do an induction on l.

The basis, $c = 1$, is easy. $D(1, l, h) = h$ for all l and h, since a PF of length 1 does not move any vertices.

Now for the induction on c, suppose that for all l and h, $D(c - 1, l, h)$ is defined. We must show that $D(c, l, h)$ is defined for all l and h. This is done by induction on l.

For the basis of this induction, we show

$$D(c, 1, h) \leq D(c - 1, 2^{h+1}, h + 1).$$

Note that when $l = 1$, we have substituted trees with a single leaf for subtrees with roots at the vertices of height h in $T(k)$ for some k. Let H be the set of vertices of height h in this $T(k)$. Clearly, in the modified tree each leaf is the proper descendant of a unique member of H. Therefore, if we could do PF's of length $c - 1$ on all the members of H, we could certainly do PF's of length c on all the leaves.

Let $k = D(c - 1, 2^{h+1}, h + 1)$. By the hypothesis for the induction on c, we know that k exists. If we consider the vertices of height $h + 1$ in $T(k)$, we see that each has 2^{h+1} sons, all of which are members of H. If we delete all proper descendants of the vertices in H from $T(k)$, we have in effect substituted trees of height 1 with 2^{h+1} leaves for each subtree having roots at height $h + 1$. By the definition of D, $k = D(c - 1, 2^{h+1}, h + 1)$ is sufficiently large so that PF's of length $c - 1$ can be done on all its leaves, i.e., the members of H.

Now, to complete the induction on c, we must do the inductive step for l. In particular, we shall show:

$$D(c, l, h) \leq D(c - 1, 2^{D(c,l-1,h)(1+D(c,l-1,h))/2}, D(c, l - 1, h)) \qquad \text{for } l > 1. \tag{4.5}$$

To prove (4.5), let $k = D(c, l - 1, h)$ and let k' be the right side of (4.5). We must find a way to substitute a tree of l leaves for each vertex of height h in

$T(k')$, then perform a PF of length c on each leaf. We begin by performing the PF's on $l-1$ of the leaves of each substituted tree. By the inductive hypothesis for the induction on l, we can perform the PF's on $l-1$ of the leaves of each substituted tree in these subtrees.

Having done PF's on $l-1$ of the leaves, we find that the lth leaf of each substituted tree now has a father distinct from that of the lth leaf of any other substituted tree. Call the set of such fathers F. If we can do PF's of length $c-1$ on the fathers, then we can do PF's of length c on the leaves. Let S be a subtree whose root had height k in $T(k')$. It is easy to check that S has $2^{k(k+1)/2}$ leaves in $T(k')$. Thus, after we have done the PF's, the number of vertices in S which are also in F is at most $2^{k(k+1)/2}$. What remains of S can thus be regarded as an arbitrary tree with $2^{k(k+1)/2}$ leaves, the vertices in F. By the inductive hypotheses for c and l, (4.5) holds. □

Theorem 4.5. Algorithm 4.3 has a time complexity which is greater than cn for any constant c.

Proof. Assume there is a constant c such that Algorithm 4.3 will execute any sequence of $n-1$ MERGE and n FIND instructions in no more than cn time units. Select $d > 4c$, and calculate $k = D(d, 1, 1)$. Construct $T'(k)$ by a sequence of UNION instructions. Since we can perform a PF of length d on each leaf of the embedded tree $T(k)$, and since the leaves of $T(k)$ make up more than one-quarter of the vertices of $T'(k)$, this sequence of UNION and PF instructions will require more than cn time units, a contradiction. □

4.8 APPLICATIONS AND EXTENSIONS OF THE UNION-FIND ALGORITHM

We have seen how a sequence of the primitive instructions UNION and FIND naturally arose in the spanning tree problem of Example 4.1. In this section we present several other problems which give rise to sequences of UNION and FIND instructions. In our first problem, the computation can be performed off-line, that is, the entire sequence of instructions can be read before any answers need to be produced.

Application 1. Off-line MIN problem

We are given two types of instructions, INSERT(i) and EXTRACT_MIN. We start with a set S which is initially empty. Each time an instruction INSERT(i) is encountered we place the integer i in S. Each time an instruction EXTRACT_MIN is executed, we find the minimum element in S and delete it.

Let σ be a sequence of INSERT and EXTRACT_MIN instructions such that for each i, $1 \le i \le n$, the instruction INSERT(i) appears at most once. Given the sequence σ, we are to find the sequence of integers deleted

```
for $i \leftarrow 1$ until $n$ do
    begin
        $j \leftarrow$ FIND($i$);
        if $j \le k$ then
            begin
                print $i$ "is deleted by the "$j$"th EXTRACT_MIN instruction";
                UNION($j$, SUCC[$j$], SUCC[$j$]);
                SUCC[PRED[$j$]] $\leftarrow$ SUCC[$j$];
                PRED[SUCC[$j$]] $\leftarrow$ PRED[$j$]
            end
    end
```

Fig. 4.23. Program for off-line MIN problem.

by the EXTRACT_MIN instructions. The problem is off-line since we assume we are given the complete sequence σ before we need to compute even the first element of the output sequence.

The off-line MIN problem can be solved by the following method. Let k be the number of EXTRACT_MIN instructions in σ. We may write σ as $\sigma_1 E \sigma_2 E \sigma_3 E \cdots \sigma_k E \sigma_{k+1}$, where each σ_j, $1 \le j \le k+1$, consists only of INSERT instructions and E stands for one EXTRACT_MIN instruction. We shall simulate σ via the set union algorithm, Algorithm 4.3. We initialize a sequence of sets for the set union algorithm by letting the set named j, $1 \le j \le k+1$, contain the element i, provided the instruction INSERT(i) occurs in the subsequence σ_j. Two arrays PRED and SUCC are used to create a doubly linked sorted list for those values of j for which a set named j exists. Initially, PRED[j] $= j - 1$ for $1 \le j \le k+1$ and SUCC[j] $= j + 1$ for $0 \le j \le k$. We then execute the program of Fig. 4.23.

It is easily seen that the execution time of this program is bounded by the running time of the set union algorithm. Hence the off-line MIN problem is $O(nG(n))$ in time complexity.

Example 4.8. Consider the sequence of instructions $\sigma = 4\ 3\ E\ 2\ E\ 1\ E$, where j stands for INSERT(j) and E stands for EXTRACT_MIN. Thus $\sigma_1 = 4\ 3$, $\sigma_2 = 2$, $\sigma_3 = 1$, and σ_4 is the empty sequence. The initial data structure is the sequence of sets

$$1 = \{3, 4\} \quad 2 = \{2\} \quad 3 = \{1\} \quad 4 = \emptyset.$$

In the first execution of the **for** loop, we determine FIND(1) = 3. Thus the answer to the third EXTRACT_MIN instruction in σ is 1. The sequence of sets becomes

$$1 = \{3. 4\} \quad 2 = \{2\} \quad 4 = \{1\}.$$

At this point SUCC[2] is set equal to 4 and PRED[4] is set equal to 2, since sets named 3 and 4 have been merged into a single set named 4.

On execution of the next pass with $i = 2$, we determine that FIND(2) = 2. Thus the answer to the second EXTRACT_MIN instruction s 2. We merge the set named 2 with the successor set (named 4) to obtain he sequence of sets

$$1 = \{3, 4\} \quad 4 = \{1, 2\}.$$

The final two passes determine that the answer to the first EXTRACT_MIN nstruction is 3, and that 4 is never extracted. □

Our next application is to a depth determination problem. One place vhere it arises is in the "equivalencing" of identifiers in an assembly language rogram. Many assembly languages allow statements which declare that two dentifiers represent the same memory location. Should an assembler encounter a statement equivalencing two identifiers α and β, then it must find he two sets S_α and S_β, representing the sets of identifiers equivalent to α and β, respectively, and replace these two sets by their union. Obviously, this problem can be modeled by a sequence of UNION and FIND instructions.

However, if we examine this problem more carefully, we can find another way to apply the data structures of the preceding section. Each identifier has n entry in a symbol table, and if a group of identifiers are equivalent, it is convenient to keep data about them in only one of the symbol table entries. This means that for each set of equivalent identifiers there is an *origin*, a place n the symbol table holding information about the set, and each member has a *isplacement* from the origin. The location in the symbol table for an identifier is found by adding its displacement to the origin of its set. However, when two sets of identifiers are made equivalent, the displacements from the rigin must be modified. Application 2 is an abstraction of this problem of pdating displacements.

Application 2. Depth determination problem

We are given a sequence of two types of instructions: LINK(v, r) and IND_DEPTH(v). We start with n undirected, rooted trees, each consisting of single vertex i, $1 \leq i \leq n$. An instruction LINK(v, r), where r is a root of a ree and v is a vertex in a different tree, results in making the root r a son of v. The conditions that v and r are in different trees and that r is a root insure that he resulting graph is still a forest. The instruction FIND_DEPTH(v) equires determining and printing the current depth of vertex v.

If we maintain the forest using a conventional adjacency list representtion and determine the depth of vertices in the obvious manner, the growth te of the algorithm will be $O(n^2)$. Instead we shall use another forest, hich we shall call the D-forest, to represent the original forest. The sole urpose of the D-forest is to enable us to calculate the depths quickly. Each vertex in the D-forest is assigned an integer weight such that the sum of the weights along a path in the D-forest from a vertex v to the root is the depth of v in the original forest. For each tree in the D-forest a count is kept of the number of vertices in the tree.

Initially, the D-forest consists of n trees, each consisting of a single vertex corresponding to a unique integer i, $1 \le i \le n$. The initial weight of each vertex is zero.

An instruction FIND_DEPTH(v) is executed by following the path from v to the root r. Let $v_1, v_2, \ldots, v_k$ be the vertices on the path ($v_1 = v$, $v_k = r$). Then

$$\text{DEPTH}(v) = \sum_{i=1}^{k} \text{WEIGHT}[v_i].$$

In addition, we do path compression. Each v_i, $1 \le i \le k-2$, is made a son of the root r. To preserve the property of weights, the new WEIGHT of v must become $\Sigma_{j=i}^{k-1}$ WEIGHT[v_j] for $1 \le i < k$. Since the new weights can be computed in $O(k)$ time, the execution time of a FIND_DEPTH instruction is of the same order of magnitude as a FIND instruction.

An instruction LINK(v, r) is executed by combining the trees containing the vertices v and r, again merging the smaller tree into the larger. Let T_v and T_r be the trees in the D-forest containing v and r, respectively. Let v' and r' be the roots of T_v and T_r. In the D-forest, the trees are not necessarily isomorphic to the trees in the original forest, so that, in particular, r may not be the root of T_r. Let COUNT(T) denote the number of vertices in the tree T. There are two cases to consider.

CASE 1. COUNT(T_r) $\le$ COUNT(T_v). We make r' a son of v'. We must also adjust the weight of r', the old root of T_r, so that the depth of each vertex w in T_r will be correctly computed by following the new path from w to v' in the merged tree. To do so, we execute the instruction FIND_DEPTH(v) and then do the following:

$$\text{WEIGHT}[r'] \leftarrow \text{WEIGHT}[r'] - \text{WEIGHT}[v'] + \text{DEPTH}(v) + 1.$$

Thus the depth of each vertex in T_r has been effectively increased by the depth of v plus one. Finally, we make the count of the merged tree equal to the sum of the counts of T_r and T_v.

CASE 2. COUNT(T_v) < COUNT(T_r). Here we execute DEPTH(v), then make v' a son of r' and do the following:

$$\text{WEIGHT}[r'] \leftarrow \text{WEIGHT}[r'] + \text{DEPTH}(v) + 1$$

$$\text{WEIGHT}[v'] \leftarrow \text{WEIGHT}[v'] - \text{WEIGHT}[r']$$

$$\text{COUNT}(T_r) \leftarrow \text{COUNT}(T_v) + \text{COUNT}(T_r)$$

In conclusion, $O(n)$ LINK and FIND_DEPTH instructions can be executed in $O(nG(n))$ time.

Application 3. Equivalence of finite automata

A deterministic finite automaton is a machine that recognizes strings of symbols. It has an input tape ruled into squares, an input tape head, and a finite state control, as shown in Fig. 4.24. We denote a finite automaton M by a 5-tuple (S, I, δ, s_0, F), where:

1. S is the set of states of the *finite control.*
2. I is the *input alphabet.* Each square of the input tape contains a symbol from I.
3. δ is a mapping from $S \times I$ into S. If $\delta(s, a) = s'$, then whenever M is in state s with input symbol a under the input head, M moves its input head right and enters state s'.
4. s_0 is a distinguished state in S called the *start state.*
5. F is a designated subset of S, called the set of *accepting* (or *final*) states.

We let I^* denote the set of all finite-length strings of symbols from I. ncluded in I^* is ϵ, the empty string. We extend the function δ to map $S \times I^*$ to S as follows:

1. $\delta(s, \epsilon) = s$
2. For all x in I^* and a in I, $\delta(s, xa) = \delta(\delta(s, x), a)$.

An input string x is *accepted by M* if $\delta(s_0, x) \in F$. The *language* accepted by M, denoted $L(M)$, is the set of strings accepted by M. Section 9.1 ontains a more extensive introduction to finite automata.

We say two states s_1 and s_2 are *equivalent* if for each x in I^*, $\delta(s_1, x)$ is n accepting state if and only if $\delta(s_2, x)$ is an accepting state.

We say two finite automata M_1 and M_2 are *equivalent* if $L(M_1) = L(M_2)$. n this section we shall show that the UNION-FIND algorithm can be used o determine whether two finite automata $M_1 = (S_1, I, \delta_1, s_1, F_1)$ and $M_2 = S_2, I, \delta_2, s_2, F_2)$ are equivalent in $O(nG(n))$ steps, where $n = \|S_1\| + \|S_2\|$.

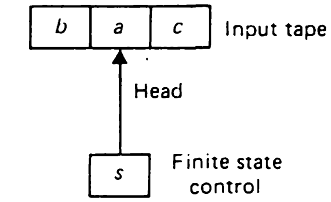


Fig. 4.24. A finite automation.

An important property of state equivalence is that if two states s and are equivalent, then for all input symbols a, $\delta(s, a)$ and $\delta(s', a)$ must also equivalent. Also, no accepting state can be equivalent to a nonaccepti state, thanks to the empty string. Thus if we begin with the assumption th s_1 and s_2, the start states of M_1 and M_2, are equivalent, then we can dedu certain other pairs of states which must also be equivalent. If one of the pairs includes both an accepting and a nonaccepting state, then s_1 and s_2 a not equivalent. While we shall not show it, the converse holds as we

To determine whether two finite automata $M_1 = (S_1, I, \delta_1, s_1, F_1)$ ai $M_2 = (S_2, I, \delta_2, s_2, F_2)$ are equivalent we may proceed as follows:

1. We use the program in Fig. 4.25 to determine all sets of states whi must be equivalent, assuming the two start states s_1 and s_2 are equivaler LIST holds pairs of states (s, s') such that s and s' have been four equivalent but whose successors $(\delta(s, a), \delta(s', a))$ have not yet bee examined. To begin, LIST contains only the pair of start states $(s_1, s_2$ To find the sets of equivalent states, the program uses the disjoint-s union algorithm. COLLECTION represents a family of sets. Initial each state in $S_1 \cup S_2$ is in a set by itself. (Without loss of generality, w can assume the states in S_1 and S_2 are disjoint.) Then whenever tw states s and s' are found to be equivalent, we merge A and A', the se

```
begin
    LIST ← $(s_1, s_2)$;
    COLLECTION ← ∅;
    for each $s$ in $S_1 \cup S_2$ do add $\{s\}$ to COLLECTION;
    comment We have just initialized a set for each state in $S_1 \cup S_2$;
    while there is a pair $(s, s')$ of states on LIST do
        begin
            delete $(s, s')$ from LIST;
            let $A$ and $A'$ be FIND($s$) and FIND($s'$), respectively;
            if $A \neq A'$ then
                begin
                    UNION($A$, $A'$, $A$);
                    for all $a$ in $I$ do
                        add $(\delta(s, a), \delta(s', a))$ to LIST
                end
        end
end
```

Fig. 4.25. Algorithm for finding sets of equivalent states, assuming s_1 and s_2 are equivalent.

containing s and s' in COLLECTION, and call the resulting set A.

2. On completion of the program, the sets in COLLECTION represent a partition of $S_1 \cup S_2$ into blocks of states that must be equivalent. M_1 and M_2 are equivalent if and only if no block contains both an accepting state and a nonaccepting state.

The running time of the algorithm (as a function of $n = \|S_1\| + \|S_2\|$, the number of states) is dominated by the set union algorithm. There can be at most $n - 1$ UNION's since each UNION instruction reduces the number of sets in COLLECTION by one, and there were only n sets to begin with. The number of FIND's is proportional to the number of pairs placed on LIST. This number is at most $n \times \|I\|$ since with the exception of the initial pair (s_1, s_2), a pair is placed on LIST only after a UNION instruction. Thus the time required to determine whether M_1 is equivalent to M_2 is $O(nG(n))$, assuming $\|I\|$ is a constant.

4.9 BALANCED TREE SCHEMES

There are several important classes of problems which are similar to the UNION-FIND problem but which apparently force us to fall back on techniques that require $O(n \log n)$ time (worst case) to process a sequence of n instructions. One such class of problems involves processing a sequence of MEMBER, INSERT, and DELETE instructions when the universe of possible elements is much larger than the number of elements actually used. In this case we cannot access an element by directly indexing into an array of pointers. We must use either hashing or a binary search tree.

If n elements have been inserted, the hashing method has an average access time which is constant but a worst-case time which is $O(n)$ per access. Using a binary search tree gives an expected access time of $O(\log n)$ per access. However, a binary search tree can also give a poor worst-case access time if the set of names is not static. If we simply add names to the tree without some mechanism to keep the tree balanced, we may eventually end up with a tree of n elements which has a depth close to n. Thus the worst-case performance of a binary search tree can be $O(n)$ per operation. The techniques of this section can be used to reduce the worst-case performance to $O(\log n)$ steps per operation.

Another class of problems requiring $O(n \log n)$ time is the on-line processing of sequences of n instructions containing the operations INSERT, DELETE, and MIN. Still a third such class of problems arises if we need to represent ordered lists and have the capability of concatenating and splitting lists.

In this section we shall present techniques that will allow us to process, on-line, sequences containing important subsets of the seven fundamental operations on sets mentioned in Section 4.1. The underlying data structure is a *balanced tree*, by which we mean a tree whose height is approximately equal to the logarithm of the number of vertices in the tree.

A balanced tree is easy to construct initially. However, the problem is to prevent the tree from becoming unbalanced while executing a sequence of INSERT and DELETE commands. For example, if we execute a sequence of DELETE commands that remove vertices from only the left side of the tree, we shall end up with a tree that is unbalanced to the right unless we periodically rebalance the tree.

There are a number of mechanisms that can be used to rebalance the tree when necessary. Several of these methods leave the structure of the tree flexible enough so that the number of vertices in a tree of height h can range anywhere from 2^h to 2^{h+1} or 3^h. Such methods allow us to at least double the number of vertices in a subtree before having to make any change above the root of a subtree.

Two specific methods of this nature which we shall discuss are 2–3 trees and AVL trees. The algorithms for manipulating 2–3 trees are conceptually easier to understand and these we shall discuss in the text. The algorithms for manipulating AVL trees are similar to those for 2–3 trees and are found in the exercises.

Definition. A *2–3 tree* is a tree in which each vertex which is not a leaf has 2 or 3 sons, and every path from the root to a leaf is of the same length. Note that the tree consisting of a single vertex is a 2–3 tree. Figure 4.26 shows the two 2–3 trees with six leaves.

The following lemma gives the relationship between the number of vertices and leaves in a 2–3 tree and its height.

Lemma 4.6. Let T be a 2–3 tree of height h. The number of vertices of T is between $2^{h+1} - 1$ and $(3^{h+1} - 1)/2$, and the number of leaves is between 2^h and 3^h.

Proof. Elementary induction on h. □

A linearly ordered set S can be represented by a 2–3 tree by assigning the elements of the set to the leaves of the tree. We shall use $E[l]$ to denote the element stored at leaf l. There are two basic methods of assigning the elements to the leaves; which method is used depends on the application.

If the universe of possible elements is much larger than the actual number of elements used and the tree is to be used as a dictionary, then we shall probably want to assign the elements in increasing order from left to right. At each vertex v which is not a leaf, we need two additional pieces of information, $L[v]$ and $M[v]$. $L[v]$ is the largest element of S assigned to the subtree whose root is the leftmost son of v; $M[v]$ is the largest element of S

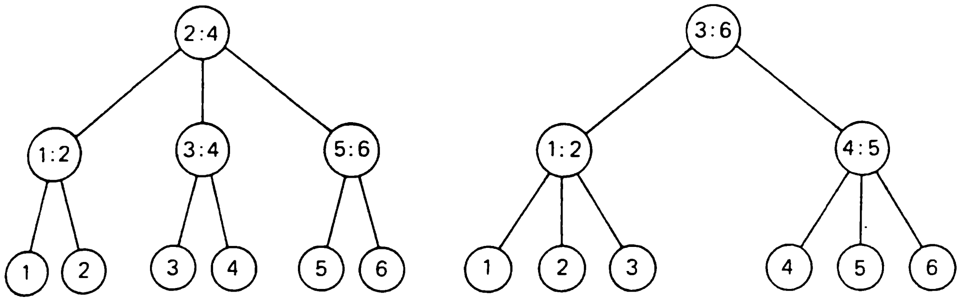


Fig. 4.26 2–3 trees.

ssigned to the subtree whose root is the second son of v. See Fig. 4.26. he values of L and M attached to the vertices enable us to start at the root nd search for an element in a manner analogous to binary search. The time find any element is proportional to the height of the tree. Thus a IEMBER instruction on a set with n elements can be processed in time (log n) if a 2–3 tree of this nature is used to represent the set.

The second method of assigning elements to leaves is to place no striction on the order in which the elements are assigned. This method is articularly useful in implementing the instruction UNION. However, to excute instructions such as DELETE a mechanism is needed to locate the leaf presenting a given element. If the elements of the sets are integers in some ed range, say 1 to n, the leaf representing element i can be found via the ith cation of some array. If the elements of the sets are from some large unirsal set, then the leaf representing element i can be found via an auxiliary ctionary.

Consider the following sets of instructions.

. INSERT, DELETE, MEMBER
. INSERT, DELETE, MIN
. INSERT, DELETE, UNION, MIN
. INSERT, DELETE, FIND, CONCATENATE, SPLIT

e shall call a data structure that can process instructions from set 1 a *dicnary*, from set 2 a *priority queue*, from set 3 a *mergeable heap*, and from 4 a *concatenable queue*.

We shall show that 2–3 trees can be used to implement dictionaries, prity queues, concatenable queues, and mergeable heaps with which we n process n instructions in time $O(n \log n)$. The techniques used are sufently powerful to execute sequences of any compatible subset of the seven tructions listed at the beginning of the chapter. The only incompatibility is t UNION implies an unordered set, and SPLIT and CONCATENATE ly order.

4.10 DICTIONARIES AND PRIORITY QUEUES

In this section we shall consider the basic operations required to implement dictionaries and priority queues. Throughout this section we shall assume that elements are assigned to the leaves of a 2–3 tree in left-to-right order, and $L[v]$ and $M[v]$ (the "largest" descendant functions described in the previous section) are present at each nonleaf v.

To insert a new element a into a 2–3 tree we must locate the position for the new leaf l that will contain a. This is done by trying to locate element a in the tree. Assuming the tree contains more than one element, the search for a terminates at a vertex f such that f has either two or three leaves as sons.

If f has only two leaves l_1 and l_2, we make l a son of f. If $a < E[l_1]$,† we make l the leftmost son of f and set $L[f] = a$ and $M[f] = E[l_1]$; if $E[l_1] < a < E[l_2]$, we make l the middle son of f and set $M[f] = a$; if $E[l_2] < a$, we make l the third son of f. The L or M values of some proper ancestors of f may have to be changed in the latter case.

Example 4.9. If we insert the element 2 into the 2–3 tree of Fig. 4.27(a), we get the 2–3 tree of Fig. 4.27(b). □

Now suppose f already has three leaves, l_1, l_2, and l_3. We make l the appropriate son of f. Vertex f now has four sons. To maintain the 2–3 property, we create a new vertex g. We keep the two leftmost sons as sons of f, but change the two rightmost sons into sons of g. We then make g a brother of vertex f by making g a son of the father of f. If the father of f had two sons, we stop here. If the father of f had three sons, we must repeat this procedure recursively until all vertices in the tree have at most three sons. If the root is given four sons, we create a new root with two new sons, each of which has two of the four sons of the former root.

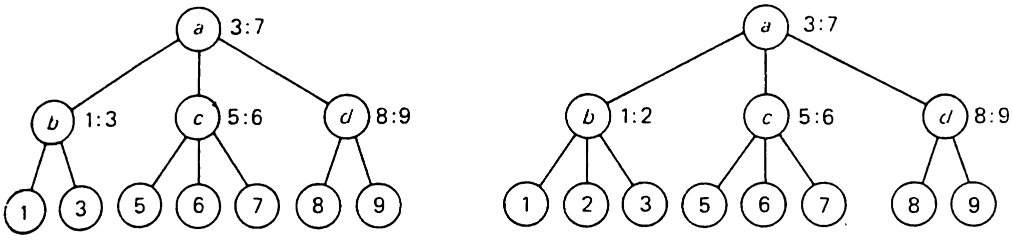
Example 4.10. If we insert element 4 into the 2–3 tree of Fig. 4.27(a), we find that the new leaf labeled 4 should be made the leftmost son of vertex c. Since vertex c already has three sons, we create a new vertex c'. We make leaves 4 and 5 sons of c, and leaves 6 and 7 sons of c'. We now make c' a son of vertex a. However, since vertex a already has three sons, we create another vertex a'. We make vertices b and c sons of the old vertex a, and vertices c' and d sons of the new vertex a'. Finally, we create a new root r and make a and a' sons of r. The resulting tree is shown in Fig. 4.28. □

Algorithm 4.4. Insertion of a new element into a 2–3 tree.

Input. A nonempty 2–3 tree T with root r and a new element a not in T.

Output. A revised 2–3 tree with a new leaf labeled a.

† $E[v]$ is the element stored at leaf v.



(a) (b)

Fig. 4.27 Insertion into a 2–3 tree: (a) tree before insertion; (b) tree after inserting 2.

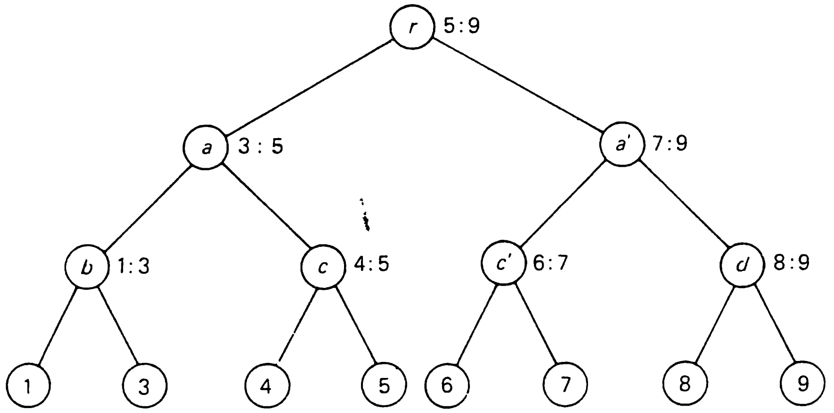


Fig. 4.28 Tree of Fig. 4.27(a), after inserting 4.

Method. We assume T has at least one element. To simplify description of the algorithm we have omitted the details of updating L and M at various vertices.

1. If T consists of a single leaf l labeled b, then create a new root r'. Create a new leaf v labeled a. Make l and v sons of r', making l the left son if $b < a$, otherwise, making l the right son.
2. If T has more than one vertex, let $f =$ SEARCH(a, r), where SEARCH is the procedure in Fig. 4.29. Create a new leaf l labeled a. If f has two sons labeled b_1 and b_2, then make l the appropriate son of f. Make l the left son if $a < b_1$, the middle son if $b_1 < a < b_2$, the right son if $b_2 < a$. If f has three sons, make l the appropriate son of f and then call ADDSON(f) to incorporate f and its four sons into T. ADDSON is the procedure in Fig. 4.30. Adjust the values of L and M along the path

```
procedure SEARCH($a$, $r$):
if any son of $r$ is a leaf then return $r$
else
    begin
        let $s_i$ be the $i$th son of $r$;
        if $a \leq L[r]$ then return SEARCH($a$, $s_1$)
        else
            if $r$ has two sons or $a \leq M[r]$ then return SEARCH($a$, $s_2$)
            else return SEARCH($a$, $s_3$)
    end
```

Fig. 4.29. Procedure SEARCH.

```
procedure ADDSON($v$):
begin
    create a new vertex $v'$;
    make the two rightmost sons of $v$ the left and right sons of $v'$;
    if $v$ has no father then
        begin
            create a new root $r$;
            make $v$ the left son and $v'$ the right son of $r$
        end
    else
        begin
            let $f$ be the father of $v$;
            make $v'$ a son of $f$ immediately to the right of $v$;
            if $f$ now has four sons then ADDSON($f$)
        end
end
```

Fig. 4.30. Procedure ADDSON.

from a to the root to account for the presence of a.† Other obvious adjustments to L and M values are made by ADDSON (these are omitted and left as an exercise). □

Theorem 4.6. Algorithm 4.4 inserts a new element into a 2–3 tree with n leaves in at most $O(\log n)$ time. Moreover, the algorithm maintains the order of the original leaves and retains the 2–3 tree structure.

† We need only follow the path from a until we reach a vertex such that a is not the largest element in its subtree.

Proof. A straightforward induction on the number of calls of SEARCH shows that the new leaf is made a son of the correct vertex. The ordering of the original leaves is not disturbed. For the timing result we noted by Lemma 4.6 that the height of a 2–3 tree with n leaves is at most $\log n$. Since ADDSON(v) calls itself recursively only on the father of v, at most $\log n$ recursive calls are possible. Since each call of ADDSON requires only a constant amount of time, the total time is at most $O(\log n)$. □

An element a can be deleted from a 2–3 tree in essentially the reverse of the manner by which an element is inserted. Suppose element a is the label of leaf l. There are three cases to consider.

CASE 1. If l is the root, remove l. (In this case, a was the only element in the tree.)

CASE 2. If l is the son of a vertex having three sons, remove l.

CASE 3. If l is the son of a vertex f having two sons s and l, then there are two possibilities:

a) f is the root. Remove l and f, and leave the remaining son s as the root.
b) f is not the root. Suppose f has a brother[†] g to its left. A brother to the right is handled similarly. If g has only two sons, make s the rightmost son of g, remove l, and call the deletion procedure recursively to delete f. If g has three sons, make the rightmost son of g be the left son of f and remove l from the tree.

Example 4.11. Let us delete element 4 from the 2–3 tree of Fig. 4.28. The leaf labeled 4 is a son of vertex c which has two sons. Thus we make the leaf labeled 5 the rightmost son of vertex b, remove the leaf labeled 4, and then recursively remove vertex c.

Vertex c is a son of vertex a, which has two sons. Vertex a' is the right brother of a. Thus, by symmetry, we make b the leftmost son of a', remove vertex c, and then recursively remove vertex a.

Vertex a is a son of the root. Applying case 3(a), we leave a' as the root of the remaining tree, which is shown in Fig. 4.31. □

We leave a formal specification of the deletion process as an exercise, along with the proof that it can be executed in at most $O(\log n)$ steps on a 2–3 ree with n leaves.

We have now seen that a MEMBER, INSERT, or DELETE instruction can be executed in at most $O(\log n)$ steps on a 2–3 tree with n leaves. Thus, a 2–3 tree can be used as an $O(n \log n)$ dictionary since it can process a equence of n MEMBER, INSERT, and DELETE instructions in at most $O(n \log n)$ steps.

Two vertices with the same father are called *brothers*.

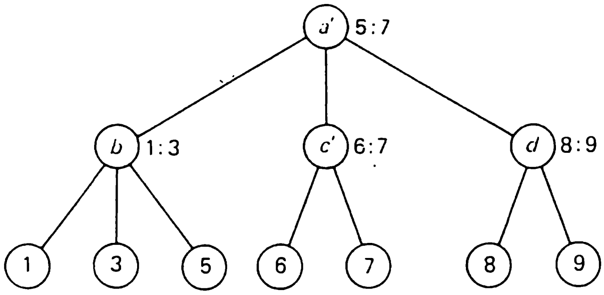


Fig. 4.31 Tree of Fig. 4.28, after removing 4.

Let us now consider the MIN instruction. The smallest element in a 2–3 tree is located at the leftmost leaf, which can certainly be found in $O(\log n)$ steps. Therefore any sequence of n INSERT, DELETE, and MIN instructions can be processed in $O(n \log n)$ time using a 2–3 tree. We have thus verified our claim that a 2–3 tree can be used to implement an $O(n \log n)$ priority queue. Two other data structures that can be used to implement an $O(n \log n)$ priority queue are the heap used in Heapsort and the AVL tree discussed in Exercises 4.30–4.33.

4.11 MERGEABLE HEAPS

In this section we present a data structure that can be used to process sequences of n INSERT, DELETE, UNION, and MIN instructions in time $O(n \log n)$. The structure, which can be thought of as a generalization of the heap discussed in Section 3.4, uses a 2–3 tree T to represent a set of elements S. Each element of S appears as the label of a leaf of T, but the leaves are not linearly ordered as in the previous two sections. To each interior vertex of T we attach a label SMALLEST[v] which indicates the value of the smallest element stored in the subtree with root v. $L[v]$ and $M[v]$ are not needed in this application of 2–3 trees.

The smallest element in the set S can be found by starting at the root of T and walking down the tree as follows. If we are at an interior vertex v, we next visit the son of v with the lowest value of SMALLEST. Thus if T has n leaves, the instruction MIN requires $O(\log n)$ steps.

In many applications whenever we delete an element from S, it is always the smallest. However, if we want to delete an arbitrary element from S, we must be able to find the leaf containing that element. In applications where the elements can be represented by the integers $1, 2, \ldots, n$, we can index the leaves directly. If the elements are arbitrary, we can use an auxiliary 2–3 dictionary whose leaves contain pointers to the leaves of T. Via this dictionary an arbitrary leaf of T can be accessed in $O(\log n)$ steps. The dic-

```
procedure IMPLANT($T_1$, $T_2$):
if HEIGHT($T_1$) = HEIGHT($T_2$) then
    begin
        create a new root $r$;
        make ROOT[$T_1$] and ROOT[$T_2$] the left and right sons of $r$
    end
else
    wlg assume HEIGHT($T_1$) > HEIGHT($T_2$) otherwise
      interchange $T_1$ and $T_2$ and interchange "left" and "right" in
        begin
            let $v$ be the vertex on the rightmost path of $T_1$ such that
                DEPTH($v$) = HEIGHT($T_1$) − HEIGHT($T_2$);
            let $f$ be the father of $v$;
            make ROOT[$T_2$] a son of $f$ immediately to the right of $v$;
            if $f$ now has four sons then ADDSON($f$)†
        end
```

† If we wish to have L and M values for the new vertex which ADDSON(f) will create, we must first find the maximum descendant of v by following the path to the rightmost leaf.

Fig. 4.32. Procedure IMPLANT.

tionary must be updated each time an INSERT instruction is executed but this requires at most $O(\log n)$ steps.

Once we have deleted a leaf l from T, we have to recompute the values of SMALLEST for each proper ancestor v of l. The new value of SMALLEST[v] will be the minimum of SMALLEST[s] over the two or three sons s of v. If we always do the recomputation bottom-up, we can show, by induction on the number of recomputations, that each computation produces the correct answer for SMALLEST. Since the only SMALLEST values that change are at ancestors of the deleted leaf, a DELETE instruction can be processed in $O(\log n)$ steps.

Let us now consider the UNION instruction. Each set is represented by a distinct 2–3 tree. To merge two sets S_1 and S_2 we call the procedure IMPLANT(T_1, T_2) of Fig. 4.32, where T_1 and T_2 are the 2–3 trees representing S_1 and S_2.‡

Let us suppose that h_1, the height of T_1, is greater than or equal to h_2, the height of T_2. IMPLANT finds, on the rightmost path in T_1, that vertex v

‡ The distinction between "left" and "right" is not important here but is made for the sake of concatenable queues which are discussed in the next section.

which is of height h_2 and makes the root of T_2 the rightmost brother of v. Should f, the father of v, now have four sons, IMPLANT calls the procedure ADDSON(f). The values of SMALLEST for the vertices whose descendants change during the IMPLANT operation can be updated in the same manner as in a DELETE operation.

We leave it as an exercise to show that the procedure IMPLANT combines T_1 and T_2 into a single 2–3 tree in time $O(h_1 - h_2)$. When we count the time to update the L and M values, then the procedure IMPLANT can use $O(\text{MAX}(\log \|S_1\|, \log \|S_2\|))$ time.

Let us consider an application in which the operations UNION, MIN, and DELETE arise naturally.

Example 4.12. We discussed one algorithm for finding minimum-cost spanning trees in Example 4.1 (p. 109). There, vertices were collected into larger and larger sets such that the members of each set were connected by edges already selected for the minimum-cost spanning tree. The strategy for finding new edges for the spanning tree was to consider edges (shortest ones first) and to see whether they connected vertices not yet connected.

Another strategy is to keep, for each set of vertices V_i, the set E_i of all unconsidered edges incident upon some vertex in V_i. If we select a previously unconsidered edge e that is incident upon a vertex in a relatively small set V_i, then there is a high probability that the other end of e is not in V_i, and we can add e to the spanning tree. If e is the minimum-cost unconsidered edge incident upon a vertex in V_i, then it can be shown that adding e to the spanning tree will lead to a minimum-cost spanning tree.

To implement this algorithm we must initially form for each vertex the set of incident edges. To find a minimum-cost unconsidered edge incident upon a set of vertices V_i, we apply the MIN operator to E_i, the set of unconsidered edges for V_i. Then we delete from E_i the edge e so found. If e turns out to have only one end in V_i and the other in a distinct set of vertices V_i', then we UNION V_i and V_i' (which can be done using the data structure of Algorithm 4.3) and also UNION E_i with E_i'.

One data structure which can be used to represent each set of edges E_i is a 2–3 tree with each leaf labeled by an edge and its cost. The edges are in no particular order. Each nonleaf v has attached to it the smallest cost of any of its descendant leaves, denoted SMALLEST[v].

Initially, we create for each vertex a 2–3 tree containing each edge incident upon that vertex. To build such a tree we start by creating the leaves. Then we add the vertices of height 1 by combining leaves into groups of two or three, with at most two groups of two. As we do so, we calculate the minimum cost of a descendant leaf for each vertex at height 1. We then group the height 1 vertices into groups of two or three, and continue until at some level only one vertex, the root, is created. The time taken to construct a tree in this manner is proportional to the number of leaves. The implementation of the remainder of the algorithm should now be straightforward. The overall running time of the algorithm is $O(e \log e)$ where e is the total number of edges. □

4.12 CONCATENABLE QUEUES

We have seen in Section 4.10 how each of the instructions INSERT, DELETE, MIN and MEMBER can be executed on a 2–3 tree with n leaves in at most $O(\log n)$ steps per instruction when the L and M values are used. We shall now show how each of the instructions CONCATENATE and SPLIT can also be executed in $O(\log n)$ time. Again we are assuming the elements appear on the leaves of a 2–3 tree in ascending order from left to right and $L[v]$ and $M[v]$ are computed for each vertex v.

The instruction CONCATENATE(S_1, S_2) takes as input two sequences S_1 and S_2 such that every element of S_1 is less than every element of S_2, and produces as output the concatenated sequence S_1S_2. If S_1 is represented as a 2–3 tree T_1 and S_2 as a 2–3 tree T_2, then we want to combine T_1 and T_2 into a single 2–3 tree T having, as leaves, the leaves of T_1 in their original order followed by the leaves of T_2 in their original order. We can do so by calling IMPLANT(T_1, T_2) of Fig. 4.32 (p. 153).

Let us consider as a final operation the SPLIT instruction. Recall that the operation SPLIT(a, S) partitions S into two sets $S_1 = \{b | b \leq a \text{ and } b \in S\}$ and $S_2 = \{b | b > a \text{ and } b \in S\}$. To implement this instruction using 2–3 trees

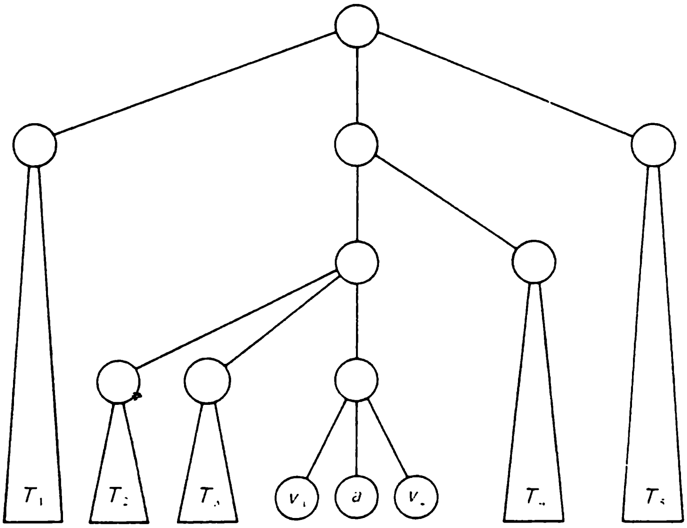


Fig. 4.33 Splitting a 2–3 tree.

we shall define a procedure DIVIDE(a, T) to split a 2–3 tree T into two 2–3 trees T_1 and T_2 such that all leaves in T_1 have labels less than or equal to a, and all leaves in T_2 have labels greater than a.

The method can be described informally as follows. Given a 2–3 tree T containing element a, we follow the path from the root to the leaf labeled a. This path divides the tree into a collection of subtrees, whose roots are sons of vertices on the path, but are not themselves on the path, as shown in Fig. 4.33. There, the trees to the left of the path are T_1, T_2, T_3, and the trivial tree consisting of vertex v_1. The trees to the right are T_4, T_5, and v_2.

The trees to the left of the path plus the tree consisting of a alone are combined using the tree-concatenating algorithm just described. Similarly the trees to the right of the path are combined. The procedure DIVIDE given in Fig. 4.34 contains the details.

Theorem 4.7. The procedure DIVIDE partitions a 2–3 tree T about a leaf a so that all leaves to the left of a and a itself are in one 2–3 tree and all leaves to the right of a are in a second 2–3 tree. The procedure takes time $O(\text{HEIGHT}(T))$. The order of the leaves is preserved.

Proof. That the trees are properly reassembled follows from the properties of the procedure IMPLANT. The timing result is obtained by the following observations. Initially there are at most two trees of any given height, with the exception that there can be three trees of height 0. When two trees are combined, the resulting tree can have height at most one greater than the maximum of the heights of the two original trees. In the case where the resulting tree is of height one greater than either of the original trees, its root is of degree 2. Thus if three trees of height h are combined the resulting tree is of height at most $h + 1$. Hence at each stage in the recombination process, there are at most three trees of the same height.

Since the time required to combine two trees of different height is proportional to the difference of their heights and the time required to combine two trees of the same height is constant, the time to recombine all trees is proportional to number of trees plus the maximum difference in the heights of any two trees. Thus the total time spent is on the order of the height of the original tree. □

We observe that with a concatenable queue we can insert a sequence S_2 between a pair of elements in a sequence S_1 in time $O(\text{MAX}(\log|S_1|, \log|S_2|))$. If $S_2 = b_1, b_2, \ldots, b_n$ and $S_1 = a_1, a_2, \ldots, a_m$, and if S_2 is to be inserted between elements a_i and a_{i+1}, then we can use the instruction SPLIT(a_i, S_1) to partition S_1 about a_i into two sequences $S_1' = a_1, \ldots, a_i$ and $S_1'' = a_{i+1}, \ldots, a_m$. We then use CONCATENATE(S_1', S_2) to get the sequence $S_3 = a_1, \ldots, a_i, b_1, \ldots, b_n$, and finally CONCATENATE(S_3, S_1'') to get the desired sequence.

```
rocedure DIVIDE($a$, $T$):
egin
    on the path from ROOT[$T$] to the leaf labeled $a$ remove all vertices ex-
      cept the leaf;
    comment At this point $T$ has been divided into two forests—the *left
      forest*, which consists of all trees with leaves to the left of and including
      the leaf labeled $a$, and the *right forest*, which consists of all trees with
      leaves to the right of $a$;
    while there is more than one tree in the left forest do
        begin
            let $T'$ and $T''$ be the two rightmost trees in the left forest;
            IMPLANT($T'$, $T''$)†
        end;
    while there is more than one tree in the right forest do
        begin
            let $T'$ and $T''$ be the two leftmost trees in the right forest;
            IMPLANT($T'$, $T''$)
        end
l
```

he result of IMPLANT(T', T'') should be considered as remaining in the left
st. Similarly, when applied to trees in the right forest, the result of IMPLANT is
ee in the right forest.

Fig. 4.34. Procedure to split a 2–3 tree.

PARTITIONING

now consider a special kind of set splitting, called *partitioning*. The
lem of partitioning a set occurs frequently, and the solution we present
is instructive in its own right. Suppose we are given a set S, and an ini-
artition π of S into disjoint blocks $\{B_1, B_2, \ldots, B_p\}$. We are also given
ction f on S. Our task is to find the *coarsest* (having fewest blocks) par-
of S, say $\pi' = \{E_1, E_2, \ldots, E_q\}$, such that:

π' is consistent with π (that is, each E_i is a subset of some B_j), and
a and b in E_i implies $f(a)$ and $f(b)$ are in some one E_j.

We shall call π' the *coarsest partition of S compatible with π and f*.

The obvious solution is to repeatedly refine the blocks of the original par-
by the following method. Let B_i be a block. Examine $f(a)$ for each a
B_i is then partitioned so that two elements a and b are put in the same
if and only if $f(a)$ and $f(b)$ are both in some block B_j. The process is
ed until no further refinements are possible. This method yields an

$O(n^2)$ algorithm, since each refinement requires time $O(n)$ and there can be $O(n)$ refinements. That the method can actually require a quadratic number of steps is illustrated by Example 4.13.

Example 4.13. Let $S = \{1, 2, \ldots, n\}$ and let $B_1 = \{1, 2, \ldots, n-1\}$, $B_2 = \{n\}$ be the original partition. Let f be the function on S such that $f(i) = i + 1$ for $1 \leq i < n$ and $f(n) = n$. On the first iteration we partition B_1 into $\{1, 2, \ldots, n-2\}$ and $\{n-1\}$. This iteration requires $n - 1$ steps since we must examine each element in B_1. On the next iteration we partition $\{1, 2, \ldots, n-2\}$ into $\{1, 2, \ldots, n-3\}$ and $\{n-2\}$. Proceeding in this manner, we need a total of $n - 2$ iterations, with the ith iteration taking $n - i$ steps. Thus a total of

$$\sum_{i=1}^{n-2} (n-i) = \frac{n(n-1)}{2} - 1$$

steps are required. The final partition has $E_i = \{i\}$, for $1 \leq i \leq n$. □

The difficulty with this method is that refining a block may require $O(n)$ steps even if only a single element is removed from the block. Here we shall develop a partitioning algorithm that in refining a block into two subblocks requires time proportional to the smaller subblock. This approach results in an $O(n \log n)$ algorithm.

For each $B \subseteq S$, let $f^{-1}(B) = \{b | f(b) \in B\}$. Instead of partitioning a block B_i by the values of $f(a)$ for $a \in B_i$, we partition with respect to B_i those blocks B_j which contain at least one element in $f^{-1}(B_i)$ and one element not in $f^{-1}(B_i)$. That is, each such B_j is partitioned into sets $\{b | b \in B_j \text{ and } f(b) \in B_i\}$ and $\{b | b \in B_j \text{ and } f(b) \notin B_i\}$.

Once we have partitioned with respect to B_i, we need not partition with respect to B_i again unless B_i itself is split. If initially $f(b) \in B_i$ for each element $b \in B_j$, and B_i is split into B_i' and B_i'', then we can partition B_j with respect to either of B_i' or B_i'' and we will get the same result since $\{b | b \in B_j \text{ and } f(b) \in B_i'\}$ is identical to $B_j - \{b | b \in B_j \text{ and } f(b) \in B_i''\}$.

Since we have our choice of partitioning with respect to either B_i' or B_i'', we partition with respect to the easier one. That is, we partition using the smaller of $f^{-1}(B_i')$ and $f^{-1}(B_i'')$. The algorithm is given in Fig. 4.35.

Algorithm 4.5. Partitioning.

Input. A set of n elements S, an initial partition $\pi = \{B[1], \ldots, B[p]\}$, and a function $f: S \rightarrow S$.

Output. A partition $\pi' = \{B[1], B[2], \ldots, B[q]\}$ such that π' is the coarsest partition of S compatible with π and f.

Method. We apply the program given in Fig. 4.35 to π. This program omits certain important implementation details. We discuss those details later, when the running time is analyzed. □

```
      begin
1.        WAITING $\leftarrow \{1, 2, \ldots, p\}$;
2.        $q \leftarrow p$;
3.        while WAITING not empty do
              begin
4.                select and delete any integer $i$ from WAITING;
5.                INVERSE $\leftarrow f^{-1}(B[i])$;
6.                for each $j$ such that $B[j] \cap$ INVERSE $\neq \emptyset$ and
                    $B[j] \not\subseteq$ INVERSE do
                      begin
7.                        $q \leftarrow q + 1$;
8.                        create a new block $B[q]$;
9.                        $B[q] \leftarrow B[j] \cap$ INVERSE;
10.                       $B[j] \leftarrow B[j] - B[q]$;
11.                       if $j$ is in WAITING then add $q$ to WAITING
                          else
12.                           if $\|B[j]\| \leq \|B[q]\|$ then
13.                               add $j$ to WAITING
14.                           else add $q$ to WAITING
                      end
              end
      end
```

Fig. 4.35. Partitioning algorithm.

Let us begin the analysis of Algorithm 4.5 by proving that it partitions S correctly. Let π be any partition of the set S and f a function on S. We say set $T \subseteq S$ is *safe for* π if for every block B in π, either $B \subseteq f^{-1}(T)$ or $B \cap f^{-1}(T) = 0$. For example, in Fig. 4.35 lines 9 and 10 assure that $B[i]$ is safe for the resulting partition, since if $B \cap f^{-1}(B[i]) \neq \emptyset$ for some block B, then either $B \subseteq$ INVERSE, in which case $B \subseteq f^{-1}(B[i])$ is immediate, or at lines 9 and 10, B is split into two blocks, one of which is a subset of $f^{-1}(B[i])$ and the other of which is disjoint from that set.

Part of the work involved in showing Algorithm 4.5 correct is proving that the final partition is not too coarse. That is, we must show the following.

Lemma 4.7. After Algorithm 4.5 terminates, every block B in the resulting partition π' is safe for π'.

Proof. What we shall actually show for each block $B[l]$ is:

> After every execution of the loop of lines 4–14 in Fig. 4.35, if l is not in WAITING, and $l \leq q$, then there is some list $q_1, q_2, \ldots, q_k$ (possibly empty) such that each q_i is on WAITING, $1 \leq i \leq k$, and $B[l] \cup B[q_1] \cup \cdots \cup B[q_k]$ is safe for the current partition. (4.6)

Intuitively, when a block $B[l]$ is removed from WAITING, lines 6–14 make $B[l]$ safe for the partition in effect after line 14. $B[l]$ remains safe until it is partitioned. When $B[l]$ is partitioned, one subblock, call it $B[q]$, is placed on WAITING. The other subblock is still called $B[l]$. Clearly, the union of these two subblocks, $B[l] \cup B[q]$, is safe since it is the old $B[l]$. Further partitioning creates blocks $B[q_1], \ldots, B[q_k]$ such that $q_1, \ldots, q_k$ are on WAITING and $R = B[l] \cup B[q_1] \cup \cdots \cup B[q_k]$ is safe. When some q_i, $1 \leq i \leq k$, is removed from WAITING, lines 6–14 again make $B[q_i]$ and $R - B[q_i]$ safe. These ideas are made precise below.

We prove (4.6) for all l by induction on the number of times we have executed lines 4–14. When the algorithm terminates, WAITING is empty and hence (4.6) implies that each block of the final partition π' is safe for π'.

For the basis, we take 0 times, whereupon (4.6) is trivial, since l is on WAITING for all $1 \leq l \leq q = p$.

For the inductive step, suppose l is not on WAITING after finishing line 14. If l was on WAITING the previous time through, then l has the value i which was defined at line 4. It is easy to show that the loop of lines 6–14 makes $B[i]$ safe for the partition in effect after line 14. We argued this statement after the definition of "safe."

If l was not on WAITING the previous time through line 14, then by the inductive hypothesis, there is some list $q_1, \ldots, q_k$ such that (4.6) was satisfied for l at the previous stage. Also, we can be sure that $i \neq l$ at line 4.

CASE 1. i is not among $L = \{q_1, q_2, \ldots, q_k\}$. Several blocks may be split at lines 9–10. For each such block $B[q_r]$, $1 \leq r \leq k$, (that is, $j = q_r$) add the subscript of the block created at line 8 to L. By line 11, L will continue to consist only of subscripts on WAITING. If $B[l]$ itself is not split, then $B[l]$ and the set of blocks on L continue to form a set that is safe for the current partition, satisfying (4.6). If $B[l]$ is split, we must also add to L that index q selected at line 8 when $j = l$. Hence $B[l] \cup \bigcup_{r \in L} B[r]$ will be safe for the current partition.

CASE 2. i is among $L = \{q_1, q_2, \ldots, q_k\}$. Assume without loss of generality that $i = q_1$. The argument proceeds almost as in case 1, but q_1 may not be on WAITING at the end of the current iteration of lines 4–14. However, we know that every block B of the current partition will be either a subset of $f^{-1}(B[q_1])$ or disjoint from it. Let $T = B[l] \cup \bigcup_{r \in L} B[r]$, where L has been modified as in case 1. If $B \subseteq f^{-1}(B[q_1])$, then surely $B \cap f^{-1}(T) = \emptyset$. If $B \cap f^{-1}(B[q_1]) = \emptyset$, then the argument $B \cap f^{-1}(T) = \emptyset$ or $B \subseteq f^{-1}(T)$ is analagous to case 1.

Finally, when Algorithm 4.5 ends, WAITING must be empty. Thus (4.6) implies that for each l, $B[l]$ is safe for the final partition. □

Theorem 4.8. Algorithm 4.5 correctly computes the coarsest partition of S compatible with π and f.

Proof. Lemma 4.7 shows that the output π' of Algorithm 4.5 is compatible with π and f. We must show that π' is as coarse as possible. A simple induction on the number of blocks split at lines 9–10 proves that every such split made by the algorithm is necessary for compatibility. We leave this induction as an exercise. □

We must now consider in detail the implementation of Algorithm 4.5 in order to show its running time to be $O(n \log n)$, where $n = \|S\|$. The crux of the timing argument is to show how the loop of lines 6–14 can be executed in time proportional to ||INVERSE||. Our first problem is how to find efficiently the appropriate set of j's at line 6. We need an array INBLOCK such that INBLOCK$[l]$ is the index of the block containing l. INBLOCK can be initialized in $O(n)$ steps and updated after line 9 in no more time than it takes to create the list $B[q]$ at that line. Thus, since we care only about the order of magnitude of the time complexity, we are justified in ignoring the handling of INBLOCK.

Using INBLOCK, it is easy to construct a list JLIST of the j's needed in line 6 in O(||INVERSE||) steps. For each element a of INVERSE, add the index of the block containing a to JLIST if it is not already there. For each j, a count is kept of the number of elements in INVERSE which are also in $B[j]$. If the count reaches $\|B[j]\|$, then $B[j] \subseteq$ INVERSE, and j is deleted from JLIST.

Also using INBLOCK, we can make, for each j on JLIST, a list INTERSECTION$[j]$ of the integers in the intersection of $B[j]$ and INVERSE. To rapidly delete the elements of INTERSECTION$[j]$ from $B[j]$ and add them to $B[q]$, we must maintain the lists $B[l]$, $1 \leq l \leq q$ in doubly linked form, i.e., with pointers to both successors and predecessors.

Lines 9 and 10 require $O(\|B[q]\|)$ steps. For a given execution of the **for** loop, the aggregate time spent on finding the proper j's and lines 7–10 is O(||INVERSE||). Also, the test of lines 12–14 is easily seen to require $O(\|B[q]\|)$ time if done properly, for an aggregate of O(||INVERSE||) time.

We need only consider line 11. To tell whether j is on list WAITING quickly, we create another array INWAITING$[j]$. INWAITING can be initialized in $O(n)$ steps and maintained without difficulty at lines 11–14. We thus have the following lemma.

Lemma 4.8. The **for** loop of lines 6–14 in Fig. 4.35 can be implemented in O(||INVERSE||) steps.

Proof. By the above. □

Theorem 4.9. Algorithm 4.5 can be implemented in $O(n \log n)$ time.

Proof. Let us consider the circumstances under which an integer s, which was in a block not on WAITING, can find its block put on WAITING. This can happen once at line 1. It cannot happen at line 11 even if $s \in B[q]$, since s would have formerly been in $B[j]$ and j was already on WAITING. If it happens at line 13 or 14, then s is in a block no more than half the size of the block containing s the previous time the index of the set containing s was put on WAITING. We may conclude that the index of the set containing s is not put on WAITING more than $1 + \log n$ times. Consequently, s cannot be in the block i chosen at line 4 more than $1 + \log n$ times.

Suppose the cost for each execution of the loop of lines 6–14 is charged to element s of $B[i]$ in proportion to $\|f^{-1}(s)\|$. Then there is a constant c such that s is charged no more than $c\|f^{-1}(s)\|$ for this execution of the loop. But we have previously argued that s cannot be in the selected $B[i]$ more than $O(\log n)$ times, so its total charge is $O(\|f^{-1}(s)\| \times \log n)$. Since the sum $\Sigma_{s \in S}\|f^{-1}(s)\|$ must be n, the aggregate cost of all executions of the **for** loop is $O(n \log n)$. The cost of the remainder of Algorithm 4.5 is easily seen to be $O(n)$, so we have our theorem. □

Algorithm 4.5 has several applications. One important application is in minimizing the number of states in a finite automaton. We are given a finite automaton $M = (S, I, \delta, s_0, F)$, and we want to find that automaton M' with the minimum number of states which is equivalent to M. For each state s and input symbol a, $\delta(s, a)$ denotes the next state of M. The states of M can be initially partitioned into F, the set of accepting states, and $S - F$, the set of nonaccepting states. The problem of minimizing states in M is equivalent to finding the coarsest partition π' of S, consistent with the initial partition $\{F, S - F\}$, such that if states s and t are in one block of π', then states $\delta(s, a)$ and $\delta(t, a)$ are also in one block of M' for each input symbol a.

The only difference between this problem and that of Algorithm 4.5 is that δ is a mapping from $S \times I$ to S rather than just a mapping from S to S. However, we can treat δ as a set $\{\delta_{a_1}, \delta_{a_2}, \ldots, \delta_{a_m}\}$ of functions on S, where each δ_a is the restriction of δ to the input symbol a.

Algorithm 4.5 can be easily modified to handle this more general problem by placing pairs (i, δ_a) in the set WAITING. Each pair (i, δ_a) consists of the index i of a block of the partition plus δ_a, the function on which to partition. Initially, WAITING $= \{(i, \delta_a) | i = 1 \text{ or } 2 \text{ and } a \in I\}$, since the initial partition $\{F, S - F\}$ has two blocks. Whenever a block $B[j]$ is split into $B[j]$ and $B[q]$, each possible function δ_a is paired with j and q. The remaining details are left for an exercise.

4.14 CHAPTER SUMMARY

Figure 4.36 summarizes the various data structures discussed in this chapter, the types of instructions they can handle, and the assumption made about the size and nature of the universal set from which elements are drawn.

			Time to process n instructions on sets of size n	
Data structure	Type of universe	Instructions permitted	Expected time	Worst-case time
1. Hash table	Arbitrary set on which a hashing function can be computed	MEMBER, INSERT, DELETE	$O(n)$	$O(n^2)$
2. Binary search tree	Arbitrary ordered set	MEMBER, INSERT, DELETE, MIN	$O(n \log n)$	$O(n^2)$
3. Tree structure of Algorithm 4.3	Integers 1 to n	MEMBER, INSERT, DELETE, UNION, FIND	$O(nG(n))$ at most	$O(nG(n))$ at most
4. 2–3 trees with leaves unordered	Arbitrary ordered set	MEMBER, INSERT, DELETE, UNION, FIND, MIN	$O(n \log n)$	$O(n \log n)$
5. 2–3 trees with leaves ordered	Arbitrary ordered set	MEMBER, INSERT, DELETE, FIND, SPLIT, MIN, CONCATENATE	$O(n \log n)$	$O(n \log n)$

Fig. 4.36. Summary of properties of data structures.

EXERCISES

4.1 Give a subset of the seven primitive operations of Section 4.1 that is sufficient to sort an arbitrary sequence of n elements. What can be said about the complexity of executing a stream of n instructions chosen from your subset?

4.2 Suppose that elements are strings of letters and the following hashing function for a table of size $m = 5$ is used: Add the "values" of the letters, where A has value 1, B has value 2, and so on. Divide the resulting sum by 5 and take the remainder. Show the contents of the hash table and lists, given that the following strings are inserted: DASHER, DANCER, PRANCER, VIXEN, COMET, CUPID, DONNER, BLITZEN.

4.3 Insert the eight strings of Exercise 4.2 into a binary search tree. What is the sequence of vertices visited if we wish to test RUDOLPH for membership in the set?

4.4 Show that the procedure SEARCH of Fig. 4.3 (p. 114) gives the smallest possible expected search time if all elements in the universal set are equally likely to be sought.

4.5 Find an optimal binary search tree for $a, b, \ldots, h$ if the elements, in order, have probabilities 0.1, 0.2, 0.05, 0.1, 0.3, 0.05, 0.15, 0.05 and all other elements have zero probability.

**4.6 Show that in Algorithm 4.2 we may restrict our search for m in line 8 of Fig. 4.9 (p. 121) to the range of the positions of $r_{i,j-1}$ through $r_{i+1,j}$ and still be guaranteed to find a maximum.

*4.7 Use Exercise 4.6 to modify Algorithm 4.2 so that it runs in $O(n^2)$ time.

4.8 Complete the proof of Theorem 4.2 by showing that the tree construction algorithm of Fig. 4.10 (p. 122) works correctly.

4.9 Complete the proof of Theorem 4.3.

*4.10 Construct an interface to translate between a set of n external names in the range 1 to r and a set of internal names consisting of the integers 1 to n. The interface must be such that one can translate in either direction, assuming $r >> n$.
a) Design the interface for good expected time behavior.
b) Design the interface for good worst-case behavior.

*4.11 Find an efficient data structure for representing a subset S of the integers from 1 to n. Operations we wish to perform on the set are:
1. Select an unspecified integer from the set and delete it.
2. Add an integer i to the set.
A mechanism must be provided to ignore a request to add integer i to S in the case where S already contains i. The data structure must be such that the time to select and delete an element and the time to add an element are constant independent of $\|S\|$.

4.12 Show the tree that results when Algorithm 4.3 is used to execute the sequence of UNION and FIND instructions generated by the following program. Assume set i is $\{i\}$ initially, for $1 \le i \le 16$.

```
begin
    for $i \leftarrow 1$ step 2 until 15 do UNION($i$, $i + 1$, $i$);
    for $i \leftarrow 1$ step 4 until 13 do UNION($i$, $i + 2$, $i$);
    UNION(1, 5, 1);
    UNION(9, 13, 9);
    UNION(1, 9, 1);
    for $i \leftarrow 1$ step 4 until 13 do FIND($i$)
end
```

4.13 Let σ be a sequence of UNION and FIND instructions in which all UNION's occur before the FIND's. Prove that Algorithm 4.3 executes σ in time proportional to the length of σ.

4.14 An S_0-tree is a tree consisting of a single vertex. For $i > 0$, an S_i-tree is obtained by making the root of one S_{i-1}-tree the son of the root of another S_{i-1}-tree. Prove the following.

a) An S_n-tree has $\binom{n}{h}$ vertices of height h.

b) An S_n-tree can be obtained from an S_m-tree, $m \le n$, by replacing each vertex of the S_m-tree by an S_{n-m}-tree. Sons of the vertex become sons of the root of the replacing S_{n-m} tree.

c) An S_n-tree contains a vertex with n sons; the sons are roots of an S_0-, an S_1-, . . . , and an S_{n-1}-tree.

4.15 Consider an algorithm for the disjoint-set union problem which makes the root of the tree with fewer vertices (ties are broken arbitrarily) a son of the root of the larger but does not use path compression. Prove that the upper bound of $O(n \log n)$ cannot be improved. That is, show that for some constant c and for arbitrarily large n, there exist sequences of UNION and FIND instructions which require $cn \log n$ steps.

**4.16 Let $T(n)$ be the worst-case time complexity of executing n UNION and FIND instructions, using the tree structure of Section 4.7 with path compression for the FIND's but executing UNION(A, B, C) by making the root of A a son of the root of B independent of which set is the larger. Show that $T(n) \geq k_1 n \log n$ for some constant $k_1 > 0$.

**4.17 Show that $T(n) \leq k_2 n \log n$ for some constant k_2, where $T(n)$ is as in Exercise 4.16.

*4.18 Show how we may execute a sequence of n instructions UNION, FIND, MEMBER, INSERT, DELETE on integers 1, 2, . . . , n in time $O(nG(n))$. Assume that DELETE(i, S) makes i the member of a new set $\{i\}$, which should be given an (arbitrary) name. This set may be subsequently merged with another. Also assume no element is a member of more than one set.

4.19 Generalize the off-line MIN problem to handle a MIN instruction of the form MIN(i) which determines all integers less than i to the left of the MIN instruction not found by a previous MIN instruction.

4.20 Develop complete data structures for the off-line MIN problem including the representation of trees by arrays, and write a program using arrays rather than the higher-level commands of the disjoint-set union algorithm.

4.21 Generalize the off-line MIN problem as follows. Let T be a tree with n vertices. An integer between 1 and n is associated with each vertex. Associated with certain vertices are EXTRACT_MIN instructions. Traverse the tree in postorder. On encountering an EXTRACT_MIN instruction at vertex v, locate and delete the smallest integer on the subtree with root v (excluding the integer at v) not previously deleted. Give an off-line $O(nG(n))$ algorithm for this process.

4.22 Design an $O(n \operatorname{loglog} n)$ solution to the UNION_FIND problem using a data structure consisting of a tree each of whose leaves is of distance 2 from the root. Restrict the degree of the root to be between 1 and $n/\log n$. Restrict the degree of each son of the root to be between 1 and $\log n$. How can the algorithm be modified to get an $O(n \operatorname{logloglog} n)$ time bound? What is the best time bound that you can obtain by a generalization of this method?

4.23 Show how to keep track of the displacements from the origin in a symbol table as mentioned in Application 2 of Section 4.8. [*Hint:* Utilize the technique used for depth determination.]

4.24 Write a program for the UNION and FIND primitives with weight computation as in Application 2 of Section 4.8.

4.25 Test the finite automata illustrated below for equivalence, using the algorithm of Fig. 4.25 (p. 144). The start states are 1 and *A*, respectively, and the sets of final states are $\{5\}$ and $\{C, E\}$, respectively.

	Input	
Present state	0	1
1	2	3
2	3	5
3	4	5
4	2	5
5	1	2

Next state

	Input	
Present state	0	1
A	D	B
B	D	C
C	A	B
D	B	E
E	A	D

Next state

4.26 Write complete programs to execute the following instructions on 2–3 trees.
a) DELETE
b) UNION
c) MEMBER (assuming leaves are ordered and each vertex has the label of its highest leaf attached)
d) SPLIT (assuming the leaf at which the split is to occur is given and leaves are ordered)

4.27 Write a complete program to insert a new vertex into a 2–3 tree assuming the leaves are ordered.

4.28 Write programs for the primitives MEMBER, INSERT, DELETE, MIN, UNION, and FIND using the 2–3 tree with the SMALLEST labels of Section 4.11. Assume the universal set is $\{1, 2, \ldots, n\}$.

4.29 Consider the 2–3 tree implementation of the mergeable heap (INSERT, DELETE, UNION, MIN). Assume that the universe from which elements are drawn is large. Describe how to implement FIND in $O(\log n)$ steps per FIND instruction, where n is the total number of elements in all heaps combined.

Definition. An *AVL tree*† is a binary search tree such that at each vertex v the heights of the left and right subtrees of v differ by at most one. If a subtree is missing it is deemed to be of "height" -1.

† Named for its originators, Adel'son-Vel'skii and Landis [1962].

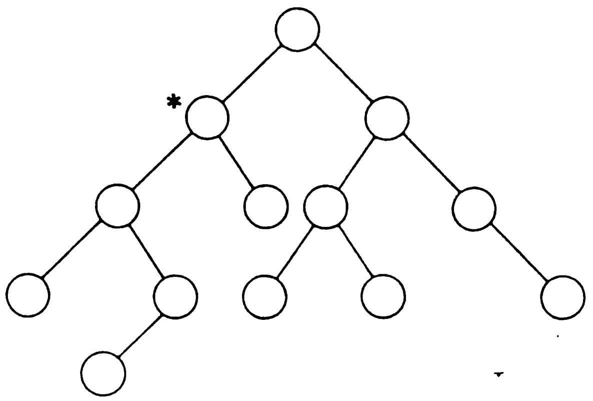


Fig. 4.37. A non-AVL tree.

Example 4.14. The tree of Fig. 4.37 is not an AVL tree because the vertex marked $*$ has a left subtree of height 2 and a right subtree of height 0. The AVL condition is, however, met at every other vertex in Fig. 4.37. □

4.30 Show that an AVL tree of height h has at most $2^{h+1} - 1$ vertices and at least

$$\frac{5+2\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^h + \frac{5-2\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^h - 1$$

vertices.

*4.31 Let T be a binary search tree with n vertices that is an AVL tree. Write $O(\log n)$ algorithms for the instructions INSERT and DELETE that keep the tree an AVL tree. You may assume the height of each vertex can be found at that vertex and that height information is automatically updated.

*4.32 Write algorithms to split an AVL tree and to concatenate two AVL trees. The algorithms should work in time proportional to the heights of the trees.

*4.33 Use an AVL tree as the basis of an algorithm to execute MIN, UNION, and DELETE on sets consisting of integers 1 through n, using $O(\log n)$ steps per operation.

Definition. The *balance* of a vertex v in a binary tree is $(1 + L)/(2 + L + R)$, where L and R are the numbers of vertices in the left and right subtrees of v. A binary search tree is *α-balanced* if every vertex has balance between α and $1 - \alpha$.

Example 4.15. The balance of the vertex marked $*$ in Fig. 4.37 is $\frac{5}{7}$. No other vertex has a balance which deviates that greatly from $\frac{1}{2}$, so the tree of Fig. 4.37 is $\frac{2}{7}$-balanced. □

*4.34 Give upper and lower bounds on the number of vertices in an α-balanced tree of height h.

*4.35 Repeat Exercises 4.31–4.33 for trees of balance α, where $\alpha \leq \frac{1}{3}$.

*4.36 Can you do Exercises 4.31–4.33 if $\alpha > \frac{1}{3}$?

4.37 Design a balanced tree with data at leaves where balancing is obtained b keeping the difference in the height of subtrees to within a fixed constant.

*4.38 Write an $O(nG(n))$ algorithm which, given an n-vertex tree and a list of n pai of vertices, determines for each pair (v, w) the vertex which is an ancestor c both v and w and is the closest of all such common ancestors of v and v

*4.39 The *external path* length of a binary tree is the sum over all leaves of their deptl The *internal path* length of a binary tree is the sum of the depths of all vertice What is the relation between external and internal path length if every vertex hε two sons or none?

*4.40 Design a data structure for implementing queues so that the following operε tions can be executed on-line.

a) ENQUEUE(i, A)—add integer i to queue A. Distinct instances of the sam integers are permitted.

b) DEQUEUE(A)—extract from queue A that element which has been ther longest.

c) MERGE(A, B, C)—merge queues A and B and call the resulting queue C Elements are deemed present on the merged queue for as long as they wer on whichever of A or B they were on. Note that the same integer may occu several times on A and B, and each occurrence is to be considered a separat element.

What is the time needed for execution of a sequence of n instructions?

*4.41 Design a data structure for implementing the operations in Exercise 4.40 off line. What is the time needed to execute a sequence of n of these instruction off-line?

*4.42 In Algorithm 4.5 suppose the initial partition $\pi = \{B_1, \ldots, B_q\}$ has the prop erty that for each $1 \leq i \leq q, f^{-1}(B_i) \subseteq B_j$ for some j. Show that at most one j i selected on line 6 of Fig. 4.35. Is the time complexity of Algorithm 4.5 reduce by this simplification?

Research Problem

4.43 There are a number of unresolved questions concerning how fast one can ex ecute sequences of n operations chosen from the seven given in Section 4.1—o other primitives for that matter. If the elements are chosen from an arbitrar set, and the only way to obtain information about them is by comparisons, the any set of primitives with which one can sort must take $O_C(n \log n)$ time. How ever, there is little that can be used to argue that anything more than $O(n)$ tim is required when the universe is the set of integers $\{1, 2, \ldots, n\}$ (or even through n^k for fixed k), or if the set of primitives is not sufficient to sort Therefore there is much room for improvement on either the expected or worst case times shown in Fig. 4.36. Alternatively, can one show lower bounds bette than $O(n)$ for some subset (or even all) of the primitives on the integers through n?

BIBLIOGRAPHIC NOTES

Three collections of information about a variety of hash table techniques are Morris [1968], Aho and Ullman [1973], and Knuth [1973a]. The latter also includes information on binary search trees. Algorithm 4.2 for constructing static binary search trees is from Gilbert and Moore [1959]. An $O(n^2)$ algorithm for the same problem was found by Knuth [1971] and the solution to Exercise 4.6 may be found there. Hu and Tucker [1971] showed that $O(n^2)$ time and $O(n)$ space are sufficient to construct an optimal binary search tree in which the data appear only at the leaves. Knuth [1973a] showed how to implement this algorithm in $O(n \log n)$ time.

Reingold [1972] gives optimal algorithms for many basic set manipulations, such as union and intersection. Algorithm 4.3 for the UNION-FIND problem was apparently first used by M. D. McIlroy and R. Morris. Knuth [1969] attributes path compression to A. Tritter. Theorem 4.4 is from Hopcroft and Ullman [1974] and Theorem 4.5 is by Tarjan [1974]. The application to equivalencing of identifiers and the displacement computation discussed in Section 4.8 is from Galler and Fischer [1964]. Application 3 on the equivalence of finite automata is from Hopcroft and Karp [1971]. Exercises 4.16 and 4.17 concerning the complexity of Algorithm 4.3 without path compression, are from Fischer [1972] and Paterson [1973], respectively.

AVL trees are from Adel'son-Vel'skii and Landis [1962], and answers to Exercises 4.31 and 4.32 can be found in Crane [1972]. The notion of bounded balance trees is from Nievergelt and Reingold [1973]. The reader may consult this reference for Exercises 4.34–4.37. An extension to the use of 2–3 trees can be found in Ullman [1974].

Exercise 4.22 represents an early solution to the UNION-FIND problem and can be found in Stearns and Rosenkrantz [1969]. Exercise 4.38 is from Aho, Hopcroft, and Ullman [1974].

The distinction between on-line and off-line algorithms is due to Hartmanis, Lewis, and Stearns [1965]. Rabin [1963] studied an important restricted form of on-line computation called real-time.

The partitioning algorithm is taken from Hopcroft [1971], as is the application to state minimization in finite automata.

ALGORITHMS ON GRAPHS

CHAPTER 5

Many problems in engineering and science can be formulated in terms of undirected or directed graphs. In this chapter we shall discuss some of the principal graph problems that have solutions that are polynomial (in running time) in the number of vertices, and hence edges, of a graph. We shall concentrate on problems dealing with the connectivity of graphs. Included are algorithms for finding spanning trees, biconnected components, strongly connected components, and paths between vertices. In Chapter 10 we shall study more difficult graph problems.

5.1 MINIMUM-COST SPANNING TREES

Let $G = (V, E)$ be a connected, undirected graph with a cost function mapping edges to real numbers. A spanning tree, we recall, is an undirected tree that connects all vertices in V. The cost of a spanning tree is just the sum of the costs of its edges. Our goal is to find a spanning tree of minimal cost for G. We shall see that a minimum-cost spanning tree for a graph with e edges can be found in $O(e \log e)$ time in general and in $O(e)$ time if e is sufficiently large compared with the number of vertices (see Exercise 5.3). Many spanning tree algorithms are based on the following two lemmas.

Lemma 5.1. Let $G = (V, E)$ be a connected, undirected graph and $S = (V, T)$ a spanning tree for G. Then

a) for all v_1 and v_2 in V, the path between v_1 and v_2 in S is unique, and
b) if any edge in $E - T$ is added to S, a unique cycle results.

Proof. Part (a) is trivial, since if there were more than one path there would be a cycle.

Part (b) is likewise trivial, since there must already be a path between the endpoints of the added edge. □

Lemma 5.2. Let $G = (V, E)$ be a connected, undirected graph and c a cost function on its edges. Let $\{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\}$ be any

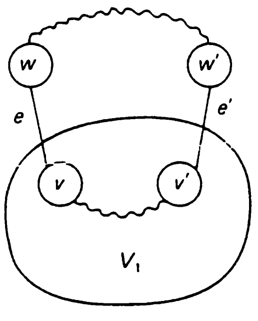


Fig. 5.1 A cycle in graph G.

spanning forest for G with $k > 1$. Let $T = \bigcup_{i=1}^{k} T_i$. Suppose $e = (v, w)$ is an edge of lowest cost in $E - T$ such that $v \in V_1$ and $w \notin V_1$. Then there is a spanning tree for G which includes $T \cup \{e\}$ and is of as low a cost as any spanning tree for G that includes T.

Proof. Suppose to the contrary that $S' = (V, T')$ is a spanning tree for G such that T' includes T but not e, and that S' is of lower cost than any spanning tree for G that includes $T \cup \{e\}$.

By Lemma 5.1(b), the addition of e to S' forms a cycle, as shown in Fig. 5.1. The cycle must contain an edge $e' = (v', w')$, other than e, such that $v' \in V_1$ and $w' \in V_1$. By hypothesis $c(e) \leq c(e')$.

Consider the graph S formed by adding e to S' and deleting e' from S'. S has no cycle, since the only cycle was broken by deletion of edge e'. Moreover, all vertices in V are still connected, since there is a path between v' and w' in S. Thus S is a spanning tree for G. Since $c(e) \leq c(e')$, S is no more costly than S'. But S contains both T and e, contradicting the minimality of S'. □

We now give one algorithm to find a minimum-cost spanning tree for an undirected graph $G = (V, E)$. The algorithm is essentially the same as the one in Example 4.1. The algorithm maintains a collection VS of disjoint sets of vertices. Each set W in VS represents a connected set of vertices forming a spanning tree in the spanning forest represented by VS. Edges are chosen from E, in order of increasing cost. We consider each edge (v, w) in turn. If v

```
begin
    $T \leftarrow \emptyset$;
    $VS \leftarrow \emptyset$;
    construct a priority queue $Q$ containing all edges in $E$;
    for each vertex $v \in V$ do add $\{v\}$ to $VS$;
    while $\|VS\| > 1$ do
        begin
            choose $(v, w)$, an edge in $Q$ of lowest cost;
            delete $(v, w)$ from $Q$;
            if $v$ and $w$ are in different sets $W_1$ and $W_2$ in $VS$ then
                begin
                    replace $W_1$ and $W_2$ in $VS$ by $W_1 \cup W_2$;
                    add $(v, w)$ to $T$
                end
        end
end
```

Fig. 5.2. Minimum-cost spanning tree algorithm.

and w are already in the same set in VS, we discard the edge. If v and w are in distinct sets W_1 and W_2 (which means W_1 and W_2 are not yet connected), we merge W_1 and W_2 into a single set and add (v, w) to T, the set of edges in the final spanning tree. The disjoint-set union algorithm of Section 4.7 can be used here. By Lemma 5.2 and an easy induction on the number of edges selected, we know that at least one minimum-cost spanning tree for G will contain this edge.

Algorithm 5.1. Minimum-cost spanning tree (Kruskal's algorithm).

Input. An undirected graph $G = (V, E)$ with a cost function c on the edges.

Output. $S = (V, T)$, a minimum-cost spanning tree for G.

Method. The program is given in Fig. 5.2 on p. 173. □

Example 5.1. Consider the undirected graph in Fig. 5.3. The list of edges, sorted in order of increasing cost, is:

Edge	Cost	Edge	Cost
(v_1, v_7)	1	(v_4, v_5)	17
(v_3, v_4)	3	(v_1, v_2)	20
(v_2, v_7)	4	(v_1, v_6)	23
(v_3, v_7)	9	(v_5, v_7)	25
(v_2, v_3)	15	(v_5, v_6)	28
(v_4, v_7)	16	(v_6, v_7)	36

Of course the edges are not actually sorted in step 3 of Algorithm 5.1 but are kept in a heap, 2–3 tree, or some other suitable data structure until they are needed. In fact, the heap of Section 3.4 is an ideal choice to implement the priority queue. The repeated request to find the minimum-cost edge at line 6

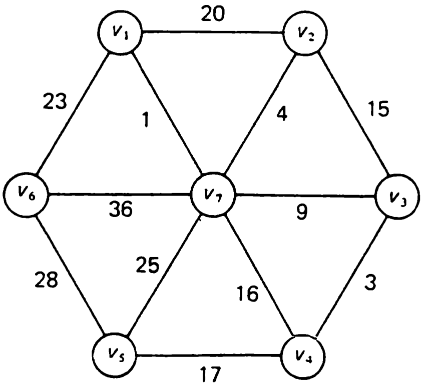


Fig. 5.3 An undirected graph with costs on edges.

Edge	Action	Sets in VS (connected components)
(v_1, v_7)	Add	$\{v_1, v_7\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_4)	Add	$\{v_1, v_7\}, \{v_2\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_2, v_7)	Add	$\{v_1, v_2, v_7\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}$
(v_3, v_7)	Add	$\{v_1, v_2, v_3, v_4, v_7\}, \{v_5\}, \{v_6\}$
(v_2, v_3)	Reject	
(v_4, v_7)	Reject	
(v_4, v_5)	Add	$\{v_1, v_2, v_3, v_4, v_5, v_7\}, \{v_6\}$
(v_1, v_2)	Reject	
(v_1, v_6)	Add	$\{v_1, \ldots, v_7\}$

Fig. 5.4. Sequence of steps for constructing a spanning tree.

s the basic operation of Heapsort. Moreover, the number of edges chosen at line 6 to construct a spanning tree is often less than $\|E\|$. In such situations, we save time since we never sort E completely.

Initially, each vertex is in a set by itself in VS. The lowest-cost edge is (v_1, v_7), so we add this edge to the tree and merge the sets $\{v_1\}$ and $\{v_7\}$ in VS. We then consider (v_3, v_4). Since v_3 and v_4 are in different sets in VS, we add (v_3, v_4) to the tree and merge $\{v_3\}$ and $\{v_4\}$. Next, we add (v_2, v_7) and merge $\{v_2\}$ with $\{v_1, v_7\}$. We also add the fourth edge (v_3, v_7) and merge $\{v_1, v_2, v_7\}$ with $\{v_3, v_4\}$.

Then, we consider (v_2, v_3). Both v_2 and v_3 are in the same set, $\{v_1, v_2, v_3, v_4, v_7\}$. Thus there is a path from v_2 to v_3 consisting of edges already in the spanning tree, so we do not add (v_2, v_3). The entire sequence of steps is summarized in Fig. 5.4. The resulting undirected spanning tree is shown in Fig. 5.5. □

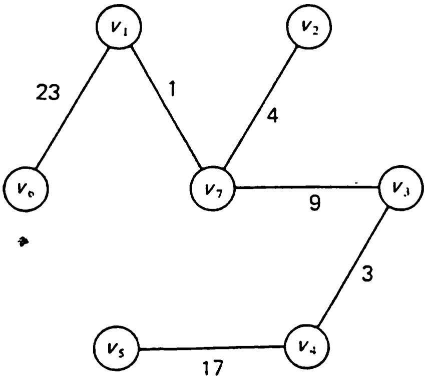


Fig. 5.5 A minimum-cost spanning tree.

Theorem 5.1. Algorithm 5.1 finds a minimum-cost spanning tree if the graph G is connected. If d edges are examined in the loop of lines 5–10, then the time spent is $O(d \log e)$, where $e = \|E\|$. Hence Algorithm 5.1 requires at most $O(e \log e)$ time.

Proof. Correctness of the algorithm follows immediately from Lemma 5.2 and from the fact that lines 8 and 9 correctly keep vertices in the same set if and only if they are in the same tree in the spanning forest represented by VS.

For the timing, suppose that d iterations of the loop of lines 5–10 are required. Discovery of a lowest-cost edge on Q (line 6) requires $O(\log e)$ steps if Q is implemented by a priority queue. The total time to find all the sets W_1 and W_2 containing v and w (line 8) and to replace them by their union (line 9) is at most $O(e\ G(e))$ if the fast disjoint-set union algorithm is used. The remainder of the loop clearly requires a constant amount of time independent of the size of G. Initialization of Q takes $O(e)$ time and initialization of VS, $O(n)$ time, where n is the number of vertices in V. □

5.2 DEPTH-FIRST SEARCH

Consider visiting the vertices of an undirected graph in the following manner. We select and "visit" a starting vertex v. Then we select any edge (v, w) incident upon v, and visit w. In general, suppose x is the most recently visited vertex. The search is continued by selecting some unexplored edge (x, y) incident upon x. If y has been previously visited, we find another new edge incident upon x. If y has not been previously visited, then we visit y and begin the search anew starting at vertex y. After completing the search through all paths beginning at y, the search returns to x, the vertex from which y was first reached. The process of selecting unexplored edges incident upon x is continued until the list of these edges is exhausted. This method of visiting the vertices of an undirected graph is called a *depth-first search* since we continue searching in the forward (deeper) direction as long as possible.

Depth-first search can be applied to a directed graph as well. If the graph is directed, then at vertex x we select only edges (x, y) directed out of x. After exhausting all edges out of y, we return to x even though there may be other edges directed into y which have not yet been searched.

If depth-first search is applied to an undirected graph which is connected, then it is easy to show that every vertex will be visited and every edge examined. If the graph is not connected, then a connected component of the graph will be searched. Upon completion of a connected component, a vertex not yet visited is selected as the new start vertex and a new search is begun.

A depth-first search of an undirected graph $G = (V, E)$ partitions the edges in E into two sets T and B. An edge (v, w) is placed in set T if vertex w has not been previously visited when we are at vertex v considering edge (v, w). Otherwise, edge (v, w) is placed in set B. The edges in T are called

```
procedure SEARCH($v$):
begin
    mark $v$ "old";
    for each vertex $w$ on $L[v]$ do
        if $w$ is marked "new" then
            begin
                add $(v, w)$ to $T$;
                SEARCH($w$)
            end
end
```

Fig. 5.6. Depth-first search.

tree edges, and those in B *back edges*. The subgraph (V, T) is an undirected forest, called a *depth-first spanning forest for G*. In the case that the forest consists of a single tree, (V, T) is called a *depth-first spanning tree*. Note that if G is connected, the depth-first spanning forest will be a tree. We consider each tree in the depth-first spanning forest to be rooted at that vertex at which the depth-first search of that tree was begun.

An algorithm for the depth-first search of a graph is given below.

Algorithm 5.2. Depth-first search of an undirected graph.

Input. A graph $G = (V, E)$ represented by adjacency lists $L[v]$, for $v \in V$.

Output. A partition of E into T, a set of tree edges, and B, a set of back edges.

Method. The recursive procedure SEARCH(v) in Fig. 5.6 adds edge (v, w) to T if vertex w is first reached during the search by an edge from v. We assume all vertices are initially marked "new." The entire algorithm is as follows:

```
begin
    $T \leftarrow \emptyset$:
    for all $v$ in $V$ do mark $v$ "new";
    while there exists a vertex $v$ in $V$ marked "new" do
        SEARCH($v$)
end
```

All edges in E not placed in T are considered to be in B. Note that if edge (v, w) is in E, then w will be on $L[v]$ and v will be on $L[w]$. Thus we cannot simply place edge (v, w) in B if we are at vertex v and vertex w is marked "old" since w might be the father of v. □

Example 5.2. We shall adopt the convention of showing the tree edges in T solid and back edges in B dashed. Also, the tree (or trees) will be drawn with

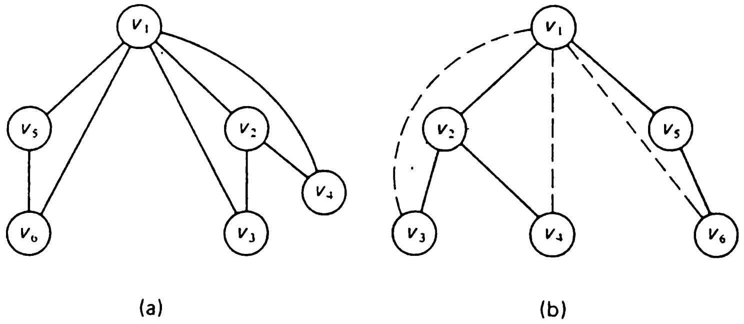


Fig. 5.7 A graph and its depth-first spanning tree.

the root (the starting vertex selected at line 8) at the top, and with the sons ol each vertex drawn from left to right in the order in which their edges were added at line 4 of SEARCH. Consider the graph of Fig. 5.7(a). One possible partition of this graph into tree and back edges resulting from a depth-first search is shown in Fig. 5.7(b).

Initially, all vertices are "new." Suppose v_1 is selected at line 8. When we perform SEARCH(v_1), we might select $w = v_2$ at line 2. Since v_2 is marked "new," we add (v_1, v_2) to T and call SEARCH(v_2). SEARCH(v_2) might select v_1 from $L[v_2]$, but v_1 has been marked "old." Then suppose we select $w = v_3$. Since v_3 is "new," we add (v_2, v_3) to T and call SEARCH(v_3). Each of the vertices adjacent to v_3 is now "old," so we return to SEARCH(v_2).

Then, proceeding with SEARCH(v_2), we find edge (v_2, v_4), add it to T, and call SEARCH(v_4). Note that v_4 is drawn to the right of v_3, a previously found son of v_2 in Fig. 5.7(b). No "new" vertices are adjacent to v_4, so we return to SEARCH(v_2). Now we find no "new" vertices adjacent to v_2, and so return to SEARCH(v_1). Continuing, SEARCH(v_1) finds v_5, and SEARCH(v_5) finds v_6. All vertices would then be on the tree and marked "old," and thus the algorithm would end. If the graph were not connected, the loop of lines 8–9 would repeat, once for each component. □

Theorem 5.2. Algorithm 5.2 requires $O(\text{MAX}(n, e))$ steps on a graph with n vertices and e edges.

Proof. Line 7 and the search for "new" vertices at line 8 require $O(n)$ steps if a list of vertices is made and scanned once. The time spent in SEARCH(v), exclusive of recursive calls to itself, is proportional to the number of vertices adjacent to v. SEARCH(v) is called only once for a given v, since v is marked "old" the first time SEARCH(v) is called. Thus the total time spent in SEARCH is $O(\text{MAX}(n, e))$, and we have the theorem. □

Part of the power of depth-first search is contained in the following lemma, which says that each edge of an undirected graph G is either an edge in the depth-first spanning forest or connects an ancestor to a descendant in some tree of the depth-first spanning forest. Thus all edges in G, whether tree or back, connect two vertices such that one vertex is an ancestor of the other in the spanning forest.

Lemma 5.3. If (v, w) is a back edge, then in the spanning forest v is an ancestor of w or vice versa.

Proof. Suppose without loss of generality that v is visited before w, in the sense that SEARCH(v) is called before SEARCH(w). Thus when v is reached, w is still labeled "new." All "new" vertices visited by SEARCH(v) will become descendants of v in the spanning forest. But SEARCH(v) cannot end until w is reached, since w is on the list $L[v]$. □

There is a natural order which depth-first search imposes on the vertices of a spanning forest. Namely, vertices can be labeled in the order they are visited if we initialize COUNT to 1 between lines 6 and 7 of Algorithm 5.2 and insert

```
DFNUMBER[$v$] ← COUNT;
COUNT ← COUNT + 1;
```

at the beginning of procedure SEARCH. Then the vertices of the forest would be labeled 1, 2, . . . , up to the number of vertices in the forest.

The labels can clearly be assigned in $O(n)$ time for a graph of n vertices. The order corresponds to a preorder traversal of each tree in the resulting spanning forest. We shall subsequently assume that all depth-first spanning forests are so labeled. We shall often treat these vertex labels as though they are the vertex names themselves. It thus makes sense to say, e.g., "$v < w$," where v and w are vertices.

Example 5.3. The depth-first order of the vertices of the graph in Fig. 5.7(a) is $v_1, v_2, v_3, v_4, v_5, v_6$, which can be ascertained either by tracing the order in which SEARCH was initiated for the various vertices or traversing the tree of Fig. 5.7(b) in preorder. □

Note especially that if v is a proper ancestor of w, then $v < w$. Also, if v is to the left of w in a tree, then $v < w$.

5.3 BICONNECTIVITY

We now consider an application of depth-first search to determining the biconnected components of an undirected graph. Let $G = (V, E)$ be a connected, undirected graph. A vertex a is said to be an *articulation point* of G if there exist vertices v and w such that v, w, and a are distinct, and every path between v and w contains the vertex a. Stated another way, a is an articulation point of G if removing a splits G into two or more parts. The graph G is *biconnected* if for every distinct triple of vertices v, w, a there

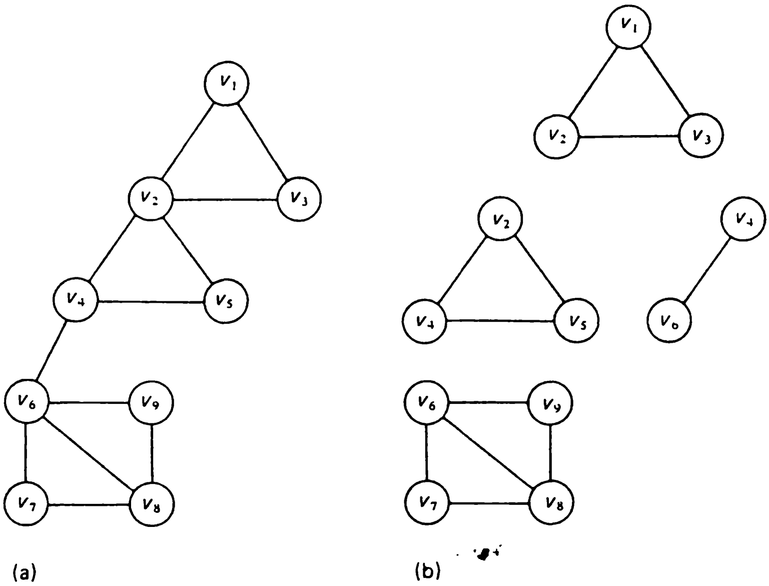


Fig. 5.8 (a) An undirected graph and (b) its biconnected components.

exists a path between v and w not containing a. Thus an undirected connected graph is biconnected if and only if it has no articulation points.

We can define a natural relation on the set of edges of G by saying that two edges e_1 and e_2 are related if $e_1 = e_2$ or there is a cycle containing both e_1 and e_2. It is easy to show that this relation is an equivalence relation† that partitions the edges of G into equivalence classes $E_1, E_2, \ldots, E_k$ such that two distinct edges are in the same class if and only if they lie on a common cycle. For $1 \leq i \leq k$, let V_i be the set of vertices of the edges in E_i. Each graph $G_i = (V_i, E_i)$ is called a *biconnected component* of G.

Example 5.4. Consider the undirected graph of Fig. 5.8(a). Vertex v_4, for example, is an articulation point, since every path between v_1 and v_7 passes through v_4. The equivalence classes of edges lying on common cycles are

$$\{(v_1, v_2), (v_1, v_3), (v_2, v_3)\},$$
$$\{(v_2, v_4), (v_2, v_5), (v_4, v_5)\},$$
$$\{(v_4, v_6)\},$$
$$\{(v_6, v_7), (v_6, v_8), (v_6, v_9), (v_7, v_8), (v_8, v_9)\}.$$

† R is an *equivalence relation* on set S if R is *reflexive* (aRa for all $a \in S$), *symmetric* (aRb implies bRa for all $a, b \in S$), and *transitive* (aRb and bRc implies aRc). It is easy to show that an equivalence relation on S partitions S into disjoint equivalence classes. (The subset $[a] = \{b|bRa\}$ is called an *equivalence class*.)

These sets give rise to the biconnected components shown in Fig. 5.8(b). The only unintuitive observation is that the edge (v_4, v_6), being in an equivalence class by itself (no cycles include this edge), gives rise to a "biconnected component" consisting of v_4 and v_6. □

The following lemma provides some useful information about biconnectivity.

Lemma 5.4. For $1 \le i \le k$, let $G_i = (V_i, E_i)$ be the biconnected components of a connected undirected graph $G = (V, E)$. Then

1. G_i is biconnected for each i, $1 \le i \le k$.
2. For all $i \neq j$, $V_i \cap V_j$ contains at most one vertex.
3. a is an articulation point of G if and only if $a \in V_i \cap V_j$ for some $i \neq j$.

Proof

1. Suppose there are three distinct vertices v, w, and a in V_i such that all paths in G_i between v and w pass through a. Then surely (v, w) is not an edge in E_i. Thus there are distinct edges (v, v') and (w, w') in E_i, and there is a cycle in G_i including these edges. By the definition of a biconnected component, all edges and vertices on this cycle are in E_i and V_i, respectively. Thus there are two paths in G_i between v and w, only one of which could contain a, a contradiction.

2. Suppose two distinct vertices v and w are in $V_i \cap V_j$. Then there exists a cycle C_1 in G_i that contains v and w, and a cycle C_2 in G_j that also contains v and w. Since E_i and E_j are disjoint, the sets of edges in C_1 and C_2 are disjoint. However, we may construct a cycle containing v and w that uses edges from both C_1 and C_2, implying that at least one edge in E_i is equivalent to an edge in E_j. Thus E_i and E_j are not equivalence classes, as supposed.

3. Suppose vertex a is an articulation point of G. Then there exist two vertices v and w such that v, w, and a are distinct, and every path between v and w contains a. Since G is connected, there is at least one such path. Let (x, a) and (y, a) be the two edges on a path between v and w incident upon a. If there is a cycle containing these two edges, then there is a path between v and w not containing a. Thus (x, a) and (y, a) are in different biconnected components, and a is in the intersection of their vertex sets.

For the converse, if $a \in V_i \cap V_j$, then there are edges (x, a) and (y, a) in E_i and E_j, respectively. Since both these edges do not occur on any one cycle, it follows that every path from x to y contains a. Thus a is an articulation point. □

Depth-first search is particularly useful in finding the biconnected components of an undirected graph. One reason for this is that by Lemma 5.3, there are no "cross edges." That is, if vertex v is neither an ancestor nor a descendant of vertex w in the spanning forest, then there can be no edge from v to w.

If vertex a is an articulation point, then the removal of a and all edges incident upon a splits the graph into two or more parts. One consists of a son s of a and all of its descendants in the depth-first spanning tree. Thus in the depth-first spanning tree a must have a son s such that there is no back edge between a descendant of s and a proper ancestor of a. Conversely, with the exception of the root of the spanning tree, the absence of cross edges implies that vertex a is an articulation point if there is no back edge from any descendant of some son of a to a proper ancestor of a. The root of the depth-first spanning tree is an articulation point if and only if it has two or more sons.

Example 5.5. A depth-first spanning tree for the graph of Fig. 5.8(a) is shown in Fig. 5.9. The articulation points are v_2, v_4, and v_6. Vertex v_2 has son v_4, and no descendant of v_4 has a back edge to a proper ancestor of v_2. Likewise, v_4 has son v_6, and v_6 has son v_8 with the analogous property. □

The preceding ideas are embodied in the following lemma.

Lemma 5.5. Let $G = (V, E)$ be a connected, undirected graph, and let $S = (V, T)$ be a depth-first spanning tree for G. Vertex a is an articula-

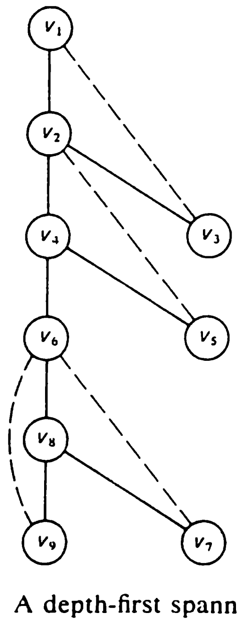


Fig. 5.9 A depth-first spanning tree.

tion point of G if and only if either

1. a is the root and a has more than one son, or
2. a is not the root, and for some son s of a there is no back edge between any descendant of s (including s itself) and a proper ancestor of a.

Proof. It is easy to show that the root is an articulation point if and only if it has more than one son. We leave this portion as an exercise.

Suppose condition 2 is true. Let f be the father of a. By Lemma 5.3 each back edge goes from a vertex to an ancestor of the vertex. Thus any back edge from a descendant v of s goes to an ancestor of v. By the hypothesis of the lemma the back edge cannot go to a proper ancestor of a. Hence it goes either to a or to a descendant of s. Thus every path from s to f contains a, implying that a is an articulation point.

To prove the converse, suppose that a is an articulation point but not the root. Let x and y be distinct vertices other than a such that every path in G between x and y contains a. At least one of x and y, say x, is a proper descendant of a in S, else there is a path in G between x and y using edges in T and avoiding a. Let s be the son of a such that x is a descendant of s (perhaps $x = s$). Either there is no back edge between a descendant v of s and a proper ancestor w of a, in which case condition 2 is immediately true, or there is such an edge (v, w). In the latter situation we must consider two cases.

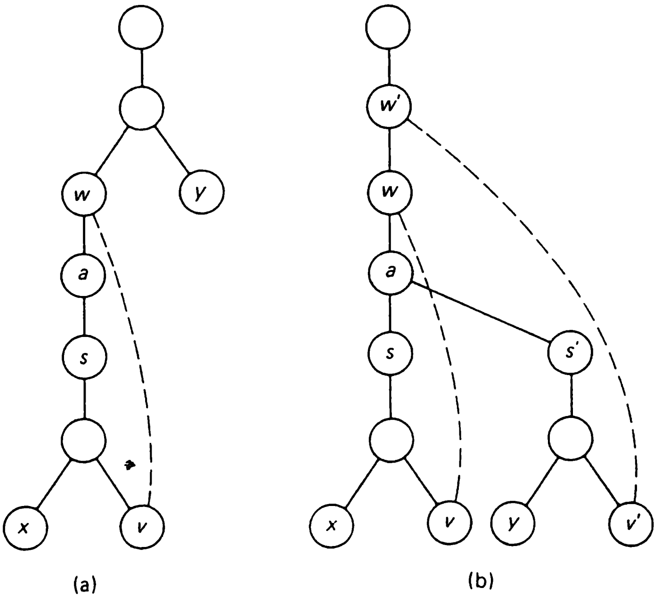


Fig. 5.10 Counterexample paths.

CASE 1. Suppose y is not a descendant of a. Then there is a path from x to v to w to y that avoids a, a contradiction. See Fig. 5.10(a).

CASE 2. Suppose y is a descendant of a. Surely y is not a descendant of s, else there is a path from x to y that avoids a. Let s' be the son of a such that y is a descendant of s'. Either there is no back edge between a descendant v' of s' and a proper ancestor w' of a, in which case condition 2 is immediately true, or there is such an edge (v', w'). In the latter case there is a path from x to v to w to w' to v' to y that avoids a, a contradiction. See Fig. 5.10(b). We conclude that condition 2 is true. □

Let T and B be the sets of tree and back edges produced by a depth-first search of a connected, undirected graph $G = (V, E)$. We assume the vertices in V are named by their depth-first numbers. For each v in V, we define

$$\text{LOW}[v] = \text{MIN}(\{v\} \cup \{w \mid \text{there exists a back edge } (x, w) \in B \text{ such that } x \text{ is a descendant of } v, \text{ and } w \text{ an ancestor of } v \text{ in the depth first spanning forest } (V, T)\}) \tag{5.1}$$

The preorder numbering implies that if x is a descendant of v and (x, w) is a back edge such that $w < v$, then w is a proper ancestor of v. Thus by Lemma 5.5, if vertex v is not the root, then v is an articulation point if and only if v has a son s such that $\text{LOW}[s] \geq v$.

We can embed into the procedure SEARCH a calculation to determine the LOW value of each vertex if we rewrite (5.1) to express LOW[v] in terms of the vertices adjacent to v via back edges and the values of LOW at the sons of v. Specifically, LOW[v] can be computed by determining the minimum value of those vertices w such that either

1. $w = v$, or
2. $w = \text{LOW}[s]$ and s is a son of v, or
3. (v, w) is a back edge in B.

The minimum value of w can be determined once $L[v]$, the list of vertices adjacent to v, is exhausted. Thus (5.1) is equivalent to

$$\text{LOW}[v] = \text{MIN}(\{v\} \cup \{\text{LOW}[s] \mid s \text{ is a son of } v\} \cup \{w \mid (v, w) \in B\}). \tag{5.2}$$

We have incorporated both the renaming of the vertices by first visit and the computation of LOW into the revised version of SEARCH shown in Fig. 5.11. In line 4 we initialize LOW[v] to its maximum possible value. If vertex v has a son w in the depth-first spanning forest, then in line 11 we adjust LOW[v] if LOW[w] is less than the current value of LOW[v]. If vertex v is connected by a back edge to vertex w, then in line 13 we make LOW[v] be DFNUMBER[w] if the depth-first number of vertex w is less than the current value of LOW[v]. The test on line 12 checks for the case

```
     procedure SEARCHB($v$):
     begin
1.       mark $v$ "old";
2.       DFNUMBER[$v$] $\leftarrow$ COUNT;
3.       COUNT $\leftarrow$ COUNT + 1;
4.       LOW[$v$] $\leftarrow$ DFNUMBER[$v$];
5.       for each vertex $w$ on $L[v]$ do
6.           if $w$ is marked "new" then
                 begin
7.                   add $(v, w)$ to $T$;
8.                   FATHER[$w$] $\leftarrow v$;
9.                   SEARCHB($w$);
10.                  if LOW[$w$] $\geq$ DFNUMBER[$v$] then a biconnected
                         component has been found;
11.                  LOW[$v$] $\leftarrow$ MIN(LOW[$v$], LOW[$w$])
                 end
12.          else if $w$ is not FATHER[$v$] then
13.              LOW[$v$] $\leftarrow$ MIN(LOW[$v$], DFNUMBER[$w$])
     end
```

Fig. 5.11. Depth-first search with LOW computation.

that (v, w) is not really a back edge because w is the father of v on the depth-first spanning tree. Thus Fig. 5.11 implements Eq. (5.2).

Having found LOW[v] for each vertex v, we can easily identify the articulation points. We first give the complete algorithm, then prove its correctness and show that it requires $O(e)$ time.

Algorithm 5.3. Finding biconnected components.

Input. A connected, undirected graph $G = (V, E)$.

Output. A list of the edges of each biconnected component of G.

Method

1. Initially set T to $\emptyset$ and COUNT to 1. Also, mark each vertex in V as being "new." Then select an arbitrary vertex v_0 in V and call SEARCHB(v_0) (Fig. 5.11) to build a depth-first spanning tree $S = (V, T)$ and to compute LOW(v) for each v in V.
2. When vertex w is encountered at line 5 of SEARCHB, put edge (v, w) on STACK, a pushdown store of edges, if it is not already there.[†] After dis-

[†] Note that if (v, w) is an edge, v is on $L[w]$ and w is on $L[v]$. Thus (v, w) is encountered twice, once when vertex v is visited and once when vertex w is visited. We can test whether (v, w) is already on STACK by checking if $v < w$ and w is "old" or if $v > w$ and w = FATHER[v].

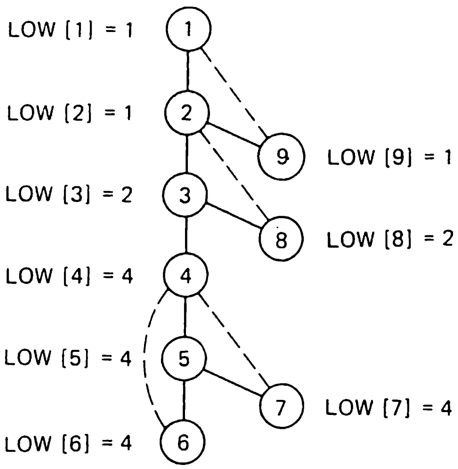


Fig. 5.12 Spanning tree of Fig. 5.9, with values of LOW.

covering a pair (v, w) at line 10 such that w is a son of v and $\text{LOW}[w] \geq v$, pop from STACK all edges up to and including (v, w). These edges form a biconnected component of G. □

Example 5.6. The depth-first spanning tree of Fig. 5.9 is reproduced as Fig. 5.12 with the vertices renamed by DFNUMBER and the values of LOW indicated. For example, SEARCHB(6) determines that $\text{LOW}[6] = 4$, since back edge (6, 4) exists. Then SEARCHB(5), which called SEARCHB(6), sets $\text{LOW}[5] = 4$, since 4 is less than the initial value of LOW[5], which is 5.

On completion of SEARCHB(5) we discover (line 10) that $\text{LOW}[5] = 4$. Thus 4 is an articulation point. At this point the pushdown store contains the edges (from bottom to top)

$$(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 4), (5, 7), (7, 4).$$

Thus we pop the edges down to and including (4, 5). That is, we output the edges (7, 4), (5, 7), (6, 4), (5, 6), and (4, 5) which are the edges of the first biconnected component found.

Observe that on completion of SEARCHB(2) we discover that $\text{LOW}[2] = 1$ and empty the pushdown store of edges even though 1 is not an articulation point. This insures that the biconnected component containing the root is emitted. □

Theorem 5.3. Algorithm 5.3 correctly finds the biconnected components of G and requires $O(e)$ time if G has e edges.

Proof. The proof that step 1 requires $O(e)$ time is a simple extension of that observation for SEARCH (Theorem 5.2). Step 2 examines each edge once, places it on a pushdown store, and subsequently pops it. Thus step 2 is $O(e)$.

For the correctness of the algorithm, Lemma 5.5 assures us that the articulation points are correctly identified. Even if the root is not an articulation point, it is treated as one in order to emit the biconnected component containing the root.

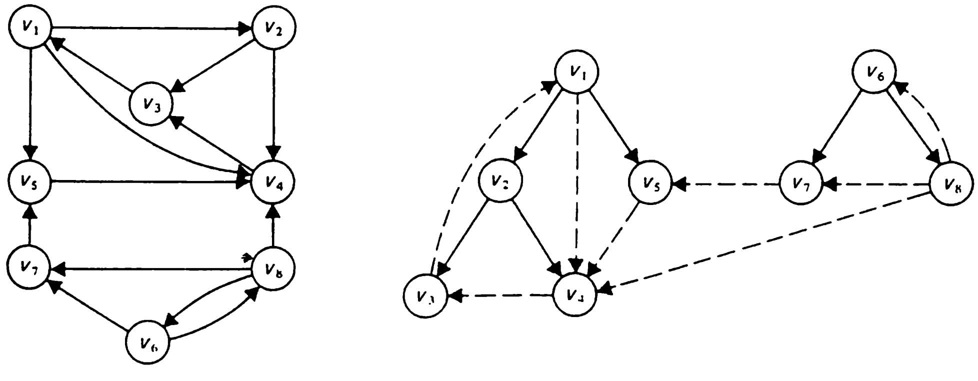
We must prove that if LOW$[w] \geq v$, then when SEARCHB(w) is completed the edges above (v, w) on STACK will be exactly those edges in the biconnected component containing (v, w). This is done by induction on the number b of biconnected components of G. The basis, $b = 1$, is trivial since in this case v is the root of the tree, (v, w) is the only tree edge out of v, and on completion of SEARCHB(w) all edges of G are on STACK.

Now, assume the induction hypothesis is true for all graphs with b biconnected components, and let G be a graph with $b + 1$ biconnected components. Let SEARCHB(w) be the first call of SEARCHB to end with LOW$(w) \geq v$, for (v, w) a tree edge. Since no edges have been removed from STACK, the set of edges above (v, w) on STACK is the set of all edges incident upon descendants of w. It is easily shown that these edges are exactly the edges of the biconnected component containing (v, w). On removal of these edges from STACK, the algorithm behaves exactly as it would on the graph G' that is obtained from G by deleting the biconnected component with edge (v, w). The induction step now follows since G' has b biconnected components. □

5.4 DEPTH-FIRST SEARCH OF A DIRECTED GRAPH

Algorithm 5.2 can also be used to find a directed spanning forest for a directed graph $G = (V, E)$ if we define the list $L[v]$ of vertices "adjacent" to v to be $\{w | (v, w) \text{ is an edge}\}$, that is, $L[v]$ is the list of vertices that are the heads of edges with tail v.

Example 5.7. A directed graph is shown in Fig. 5.13(a), and a depth-first



(a) (b)

Fig. 5.13 Depth-first search of a directed graph: (a) directed graph; (b) spanning forest.

spanning forest for it is shown in Fig. 5.13(b). As before, we show tree edges solid and others dashed.

To construct the spanning forest we begin at v_1. SEARCH(v_1) calls SEARCH(v_2), which calls SEARCH(v_3). The latter terminates with no additions to the tree, since the only edge with tail v_3 goes to v_1, which is already marked "old." Thus we return to SEARCH(v_2), which then adds v_4 as a second son of v_2. SEARCH(v_4) terminates, making no additions to the forest, since v_3 is already "old." Then SEARCH(v_2) ends, since all edges leaving v_2 have now been considered. Thus we fall back to v_1, which calls SEARCH(v_5). The latter terminates with no additions to the tree, and likewise, SEARCH(v_1) can add no more.

We now choose v_6 as the root of a new depth-first spanning tree. Its construction is similar to the preceding, and we leave it to the reader to follow it. Note that the order in which we have chosen to visit the vertices is $v_1, v_2, \ldots, v_8$. Thus the depth-first number of vertex v_i is i for $1 \leq i \leq 8$. □

Note that in the depth-first search of a directed graph there are three types of edges in addition to tree edges. There are back edges such as (v_3, v_1) in Fig. 5.13(b), right-to-left cross edges such as (v_4, v_3), and forward edges such as (v_1, v_4). However, no edge goes from a vertex with a lower depth-first number to one with a higher number unless the latter is a descendant of the former. This is not accidental.

The explanation is similar to the reason that there are no cross edges in the undirected case. Assume (v, w) is an edge and that v is visited before w (i.e., $v < w$). Every vertex assigned a number between the time SEARCH(v) began and the time it ended becomes a descendant of v. But w must be assigned a number at the time edge (v, w) is explored unless w was already assigned a number. If w is assigned a number at the time edge (v, w) is explored, (v, w) becomes a tree edge. Otherwise, (v, w) is a forward edge. Thus there can be no cross edge (v, w) with $v < w$.

The edges in a directed graph G are partitioned into four categories by a depth-first search of G:

1. *Tree edges*, which are edges leading to new vertices during the search.
2. *Forward edges*, which go from ancestors to proper descendants but are not tree edges.
3. *Back edges*, which go from descendants to ancestors (possibly from a vertex to itself).
4. *Cross edges*, which go between vertices that are neither ancestors nor descendants of one another.

The key property of cross edges is stated in the next lemma.

Lemma 5.6. If (v, w) is a cross edge, then $v > w$, i.e., cross edges go from right to left.

Proof. The proof is similar to that of Lemma 5.3 and is left as an exercise for the reader. □

5.5 STRONG CONNECTIVITY

As an example of an efficient algorithm made possible by depth-first search of a directed graph, consider the problem of determining whether a directed graph is strongly connected, i.e., whether there is a path from each vertex to every other vertex.

> **Definition.** Let $G = (V, E)$ be a directed graph. We can partition V into equivalence classes V_i, $1 \leq i \leq r$, such that vertices v and w are equivalent if and only if there is a path from v to w and a path from w to v. Let E_i, $1 \leq i \leq r$, be the set of edges connecting the pairs of vertices in V_i. The graphs $G_i = (V_i, E_i)$ are called the *strongly connected components* of G. Even though every vertex of G is in some V_i, G may have edges not in any E_i. A graph is said to be *strongly connected* if it has only one strongly connected component.

We now make use of the depth-first search to find the strongly connected components of a graph. We first show that the vertices of each strongly connected component are a connected subgraph of the spanning forest determined by the depth-first search. This connected subgraph is a tree and the root of the tree is called the *root of the strongly connected component.* However, not every tree in the depth-first spanning forest necessarily represents a strongly connected component.

> **Lemma 5.7.** Let $G_i = (V_i, E_i)$ be a strongly connected component of a directed graph G. Let $S = (V, T)$ be a depth-first spanning forest for G. Then the vertices of G_i together with the edges which are common to both E_i and T form a tree.

Proof. Let v and w be vertices in V_i. (We assume vertices are named by their depth-first numbers.) Without loss of generality assume that $v < w$. Since v and w are both in the same strongly connected component, there exists a path P in G_i from v to w. Let x be the lowest-numbered vertex on P (possibly v itself). Once path P reaches a descendant of x, it cannot leave the subtree of descendants of x, since the only edges out of the subtree are cross edges and back edges to vertices numbered lower than x. (Since the descendants of x are numbered consecutively starting with x, a cross edge or back edge out of the subtree of descendants of x must go to a vertex numbered less than x.) Thus w is a descendant of x. Because vertices are numbered in preorder, all vertices numbered between x and w are also descendants of x. Since $x \leq v < w$, v is a descendant of x.

We have just shown that any two vertices in G_i have a common ancestor in G_i. Let r be the lowest-numbered common ancestor of vertices in G_i. If v is in G_i, then any vertex on the spanning tree path from r to v is also in G_i. □

The strongly connected components of a directed graph G can be found by discovering the roots of the components in the order that the roots are last encountered during a depth-first search of G. Let $r_1, r_2, \ldots, r_k$ be the roots in the order in which the depth-first search of these vertices terminated (i.e., the search of r_i terminated before the search of r_{i+1}). Then for each $i < j$ either r_i is to the left of r_j or r_i is a descendant of r_j in the depth-first spanning forest.

Let G_i be the strongly connected component with root r_i for $1 \le i \le k$. Then G_1 consists of all descendants of r_1, since no $r_j, j > 1$, can be a descendant of r_1. Similarly, we can show the following.

Lemma 5.8. For each i, $1 \le i \le k$, G_i consists of those vertices which are descendants of r_i but are in none of $G_1, G_2, \ldots, G_{i-1}$.

Proof. The root r_j, for $j > i$, cannot be a descendant of r_i, since the call of SEARCH(r_j) terminates after SEARCH(r_i). □

To aid in finding the roots, a function called LOWLINK is defined:

$$\text{LOWLINK}[v] = \text{MIN}(\{v\} \cup \{w \mid \text{there is a cross edge or back edge from a descendant of } v \text{ to } w\text{, and the root of the strongly connected component containing } w \text{ is an ancestor of } v\}) \tag{5.3}$$

Figure 5.14 illustrates a cross edge from a descendant of v to w, where the root r of the strongly connected component containing w is an ancestor of v.

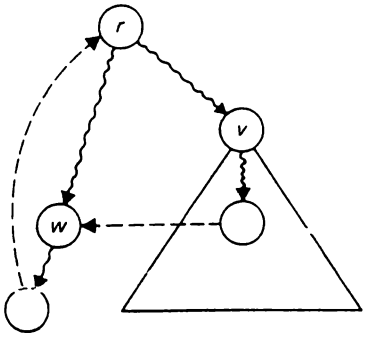


Fig. 5.14 A cross edge satisfying the condition of LOWLINK.

We shall soon see how to calculate LOWLINK as we perform a depth-first search. First, we provide a characterization of the roots of the strongly connected components in terms of LOWLINK.

Lemma 5.9. Let G be a directed graph. A vertex v is the root of a strongly connected component of G if and only if $\text{LOWLINK}[v] = v$.

Proof

ONLY IF. Assume v is the root of a strongly connected component of G. By definition of LOWLINK, $\text{LOWLINK}[v] \leq v$. Suppose that we have $\text{LOWLINK}[v] < v$. Then there are vertices w and r such that

1. w is reached by a cross or back edge from a descendant of v,
2. r is the root of the strongly connected component containing w,
3. r is an ancestor of v, and
4. $w < v$.

By condition 2, r is an ancestor of w, so $r \leq w$. Thus by condition 4, $r < v$, implying by condition 3 that r is a proper ancestor of v. But r and v must be in the same strongly connected component, since there is a path in G from r to v and a path from v to r via w. Thus v is not the root of a strongly connected component, which is a contradiction. Hence we may conclude that $\text{LOWLINK}[v] = v$.

IF: Assume $\text{LOWLINK}[v] = v$. If v is not the root of the strongly connected component containing v, then some proper ancestor r of v is the root. Thus there is a path P from v to r. Consider the first edge of P from a descendant of v to a vertex w which is not a descendant of v. The edge is either a back edge to an ancestor of v or a cross edge to a vertex numbered lower than v. In either case $w < v$.

It remains to be shown that r and w are in the same strongly connected component of G. There must be a path from r to v since r is an ancestor of v. The path P goes from v to w to r. Hence r and w are in the same strongly connected component. Thus $\text{LOWLINK}[v] \leq w < v$, a contradiction. □

The values of LOWLINK are easy to calculate during the depth-first search. The strongly connected components can also be easily found as follows. The vertices of G are placed on a stack in the order they are visited during the search. Whenever a root is found, all vertices of the corresponding strongly connected component are on top of the stack and are popped. This strategy "works" by Lemma 5.8 and the properties of depth-first numbering.

On reaching vertex v the first time, $\text{LOWLINK}[v]$ is set equal to v. If a back or cross edge (v, w) is explored, and w is in the same strongly connected component as some ancestor of v, then $\text{LOWLINK}[v]$ is set to the minimum of its current value and w. If a tree edge (v, w) is explored, then the subtree with root w is recursively explored and LOWLINK[w] calculated. After r turning to v, LOWLINK[v] is set to the minimum of its current value ar LOWLINK[w].

The test to determine whether w is in the same component as some ai cestor of v is accomplished by checking to see whether w is still on the stac of vertices. This test is implemented by using an array to indicate whether c not a vertex is on the stack. Note that by Lemma 5.8, if w is still on th stack, the root of the strongly connected component containing w is an ar cestor of v.

We now give a modification of the procedure SEARCH which compute LOWLINK. It makes use of a pushdown list STACK for vertices. Th procedure is shown in Fig. 5.15.

```
procedure SEARCHC(v):
begin
    mark v "old";
    DFNUMBER[v] ← COUNT;
    COUNT ← COUNT + 1;
    LOWLINK[v] ← DFNUMBER[v];
    push v on STACK;
    for each vertex w on L[v] do
        if w is marked "new" then
            begin
                SEARCHC(w);
                LOWLINK[v] ← MIN(LOWLINK[v],
                  LOWLINK[w])
            end
        else
            if DFNUMBER[w] < DFNUMBER[v] and w is on
              STACK then
                LOWLINK[v] ← MIN(DFNUMBER[w],
                  LOWLINK[v]);
    if LOWLINK[v] = DFNUMBER[v] then
        begin
            repeat
                begin
                    pop x from top of STACK;
                    print x
                end
            until x = v;
            print "end of strongly connected component"
        end
end
```

Fig. 5.15. Procedure to compute LOWLINK.

The LOWLINK computation occurs at lines 4, 9, and 11. At line 4 LOWLINK[v] is initialized to the depth-first number of vertex v. At line 9 LOWLINK[v] is set to LOWLINK[w], if for some son w, LOWLINK[w] is less than the current value of LOWLINK[v]. At line 10 we determine whether (v, w) is either a back edge or cross edge and we check to see whether the strongly connected component containing w has been found. If not, then the root of the strongly connected component containing w is an ancestor of v. Perforce, at line 11 we set LOWLINK[v] to the depth-first number of w if it does not already have a lower value.

We now give the entire algorithm to find the strongly connected components of a directed graph.

Algorithm 5.4. Strongly connected components of a directed graph.

Input. A directed graph $G = (V, E)$.

Output. A list of the strongly connected components of G.

Method

```
begin
    COUNT ← 1;
    for all v in V do mark v "new";
    initialize STACK to empty;
    while there exists a vertex v marked "new" do SEARCHC(v)
end □
```

Example 5.8. Consider the depth-first spanning forest of Fig. 5.13, which is reproduced as Fig. 5.16 with LOWLINK computed. The first SEARCHC call to terminate is SEARCHC(3). When we examine the edge (3, 1), we set LOWLINK[3] to MIN(1, 3) = 1. On returning to SEARCHC(2), we set LOWLINK[2] to MIN(2, LOWLINK[3]) = 1. Then we call SEARCHC(4), which considers edge (4, 3). Since $3 < 4$ and 3 is still on STACK, we set LOWLINK[4] to MIN(3, 4) = 3.

Then we return to SEARCHC(2) and set LOWLINK[2] to the minimum of LOWLINK[4] and the current value of LOWLINK[2], which is 1. Since the latter is smaller, no change to LOWLINK[2] occurs. We return to

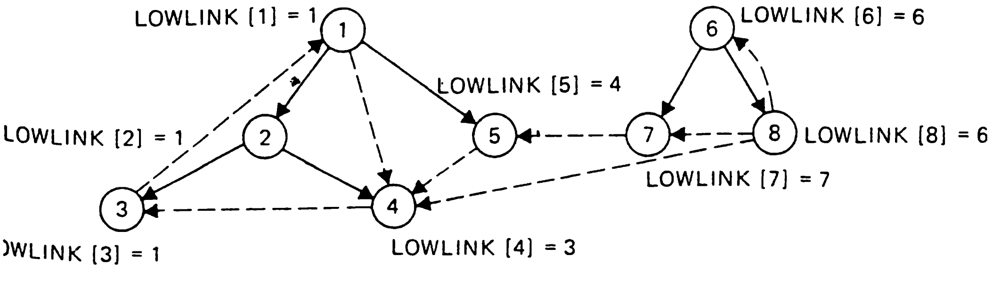


Fig. 5.16 A spanning forest with LOWLINK computed.

SEARCHC(1), setting LOWLINK[1] to MIN(1, LOWLINK[2]) = 1. If we then consider edge (1, 4), we do nothing since (1, 4) is a forward edge and the condition of line 10 of SEARCHC is not met because 4 > 1.

Next, we call SEARCHC(5), and cross edge (5, 4) causes us to set LOWLINK[5] to 4, since 4 < 5 and 4 is on STACK. When we again return to SEARCHC(1), we set LOWLINK[1] to the minimum of its former value 1 and LOWLINK[5], which yields 1.

Then, since all edges out of 1 have been considered, and LOWLINK[1] = 1, we discover that 1 is the root of a strongly connected component. This component consists of 1 and all vertices above 1 on the stack. Since 1 was the first vertex visited, vertices 2, 3, 4, and 5 are all above 1, in that order. Thus the stack is emptied and the list of vertices 1, 2, 3, 4, 5 is printed as a strongly connected component of G.

The remaining strongly connected components are {7} and {6, 8}. We leave it to the reader to complete the calculation of LOWLINK and the strongly connected components, starting at vertex 6. Note that the roots of the strongly connected components were last encountered in the order 1, 7, 6. □

Theorem 5.4. Algorithm 5.4 correctly finds the strongly connected components of G in $O(\text{MAX}(n, e))$ time on an n-vertex, e-edge directed graph G.

Proof. It is easy to check that the time spent by one call of SEARCHC(v), exclusive of recursive calls to SEARCHC, is a constant plus time proportional to the number of edges leaving vertex v. Thus all calls to SEARCHC together require time proportional to the number of vertices plus the number of edges, as SEARCHC is called only once at any vertex. The portions of Algorithm 5.4 other than SEARCHC can clearly be implemented in time $O(n)$. Thus the time bound is proven.

To prove correctness it suffices to prove, by induction on the number of calls to SEARCHC that have terminated, that when SEARCH(v) terminates, LOWLINK[v] is correctly computed. By lines 12–16 of SEARCHC, v is made the root of a strongly connected component if and only if we have LOWLINK[v] = v. Moreover, the vertices printed out are exactly those descendants of v which are not in components whose roots have been found before v, as required by Lemma 5.8. That is, the vertices above v on the stack are descendants of v, and their roots have not been found before v since they are still on the stack.

To prove LOWLINK is computed correctly, note that there are two places in Fig. 5.15 where LOWLINK[v] could receive a value less than v, that is, at lines 9 and 11 of SEARCHC. In the first case, w is a son of v, and LOWLINK[w] < v. Then there is a vertex x = LOWLINK[w] that can be reached from a descendant y of w by a cross or back edge. Moreover, the root r of the strongly connected component containing x is an ancestor of w. Since $x < v$, we have $r < v$, so r is a proper ancestor of v. Thus we see LOWLINK[v] should be at least as small as LOWLINK[w].

In the second case, at line 11, there is a cross or back edge from v to a vertex $w < v$ whose strongly connected component C has not yet been found. The call of SEARCHC on the root r of C has not terminated, so r must be an ancestor of v. (Since $r \leq w < v$, either r is to the left of v or r is an ancestor of v. But if r were to the left of v, SEARCHC(r) would have terminated.) Again it follows that LOWLINK[v] should be at least as low as w.

We must still show that SEARCHC computes LOWLINK[v] to be as low as it should be. Suppose in contradiction that there is a descendant x of v with a cross or back edge from x to y, and the root r of the strongly connected component containing y is an ancestor of v. We must show that LOWLINK is set at least as low as y.

CASE 1. $x = v$. We may assume by the inductive hypothesis and Lemma 5.9 that all strongly connected components found so far are correct. Then y must still be on STACK, since SEARCH(r) has not terminated. Thus line 11 sets LOWLINK[v] to y or lower.

CASE 2. $x \neq v$. Let z be the son of v of which x is a descendant. Then by the inductive hypothesis, when SEARCHC(z) terminates, LOWLINK[z] has been set to y or lower. At line 9 LOWLINK[v] is set this low, if it is not already lower. □

5.6 PATH-FINDING PROBLEMS

In this section we consider two frequently occurring problems having to do with paths between vertices. In what follows let G be a directed graph. The graph G^* which has the same vertex set as G, but has an edge from v to w if and only if there is a path (of length 0 or more) from v to w in G, is called the (*reflexive* and) *transitive closure* of G.

A problem closely related to finding the transitive closure of a graph is the *shortest-path problem*. Associate with each edge e of G a nonnegative *cost* $c(e)$. The *cost of a path* is defined to be the sum of the costs of the edges in the path. The shortest-path problem is to find for each ordered pair of vertices (v, w) the lowest cost of any path from v to w.

It turns out that the ideas behind the best algorithms known for both the transitive closure and shortest-path problems are (easy) special cases of the problem of finding the (infinite) set of all paths between each pair of vertices. To discuss the problem in its generality, we introduce a special algebraic structure.

Definition. A *closed semiring* is a system $(S, +, \cdot, 0, 1)$, where S is a set of elements, and $+$ and $\cdot$ are binary operations on S, satisfying the following five properties:

1. $(S, +, 0)$ is a *monoid*, that is, it is *closed* under + [i.e., $a + b \in S$ for all a and b in S], + is *associative* [i.e., $a + (b + c) = (a + b) + c$ for all a, b, c in S], and 0 is an *identity* [i.e., $a + 0 = 0 + a = a$ for all a in S]. Likewise, $(S, \cdot, 1)$ is a monoid. We also assume 0 is an *annihilator*, i.e., $a \cdot 0 = 0 \cdot a = 0$.
2. + is *commutative*, i.e., $a + b = b + a$, and *idempotent*, i.e., $a + a = a$.
3. $\cdot$ *distributes* over +, that is, $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$.
4. If $a_1, a_2, \ldots, a_i, \ldots$ is a countable sequence of elements in S, then $a_1 + a_2 + \cdots + a_i + \cdots$ exists and is unique. Moreover, associativity, commutativity, and idempotence apply to infinite as well as finite sums.
5. $\cdot$ must distribute over countably infinite sums as well as finite ones (this does not follow from property 3). Thus (4) and (5) imply

$$\left(\sum_i a_i\right) \cdot \left(\sum_j b_j\right) = \sum_{i,j} a_i \cdot b_j = \sum_i \left(\sum_j (a_i \cdot b_j)\right).$$

Example 5.9. The following three systems are closed semirings.

1. Let $S_1 = (\{0, 1\}, +, \cdot, 0, 1)$ with addition and multiplication tables as follows:

+	0	1
0	0	1
1	1	1

$\cdot$	0	1
0	0	0
1	0	1

Then properties 1–3 are easy to verify. For properties 4 and 5, note that a countable sum is 0 if and only if all terms are 0.

2. Let $S_2 = (R, \text{MIN}, +, +\infty, 0)$, where R is the set of nonnegative reals including $+\infty$. It is easy to verify that $+\infty$ is the identity under MIN and 0 the identity under +.
3. Let Σ be a finite alphabet (i.e., a set of symbols), and let $S_3 = (F_\Sigma, \cup, \cdot, \emptyset, \{\epsilon\})$, where F_Σ is the family of sets of finite-length strings of symbols from Σ, including ϵ, the empty string (i.e., the string of length 0). Here the first operator is set union and $\cdot$ denotes set concatenation.† The $\cup$ identity is $\emptyset$ and the $\cdot$ identity is $\{\epsilon\}$. The reader may verify properties 1–3. For properties 4 and 5, we must observe that

† The *concatenation* of sets A and B, denoted $A \cdot B$, is the set $\{x \mid x = yz,\ y \in A \text{ and } z \in B\}$.

countable unions behave as they should if we define $x \in (A_1 \cup A_2 \cup \cdots)$ if and only if $x \in A_i$ for some i. □

A unary operation, denoted * and called *closure*, is central to our analysis of closed semirings. If $(S, +, \cdot, 0, 1)$ is a closed semiring, and $a \in S$, then we define a^* to be $\Sigma_{i=0}^{\infty} a^i$, where $a^0 = 1$ and $a^i = a \cdot a^{i-1}$. That is, a^* is the infinite sum $1 + a + a \cdot a + a \cdot a \cdot a + \cdots$. Note that property 4 of the definition of a closed semiring assures that $a^* \in S$. Properties 4 and 5 imply $a^* = 1 + a \cdot a^*$. Note that $0^* = 1^* = 1$.

Example 5.10. Let us refer to the semirings S_1, S_2, and S_3 of Example 5.9. For S_1, $a^* = 1$ for $a = 0$ or 1. For S_2, $a^* = 0$ for all a in R. For S_3, $A^* = \{\epsilon\} \cup \{x_1 x_2 \cdots x_k | k \geq 1 \text{ and } x_i \in A \text{ for } 1 \leq i \leq k\}$ for all $A \in F_\Sigma$. For example, $\{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$, that is, all strings of a's and b's including the empty string. In fact, $F_\Sigma = \mathscr{P}(\Sigma^*)$, where $\mathscr{P}(X)$ denotes the power set of set X. □

Now, let us suppose we have a directed graph $G = (V, E)$ in which each edge is labeled by an element of some closed semiring $(S, +, \cdot, 0, 1)$.[†] We define the *label of a path* to be the product $(\cdot)$ of the labels of the edges in the path, taken in order. As a special case, the label of the path of zero length is 1 (the $\cdot$ identity of the semiring). For each pair of vertices (v, w), we define $c(v, w)$ to be the sum of the labels of all the paths between v and w. We shall refer to $c(v, w)$ as the *cost* of going from v to w. By convention, the sum over an empty set of paths is 0 (the + identity of the semiring). Note that if G has cycles, there may be an infinity of paths between v and w, but the axioms of a closed semiring assure us that $c(v, w)$ will be well defined.

Example 5.11. Consider the directed graph in Fig. 5.17, in which each edge has been labeled by an element from the semiring S_1 of Example 5.9. The label of the path v, w, x is $1 \cdot 1 = 1$. The simple cycle from w to w has label $1 \cdot 0 = 0$. In fact, every path of length greater than zero from w to w has label 0. However, the path of zero length from w to w has cost 1. Consequently, $c(w, w) = 1$. □

We now give an algorithm to compute $c(v, w)$ for all pairs of vertices v and w. The basic unit-time steps of the algorithm are the operations +, $\cdot$, and

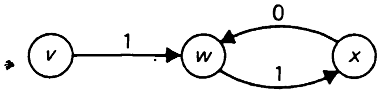


Fig. 5.17 A labeled directed graph.

[†] The reader should not miss the analogy between such a situation and a nondeterministic finite automaton (see Hopcroft and Ullman [1969] or Aho and Ullman [1972]), as we shall discuss in Section 9.1. There, the vertices are states and the edge labels are symbols from some finite alphabet.

* of an arbitrary closed semiring. Of course, for structures such as the set of all sets of strings over an alphabet, it is not clear that such operations can be implemented at all, let alone in "unit time." However, for the semirings of which we shall make use, the operations will be easy to perform.

Algorithm 5.5. Computation of costs between vertices.

Input. A directed graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and a labeling function l: $(V \times V) \rightarrow S$, where $(S, +, \cdot, 0, 1)$ is a closed semiring. We take $l(v_i, v_j) = 0$ if (v_i, v_j) is not in E.

Output. For all i and j between 1 and n, the element $c(v_i, v_j)$ of S which is equal to the sum over all paths from v_i to v_j of the label of that path.

Method. We compute C^k_{ij} for all $1 \leq i \leq n$, $1 \leq j \leq n$, and $0 \leq k \leq n$. The intention is that C^k_{ij} should be the sum of the labels of all paths from v_i to v_j such that all vertices on the path, except possibly the endpoints, are in the set $\{v_1, v_2, \ldots, v_k\}$. For example, the path v_9, v_3, v_8 is considered in C^4_{98} and C^3_{98} but not in C^2_{98}. The algorithm is as follows.

```
      begin
1.        for $i \leftarrow 1$ until $n$ do $C^0_{ii} \leftarrow 1 + l(v_i, v_i)$;
2.        for $1 \leq i, j \leq n$ and $i \neq j$ do $C^0_{ij} \leftarrow l(v_i, v_j)$;
3.        for $k \leftarrow 1$ until $n$ do
4.            for $1 \leq i, j \leq n$ do
5.                $C^k_{ij} \leftarrow C^{k-1}_{ij} + C^{k-1}_{ik} \cdot (C^{k-1}_{kk})^* \cdot C^{k-1}_{kj}$;
6.        for $1 \leq i, j \leq n$ do $c(v_i, v_j) \leftarrow C^n_{ij}$
      end □
```

Theorem 5.5. Algorithm 5.5 uses $O(n^3)$ +, $\cdot$, and * operations from the semiring and computes $c(v_i, v_j)$ for $1 \leq i, j \leq n$.

Proof. It is easy to check that line 5 is executed n^3 times, requiring four operations each time, and that the **for** loops of lines 1, 2, and 6 are iterated at most n^2 times each. Thus $O(n^3)$ operations suffice.

To show correctness, we must prove by induction on k that C^k_{ij} is the sum, over all paths from v_i to v_j with no intermediate vertex (excluding the endpoints) of index higher than k, of the label of such a path. The basis, $k = 0$, is trivial by lines 1 and 2, since any such path is of zero length, or consists of a single edge. The induction step follows from line 5, since a path from v_i to v_j with no intermediate vertex higher than k either

i) has no intermediate vertex higher than $k - 1$ (the term C^{k-1}_{ij}), or

ii) goes from v_i to v_k, then from v_k to v_k some number of times (possibly 0), and finally from v_k to v_j, all with no intermediate vertex higher than $k - 1$ [the term $C^{k-1}_{ik} \cdot (C^{k-1}_{kk})^* \cdot C^{k-1}_{kj}$].

The laws of a closed semiring insure that line 5 correctly computes the sum of the labels of all these paths. □

5.7 A TRANSITIVE CLOSURE ALGORITHM

We specialize Algorithm 5.5 to two interesting cases. The first is to the closed semiring S_1 described in Example 5.9. In S_1 addition and multiplication are easy to perform, and $*$ is also easy, since $0^* = 1^* = 1$. In fact, since 1 is the $\cdot$ identity, we may replace line 5 of Algorithm 5.5 by†

$$C^k_{ij} \leftarrow C^{k-1}_{ij} + C^{k-1}_{ik} \cdot C^{k-1}_{kj}. \qquad (5.4)$$

To compute the reflexive-transitive closure of a graph, we define the labeling function

$$l(v, w) = \begin{cases} 1, & \text{if } (v, w) \text{ is an edge,} \\ 0, & \text{if not.} \end{cases}$$

Then $c(v, w) = 1$ if and only if there is a path of length 0 or more from v to w, as may easily be checked, using the laws of the closed semiring $\{0, 1\}$.

Example 5.12. Consider the graph of Fig. 5.18, ignoring the numbers on the edges temporarily. The labeling function $l(v_i, v_j)$ is given as follows.

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	0	0
v_3	0	1	0

$l(v_i, v_j)$

Then, line 1 of Algorithm 5.5 sets $C^0_{11} = C^0_{22} = C^0_{33} = 1$. Line 2 sets $C^0_{ij} = l(v_i, v_j)$ for $i \neq j$, so we have the following values for the C^0_{ij}'s.

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	0
v_3	0	1	1

C^0_{ij}

† It is natural to interpret this observation as saying "it is sufficient to consider only cycle-free paths." However, it should be noted that with (5.4) in place of line 5, Algorithm 5.5 will compute sums over sets of paths which include all cycle-free paths, but some others as well.

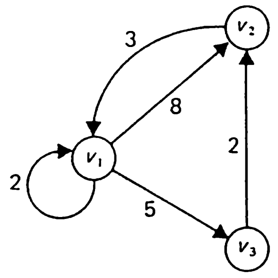


Figure 5.18

We now set $k = 1$ and perform the loop of lines 4 and 5, with (5.4) in place of line 5. For example, $C^1_{23} = C^0_{23} + C^0_{21} \cdot C^0_{13} = 0 + 1 \cdot 1 = 1$. The tables giving C^1_{ij}, C^2_{ij}, and C^3_{ij} are shown in Fig. 5.19. □

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	1
v_3	0	1	1

C^1_{ij}

	v_1	v_2	v_3
v_1	1	1	1
v_2	1	1	1
v_3	1	1	1

$C^2_{ij} = C^3_{ij} = c(v_i, v_j)$

Fig. 5.19. Values of C^k_{ij}.

5.8 A SHORTEST-PATH ALGORITHM

For computation of shortest paths, we use the second closed semiring discussed in Example 5.9, namely the nonnegative reals with $+\infty$. Recall that the additive operation is MIN and the multiplicative operation is addition in the reals. That is, we are considering the structure (R, MIN, $+$, $+\infty$, 0). We observed in Example 5.10 that $a^* = 0$ for all $a \in R$, so we may again delete the * operation on line 5 of Algorithm 5.5, replacing that line by

$$C^k_{ij} \leftarrow \text{MIN}(C^{k-1}_{ij}, C^{k-1}_{ik} + C^{k-1}_{kj}). \tag{5.5}$$

Informally, (5.5) says that the shortest path from v_i to v_j which passes through no vertex higher than v_k is the shorter of

i) the shortest path which passes through no vertex higher than v_{k-1}, and
ii) the path which in as short a distance as possible goes from v_i to v_k and then to v_j, passing through no vertex higher than v_{k-1} between these points.

To convert Algorithm 5.5 into a shortest-path algorithm, let $l(v_i, v_j)$ be

$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 2 & 8 & 5 \\ v_2 & 3 & \infty & \infty \\ v_3 & \infty & 2 & \infty \end{array} \qquad \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 0 & 8 & 5 \\ v_2 & 3 & 0 & \infty \\ v_3 & \infty & 2 & 0 \end{array}$$

$l(v_i, v_j)$ $\qquad$ C^0_{ij}

$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 0 & 8 & 5 \\ v_2 & 3 & 0 & 8 \\ v_3 & \infty & 2 & 0 \end{array} \qquad \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 0 & 8 & 5 \\ v_2 & 3 & 0 & 8 \\ v_3 & 5 & 2 & 0 \end{array}$$

C^1_{ij} $\qquad$ C^2_{ij}

$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 0 & 7 & 5 \\ v_2 & 3 & 0 & 8 \\ v_3 & 5 & 2 & 0 \end{array}$$

$C^3_{ij} = c(v_i, v_j)$

Fig. 5.20. Shortest-path calculation.

the cost of edge (v_i, v_j), if one exists, and $+\infty$ otherwise. Then substitute (5.5) for line 5 and by Theorem 5.5, the value of $c(v_i, v_j)$ produced by Algorithm 5.5 will be the minimum over all paths between v_i and v_j of the cost (i.e., sum of costs of edges) of the path.

Example 5.13. Consider the graph of Fig. 5.18 again. Let the label of each edge be as indicated there. Then Fig. 5.20 shows the functions l, C^0_{ij}, C^1_{ij}, C^2_{ij}, and C^3_{ij}. For example, $C^3_{12} = \mathrm{MIN}(C^2_{12}, C^2_{13} + C^2_{32}) = \mathrm{MIN}(8, 5+2) = 7$. □

5.9 PATH PROBLEMS AND MATRIX MULTIPLICATION

Let $(S, +, \cdot, 0, 1)$ be a closed semiring. We can define $n \times n$ matrices of elements of S with the usual sum and product. That is, let A, B, and C have elements a_{ij}, b_{ij}, and c_{ij}, respectively, for $1 \le i, j \le n$. Then $C = A + B$ means $c_{ij} = a_{ij} + b_{ij}$ for all i and j, and $C = A \cdot B$ means $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$ for all i and j. It is easy to check the following.

Lemma 5.10. Let $(S, +, \cdot, 0, 1)$ be a closed semiring and M_n the set of $n \times n$ matrices over S. Let $+_n$ and $\cdot_n$ be matrix addition and multiplication, respectively, let 0_n be the matrix each of whose elements is 0 and let I_n be the matrix with 1's on the diagonal and 0's elsewhere. Then $(M_n, +_n, \cdot_n, 0_n, I_n)$ is a closed semiring.

Proof. Exercise. □

Let $G = (V, E)$ be a directed graph, where $V = \{v_1, v_2, \ldots, v_n\}$. Suppose $l:(V \times V) \rightarrow S$ is a labeling function as in Algorithm 5.5. Let A_G be the $n \times n$ matrix whose ijth entry is $l(v_i, v_j)$. The next lemma relates the computation performed by Algorithm 5.5 to the computation of the closure A^*_G in the semiring M_n of $n \times n$ matrices over S.

Lemma 5.11. Let G and A_G be as above. Then the ijth element of A^*_G is $c(v_i, v_j)$.

Proof. $A^*_G = \Sigma_{i=0}^{\infty} A^i_G$, where $A^0_G = I_n$ and $A^i_G = A_G \cdot A^{i-1}_G$ for $i \geq 1$. It is straightforward to show by induction on i that the ijth element of A^k_G is the sum, over all paths of length k from v_i to v_j, of the cost of that path. It follows immediately that the ijth element of A^*_G is the sum of the costs of all paths from v_i to v_j. □

As a consequence of Lemma 5.11, Algorithm 5.5 can be regarded as an algorithm to compute the closure of a matrix.. It was claimed in Theorem 5.5 that Algorithm 5.5 requires $O(n^3)$ *scalar* operations (i.e., operations $+$, $\cdot$, and $*$ taken from the semiring). The obvious algorithm for matrix multiplication likewise requires $O(n^3)$ scalar operations. We shall show that when the scalars come from a closed semiring, computing the product of two matrices is computationally equivalent to computing the closure of a matrix. That is, given any algorithm for computing the product, we can construct an algorithm for closure and vice versa, and the order of the execution times will be the same. This result will be better appreciated in Chapter 6, where we show that when the scalars form a ring, fewer than $O(n^3)$ operations suffice for matrix multiplication. The construction is in two parts.

Theorem 5.6. If the closure of an arbitrary $n \times n$ matrix over a closed semiring can be computed in time $T(n)$, where $T(n)$ satisfies $T(3n) \leq 27\ T(n)$,† then there exists a constant c such that the time $M(n)$ to multiply two $n \times n$ matrices A and B satisfies $M(n) \leq cT(n)$.

Proof. Assume we wish to compute the product $A \cdot B$, where A and B are

† This is a likely assumption, since $T(n)$ is at worst $O(n^3)$. Actually any constant would do in place of 27 in the statement of the theorem.

$n \times n$ matrices. First form the $3n \times 3n$ matrix

$$X = \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}$$

and find the closure of X. Note that

$$X^2 = \begin{pmatrix} 0 & 0 & A \cdot B \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad X^3 = X^4 = \cdots = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Thus

$$X^* = I_{3n} + X + X^2 = \begin{pmatrix} I_n & A & A \cdot B \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Since $A \cdot B$ can be read from the upper right corner, we conclude that $M(n) \leq T(3n) \leq 27T(n)$. □

The proof of Theorem 5.6 has the following graphical interpretation. Consider a graph G with $3n$ vertices $\{1, 2, \ldots, 3n\}$ partitioned into three sets $V_1 = \{1, 2, \ldots, n\}$, $V_2 = \{n+1, n+2, \ldots, 2n\}$, and $V_3 = \{2n+1, 2n+2, \ldots, 3n\}$. Assume each edge of G has either its tail in V_1 and its head in V_2 or its tail in V_2 and its head in V_3. Such a graph is illustrated in Fig. 5.21. Let A and B be $n \times n$ matrices where the ijth element of A is the label of the edge from vertex i to vertex $n+j$ and the ijth element of

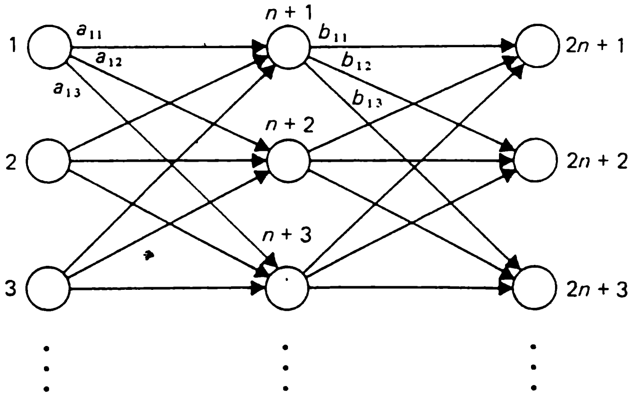


Fig. 5.21 Graphical interpretation of Theorem 5.6.

B is the label of the edge from vertex $n+i$ to vertex $2n+j$. The ijth element of the product $A \cdot B$ happens to be the sum of the labels of the paths from vertex i to vertex $2n+j$ since each such path consists of an edge from vertex i to some vertex k, $n < k \leq 2n$, followed by an edge from vertex k to vertex $2n+j$. Thus the sum of the labels of all paths from i to $2n+j$ is the same as the ijth element of $A \cdot B$.

We now proceed to show the converse of Theorem 5.6. Given an algorithm for matrix multiplication we can find a closure algorithm that is of the same speed, neglecting constant factors.

Theorem 5.7. If the product of two arbitrary $n \times n$ matrices over a closed semiring can be computed in time $M(n)$, where $M(n)$ satisfies $M(2n) \geq 4M(n)$, then there exists a constant c such that the time $T(n)$ to compute the closure of an arbitrary matrix satisfies $T(n) \leq cM(n)$.

Proof. Let X be an $n \times n$ matrix. First consider the case in which n is a power of 2, say 2^k. Partition X into four matrices of size $2^{k-1} \times 2^{k-1}$,

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

Using Lemma 5.11, we may suppose that matrix X represents a graph $G = (V, E)$, where the vertices are partitioned into two sets V_1 and V_2 of size 2^{k-1}. Matrix A represents edges between vertices in V_1, and matrix D represents edges between vertices in V_2. Matrix B represents the edges from vertices in V_1 to vertices in V_2 and matrix C represents edges from vertices in V_2 to vertices in V_1. The arrangement is shown in Fig. 5.22.

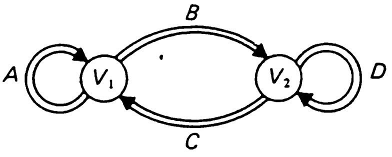


Fig. 5.22 The graph G.

Now a path from v_1 to v_2 with both vertices in V_1 is of one of two forms. Either

1. the path never leaves V_1, or
2. for some $k \geq 1$ there are vertices w_i, x_i, y_i, and z_i, for $1 \leq i \leq k$, where the w's and z's are in V_1 and the x's and y's in V_2, such that the path goes from v_1 to w_1 within V_1, then along an edge to x_1, then along a path in V_2 to y_1, then along an edge to z_1, then along a path in V_1 to w_2, and so on, to z_k, whereupon the path stays within V_1 to v_2. The pattern is shown in Fig. 5.23.

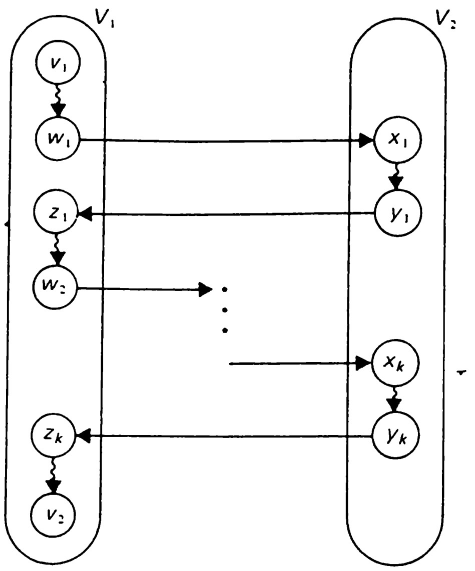


Fig. 5.23 Paths satisfying condition 2.

The sum of the labels of the paths satisfying condition 1 or 2 is easily seen to be $(A + B \cdot D^* \cdot C)^*$. That is, $B \cdot D^* \cdot C$ represents the paths from w_i to x_i to y_i to z_i in Fig. 5.23. Thus $A + B \cdot D^* \cdot C$ represents the paths between vertices in V_1 that either are a single edge or jump directly to V_2, stay within V_2 for a while, and then jump back to V_1. All paths between vertices in V_1 can be represented as the succession of paths represented by $A + B \cdot D^* \cdot C$. Thus $(A + B \cdot D^* \cdot C)^*$ represents all paths between vertices of V_1. Hence the upper left quadrant of X^* has $(A + B \cdot D^* \cdot C)^*$ in it.

Let $E = (A + B \cdot D^* \cdot C)^*$. By similar reasoning, we may write each of the four quadrants of X^* as:

$$X^* = \begin{pmatrix} E & E \cdot B \cdot D^* \\ D^* \cdot C \cdot E & D^* + D^* \cdot C \cdot E \cdot B \cdot D^* \end{pmatrix} = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

We may compute these four quadrants E, F, G, and H by the sequence of steps:

$$\begin{aligned}
T_1 &= D^*, \\
T_2 &= B \cdot T_1, \\
E &= (A + T_2 \cdot C)^*, \\
F &= E \cdot T_2, \\
T_3 &= T_1 \cdot C, \\
G &= T_3 \cdot E, \\
H &= T_1 + G \cdot T_2.
\end{aligned}$$

The above steps require two closures, six multiplications, and two additions of $2^{k-1} \times 2^{k-1}$ matrices. Thus we may write:

$$\begin{aligned} &T(1) = 1, \\ &T(2^k) \leq 2T(2^{k-1}) + 6M(2^{k-1}) + 2 \cdot 2^{2k-2},\ k > 1. \end{aligned} \tag{5.6}$$

The three terms on the right of (5.6) represent the costs of the closures, multiplications, and additions, respectively. We claim that there exists a constant c such that $T(2^k) \leq cM(2^k)$ satisfies (5.6) for all k. The basis, $k = 0$, is trivial since we may choose c as large as we like. For the inductive step, assume for some c that $T(2^{k-1}) \leq cM(2^{k-1})$. From the hypothesis of the theorem [$M(2n) \geq 4M(n)$] we have $M(2^{k-1}) \geq 2^{2k-2}$. [Note $M(1) = 1$]. Thus from (5.6)

$$T(2^k) \leq (2c + 8)M(2^{k-1}).$$

Again, by the theorem hypothesis, $M(2^{k-1}) \leq \frac{1}{4}M(2^k)$, so

$$T(2^k) \leq (\tfrac{1}{2}c + 2)M(2^k). \tag{5.7}$$

If we choose $c \geq 4$, (5.7) implies $T(2^k) \leq cM(2^k)$, as was to be shown.

If n is not a power of 2 we can embed X in a larger matrix, of dimension which is a power of 2, of the form

$$\begin{pmatrix} X & 0 \\ 0 & I \end{pmatrix},$$

where I is an identity matrix of the smallest possible size. This will at most double the size of the matrix and hence will increase the constant by at most a factor of 8. Therefore there exists a new constant c' such that $T(n) \leq c'M(n)$ for all n, whether or not n is a power of 2. □

Corollary 1. The time necessary to compute the closure of a Boolean matrix is of the same order as the time to compute the product of two Boolean matrices of the same size.

In Chapter 6 we shall see asymptotically faster algorithms for computing the product of Boolean matrices and thereby show that the transitive closure of a graph can be computed in time less than $O(n^3)$.

Corollary 2. The time necessary to compute the closure of a matrix of nonnegative reals, with operations MIN and + playing the role of addition and multiplication of scalars, is of the same order as the time to compute the product of two matrices of this type.

At present no known method for the all-pairs shortest-path problem uses less than cn^3 time, for some constant c.

) SINGLE-SOURCE PROBLEMS

nany applications we may wish only to find shortest paths from one vertex : *source*). In fact, it may be desirable to find only the shortest path ween two particular vertices, but there is no known algorithm for that blem that is more efficient in the worst case than the best single-source rithm.

For single-source problems we know of no unifying concept analogous to sed semirings and Algorithm 5.5. Moreover, if we only wish to know to ch vertices there exists a path from the source, the problem is trivial and be solved by a number of algorithms which operate in $O(e)$ on an e-edge ph. Since any algorithm which does not "look at" all edges cannot be cort, it is not hard to believe that $O(e)$ is the best we can hope for.

When it comes to finding shortest paths from a single source, the situan changes. While there is no reason to suppose that more than $O(e)$ time uld be required, no such algorithm is known. We shall discuss an $O(n^2)$ rithm, which works by constructing a set S of vertices whose shortest disce from the source is known. At each step, we add to S that remaining tex v whose distance from the source is shortest of the remaining vertices. ll edges have nonnegative costs, then we can be sure the path from the rce to v passes only through vertices in S. Thus it is only necessary to ord for each vertex v the shortest distance from the source to v along ath that passes only through vertices of S. We now give the algorithm nally.

orithm 5.6. Single-source shortest path (Dijkstra's algorithm).

ut. A directed graph $G = (V, E)$, a source $v_0 \in V$, and a function l from es to nonnegative reals. We take $l(v_i, v_j)$ to be $+\infty$ if (v_i, v_j) is not an e, $v_i \neq v_j$, and $l(v, v) = 0$.

tput. For each $v \in V$, the minimum over all paths P from v_0 to v of the of the labels of the edges of P.

thod. We construct a set $S \subseteq V$ such that the shortest path from the rce to each vertex v in S lies wholly in S. The array $D[v]$ contains the t of the current shortest path from v_0 to v passing only through vertices of The algorithm is given in Fig. 5.24. □

mple 5.14. Consider the graph of Fig. 5.25. Initially, $S = \{v_0\}$, $v_0] = 0$, and $D[v_i]$ is 2, $+\infty$, $+\infty$, 10 for $i = 1, 2, 3, 4$, respectively. At the iteration of the loop of lines 4–8, $w = v_1$ is selected, since $D[v_1] = 2$ is minimum value of D. Then we set $D[v_2] = \text{MIN}(+\infty, 2 + 3) = 5$ and $v_4] = \text{MIN}(10, 2 + 7) = 9$. The sequence of values of D and other comations of Algorithm 5.6 are summarized in Fig. 5.26. □

```
    begin
1.      $S \leftarrow \{v_0\}$;
2.      $D[v_0] \leftarrow 0$;
3.      for each $v$ in $V - \{v_0\}$ do $D[v] \leftarrow l(v_0, v)$;
4.      while $S \neq V$ do
            begin
5.              choose a vertex $w$ in $V - S$ such that $D[w]$ is a minimum;
6.              add $w$ to $S$;
7.              for each $v$ in $V - S$ do
8.                  $D[v] \leftarrow \text{MIN}(D[v], D[w] + l(w, v))$
            end
    end
```

Fig. 5.24. Dijkstra's algorithm.

Theorem 5.8. Algorithm 5.6 computes the cost of the shortest path from v_0 to each vertex and requires $O(n^2)$ time.

Proof. The **for** loop of lines 7–8 requires $O(n)$ steps, as does the selection of w at line 5. These are the dominant costs of the **while** loop of lines 4–8. In turn, the latter is executed $O(n)$ times, for a total cost of $O(n^2)$. Lines 1–3 clearly require $O(n)$ time.

For correctness, we must prove by induction on the size of S that for each v in S, $D[v]$ is equal to the length of a shortest path from v_0 to v. Moreover, for all $v \in V - S$, $D[v]$ is the length of the shortest path from v_0 to v that lies wholly within S, except for v itself.

Basis. $\|S\| = 1$. The shortest path from v_0 to itself has length 0 and a path from v_0 to v, wholly within S except for v, consists of the single edge (v_0, v). Thus lines 2 and 3 correctly initialize the D array.

Inductive step. Suppose vertex w is chosen on line 5. If $D[w]$ is not the length of a shortest path from v_0 to w, then there must exist a shorter path

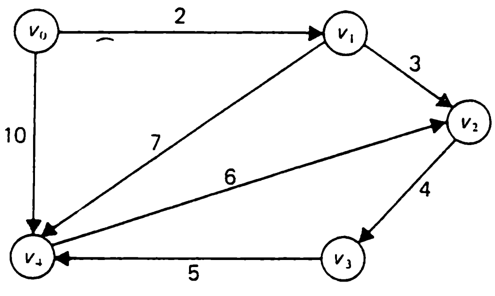


Fig. 5.25 A graph with labeled edges.

Iteration	S	w	$D[w]$	$D[v_1]$	$D[v_2]$	$D[v_3]$	$D[v_4]$
Initial	$\{v_0\}$	—	—	2	$+\infty$	$+\infty$	10
1	$\{v_0, v_1\}$	v_1	2	2	5	$+\infty$	9
2	$\{v_0, v_1, v_2\}$	v_2	5	2	5	9	9
3	$\{v_0, v_1, v_2, v_3\}$	v_3	9	2	5	9	9
4	All	v_4	9	2	5	9	9

Fig. 5.26. Computation of Algorithm 5.6 on graph of Fig. 5.25.

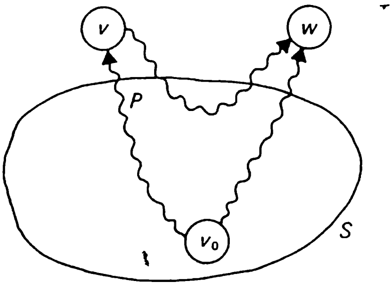


Fig. 5.27 Paths to vertex v.

P. The path P must contain some vertex other than w which is not in S. Let v be the first such vertex on P. But then the distance from v_0 to v is shorter than $D[w]$, and moreover, the shortest path to v lies wholly within S, except for v itself. (See Fig. 5.27.) Thus by the inductive hypothesis, $D[v] < D[w]$ when w was selected, a contradiction. We conclude that the path P does not exist and $D[w]$ is the length of the shortest path from v_0 to w.

The second part of the inductive hypothesis, that $D[w]$ remains correct, is obvious because of line 8. □

5.11 DOMINATORS IN A DIRECTED ACYCLIC GRAPH: PUTTING THE CONCEPTS TOGETHER

In this chapter we have seen several techniques, such as depth-first search and judicious ordering of computations, for designing efficient graph algorithms. In Chapter 4 we studied a number of data structures that were useful in representing sets being manipulated by various operations. We conclude this chapter by an example which illustrates how efficient algorithms can be designed by combining the data structures of Chapter 4 with the graph techniques of this chapter. In particular, we shall use the off-line MIN algorithm, the disjoint-set union algorithm, and 2–3 trees in conjunction with a depth-first search to find the dominators of a directed acyclic graph.

A directed graph $G = (V, E)$ is *rooted* at vertex r if there is a path from r to every vertex in V. For the remainder of this section we shall assume that $G = (V, E)$ is a rooted directed acyclic graph with root r.

Vertex v is a *dominator* of vertex w if every path from the root to w contains v. Every vertex is a dominator of itself, and the root dominates every vertex. The set of dominators of a vertex w can be linearly ordered by their order of occurrence on a shortest path from the root to w. The dominator of w closest to w (other than w itself) is called the *immediate dominator* of w. Since the dominators of each vertex are linearly ordered, the relation "v dominates w" can be represented by a tree with root r called the *dominator tree* for G. Computing dominators is useful in code optimization problems (see [Aho and Ullman, 1973] and [Hecht, 1973]).

We shall develop an $O(e \log e)$ algorithm to compute the dominator tree for a rooted directed acyclic graph with e edges. The main purpose of this algorithm is to illustrate how the techniques of this and the previous chapter can be combined. The algorithm is based on the next three lemmas.

Let $G = (V, E)$ be a graph and $G'' = (V', E')$ a subgraph of G. If $(a, b) \in E'$, we write $a \xrightarrow[G']{} b$. We use $\xrightarrow[G']{+}$ and $\xrightarrow[G']{*}$, respectively, to denote the transitive closure and the reflexive transitive closure of $\xrightarrow[G']{}$. If $(a, b) \in E$, we write $a \Longrightarrow b$.

Lemma 5.12. Let $G = (V, E)$ be a directed acyclic graph with root r. Let $S = (V, T)$ be a depth-first spanning tree for G with root r. Let a, b, c, and d be vertices in V, such that $a \xrightarrow[S]{*} b \xrightarrow[S]{+} c \xrightarrow[S]{*} d$. Let (a, c) and (b, d) be forward edges. Then replacing forward edge (b, d) by forward edge (a, d) does not change the dominators of any vertex in G. (See Fig. 5.28.)

Proof. Let G' be the graph resulting from replacing edge (b, d) in G by the edge (a, d). Assume v is a dominator of w in G but not in G'. Then there

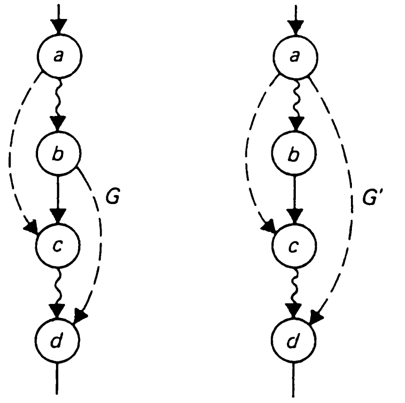


Fig. 5.28 Transformation of Lemma 5.12.

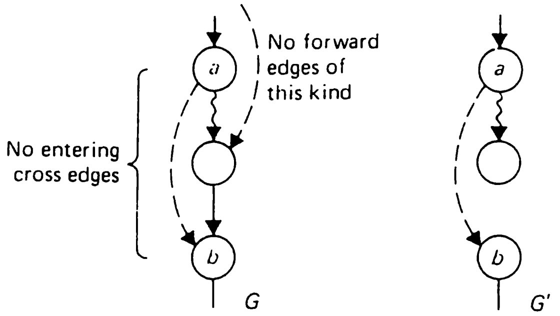


Fig. 5.29 Transformation of Lemma 5.13.

exists a path P from r to w in G' not containing v. Clearly, this path must use the edge (a, d) since it is the only edge in G' not in G. Thus we can write $P: r \overset{*}{\Longrightarrow} a \Longrightarrow d \overset{*}{\Longrightarrow} w$. It follows that v must lie on the path $a \underset{S}{\overset{+}{\longrightarrow}} d$ and that v is distinct from a and d. If $a < v \leq b$ in the depth-first numbering, then replacing the edge $a \Longrightarrow d$ in P by the path $a \Longrightarrow c \underset{S}{\overset{*}{\longrightarrow}} d$ results in a path from r to w in G not containing v. If $b < v < d$, then replacing the edge $a \Longrightarrow d$ in P by the path $a \underset{S}{\overset{*}{\longrightarrow}} b \Longrightarrow d$ results in a path from r to w in G not containing v. Since either $a < v \leq b$ or $b < v < d$, there is a path from r to w in G not containing v, a contradiction.

Assume v is a dominator of w in G' but not in G. Then there exists a path P from r to w in G not containing v. Since this path is not in G', the path P nust contain the edge $b \Longrightarrow d$. It follows that v is on the path $b \overset{*}{\Longrightarrow} d$ and $b < v < d$. Thus there is a path $r \underset{S}{\overset{*}{\longrightarrow}} a \Longrightarrow d \overset{*}{\Longrightarrow} v$ in G' not containing v, a contradiction. □

Lemma 5.13. Let G and S be as in Lemma 5.12. Let $a \underset{S}{\overset{*}{\longrightarrow}} b$ and let (a, b) be a forward edge of G. If there is no forward edge (c, d) with $c < a$ and $a < d \leq b$, and no cross edge (e, d) with $a < d \leq b$, then deleting the tree edge into b does not change the dominators of a vertex. (See Fig. 5.29.)

roof. The proof is similar to that of Lemma 5.12. □

Lemma 5.14. Let G and S be as in Lemma 5.12. Let (c, d) be a cross edge of G and let b be the highest-numbered common ancestor of c and d. Let a be MIN($\{b\} \cup \{v|(v, w)$ is a forward edge and $b < w \leq c\}$). If there is no cross edge entering any vertex on path $b \underset{S}{\overset{*}{\longrightarrow}} c$, excluding vertex b, then replacing cross edge (c, d) by a forward edge (a, d) does not change the dominators of any vertex in G. (See Fig. 5.30.)

roof. Let G' be the graph resulting from the replacement of edge (c, d) in G y the edge (a, d). Assume vertex v is a dominator of vertex w in G but not in G'. Then there exists a path P from r to w in G' not containing v. Clearly, P lust contain the edge (a, d). Thus v must lie on the spanning tree path $a \underset{S}{\overset{*}{\longrightarrow}} b$

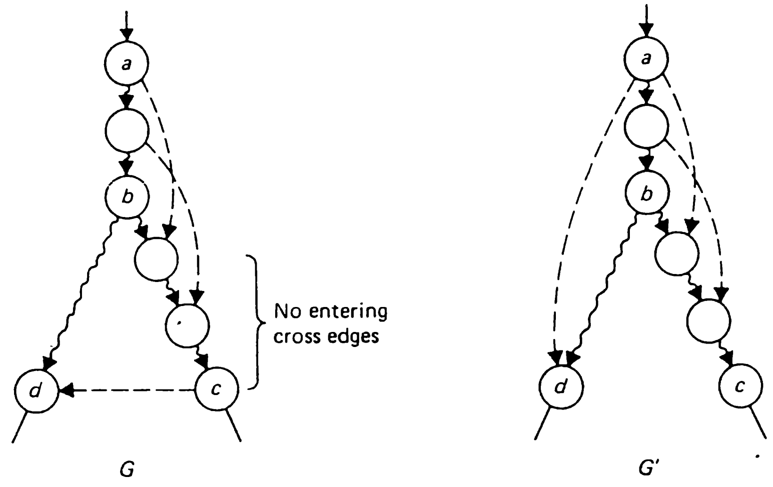


Fig. 5.30 Transformation of Lemma 5.14.

else replacing $a \Longrightarrow d$ in P by either $a \xrightarrow[S]{*} d$ or $a \xrightarrow[S]{*} c \Longrightarrow d$ results in a path from r to w in G not containing v. But then replacing the edge $a \Longrightarrow d$ in P by $a \Longrightarrow u \xrightarrow[S]{*} c \Longrightarrow d$, where u is on the path $b \xrightarrow[S]{*} c$, and $u > b$, results in a path from r to w in G not containing v, a contradiction.

Assume v is a dominator of w in G' but not in G. Then there exists a path P from r to w in G not containing v. Clearly, P contains the edge (c, d). Write P as $r \overset{*}{\Longrightarrow} c \Longrightarrow d \overset{*}{\Longrightarrow} w$. The path $r \overset{*}{\Longrightarrow} c$ must contain some vertex on path $a \xrightarrow[S]{*} b$, since there are no cross edges with heads on $b \xrightarrow[S]{*} c$ (excluding b). Let x be the highest-numbered such vertex. Let P_1 be the portion of the path P from r to x, followed by $x \xrightarrow[S]{*} d$, followed by the portion of P from d to w. Let P_2 be the path $r \xrightarrow[S]{*} a \Longrightarrow d$ followed by the portion of P from d to w. If v is on P_1, then v must be on $x \xrightarrow[S]{*} d$ with $v > x$. If v is on P_2, then v must be on $r \xrightarrow[S]{*} a$. Since $a \leq x$, one of these paths in G' does not contain v, a contradiction. □

It is easily shown that repeated application of the transformations in Lemmas 5.12–5.14, until they can no longer be applied, will transform G into a tree. Since the transformations do not change the dominator relation, the final tree must be the dominator tree for G. This will be our algorithm to compute the dominators of G. The whole trick is to design a data structure that will allow us to find efficiently the appropriate edges to which to apply the transformations of Lemmas 5.12, 5.13, and 5.14.

Intuitively, we construct the dominator tree for the given directed acyclic graph $G = (V, E)$ as follows. First we perform a depth-first search of G starting at the root to construct a depth-first spanning tree $S = (V, T)$. We relabel the vertices of G according to their depth-first numbering. We then apply the transformations in Lemmas 5.12–5.14 to G. The transformations ıre implemented by two interrelated routines, one which handles forward :dges, the other cross edges.

. Forward edges

Ve assume for the time being that G has no cross edges. If a vertex v is the ead of more than one forward edge, then by Lemma 5.12, all forward edges /ith head v, except for the one with lowest tail, can be deleted without chang-ıg the dominator of any vertex.

A *composite edge* is an ordered pair (t, H), where t is a vertex called the *ıil* of the composite edge and H is a set of vertices called the *head* of the ›mposite edge. A composite edge $(t, \{h_1, h_2, \ldots, h_k\})$ represents the set of lges $(t, h_1), (t, h_2), \ldots, (t, h_k)$.

Lemma 5.12 is repeatedly applied to change the tails of various forward lges. Sometimes each edge in a set of edges with common tail t will have its il changed to t'. To do this efficiently, certain sets of forward edges with ·mmon tails are represented by a single composite edge. Initially, each rward edge (t, h) is represented by a composite edge $(t, \{h\})$.

We associate with each vertex v of G a set $F[v]$. $F[v]$ contains pairs of : form $(t, \{h_1, h_2, \ldots, h_m\})$ such that t is an ancestor of v, each h_i is a scendant of v, and (t, h_i) is a forward edge in G. Initially, $F[v] =$, $\{v\})\}$ where t is the lowest-numbered tail of a forward edge with ıd v.

Now, we traverse the spanning tree in reverse preorder. On returning ng spanning tree edge (v, w), we find that the set $F[w]$ contains a compos-edge for each proper ancestor t of w which is currently the tail of a forward ;e of a descendant of w. We then perform the following actions.

Let $(t, \{h_1, h_2, \ldots, h_m\})$ be the composite edge in $F[w]$ with the highest-numbered tail. If $t = v$, then remove this composite edge from $F[w]$. (The composite edge represents a set of forward edges whose tails have been pulled up to v but will not be pulled up further by application of Lemma 5.12.) Delete from G the spanning tree edge with head h_i, for $1 \le i \le m$. (This step corresponds to an application of Lemma 5.13.)
If there is a forward edge (t, v) in G.† then for each composite edge $(u, \{h_1, \ldots, h_m\})$ in $F[w]$ such that $u \ge t$, do the following:
a) Remove $(u, \{h_1, \ldots, h_m\})$ from $F[w]$.
b) Union $\{h_1, \ldots, h_m\}$ with the head of the composite edge in $F[v]$ which represents among other edges, the edge (t, v).
(This step corresponds to an application of Lemma 5.12.)
Replace $F[v]$ by $F[v] \cup F[w]$.

:all that we assume all forward edges with head v, except that with lowest tail, been removed from G.

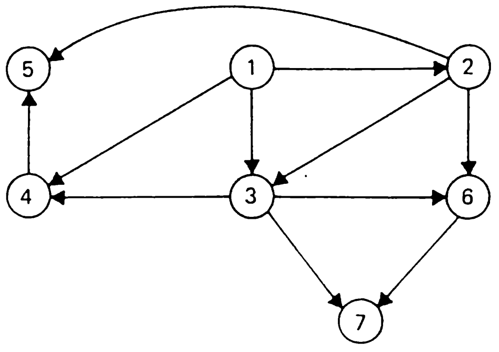


Fig. 5.31 A rooted directed acyclic graph.

Example 5.15. Consider the rooted directed acyclic graph G shown in Fig. 5.31. A depth-first search of G can produce the graph in Fig. 5.32(a). Also shown on the graph are the F-sets associated with each vertex. Figure 5.32(b)–(d) shows the effect of processing forward edges. The final dominator tree is the one in Fig. 5.32(d) □

It is left to you to show that upon reaching the root, the resulting graph is the correct dominator tree (assuming no cross edges). The algorithm can be efficiently implemented by using the disjoint-set union algorithm to manipulate the sets of heads of composite edges. The sets $F[v]$, consisting of composite edges, can be represented by 2–3 trees, since we must be able to delete efficiently a composite edge, to find that composite edge with greatest tail in a set of composite edges, and to form unions of sets of composite edges. With such a data structure the time required to process a graph with e edges is $O(e \log e)$.

II. Cross edges

We cannot in general assume that there are no cross edges. We can, by the method to be described, replace cross edges by forward edges. However, we should not do this replacement before working on forward edges as in part I, since the data structures built during part I help us to apply Lemma 5.14 efficiently. Moreover, we should not execute part I completely before eliminating cross edges, since each cross edge eliminated will become a forward edge. What we should do is add cross-edge handling steps to the reverse preorder traversal described for forward edges. Note that part I requires, because of the use of Lemma 5.13, that there be no cross edges into certain vertices at certain times. The fact that the traversal is done in reverse preorder, together with the steps outlined below, should convince you that each cross edge will have been changed to a forward edge before that time when its existence would make Lemma 5.13 inapplicable.

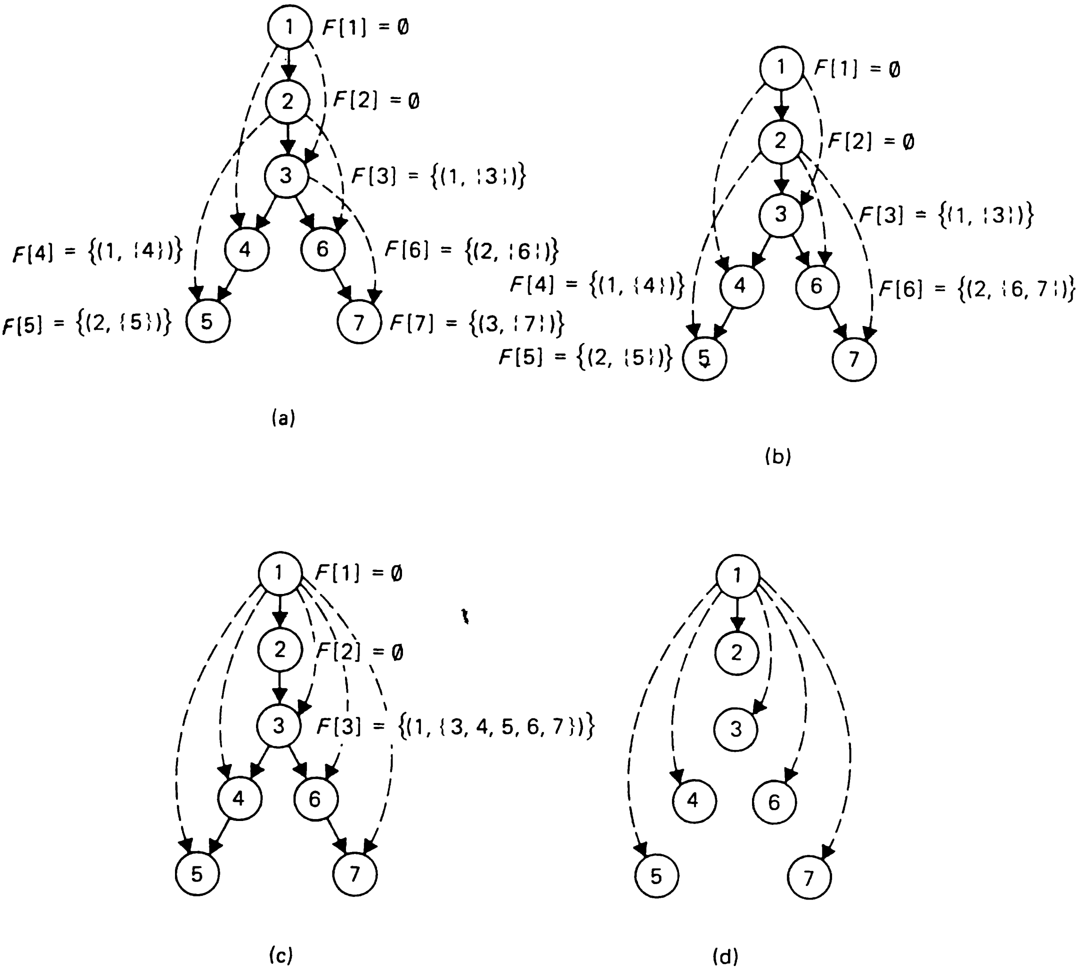


Fig. 5.32 Effect of forward edges transformation: (a) initially; (b) on returning along spanning tree edge (6, 7); (c) on returning along spanning tree edge (3, 4); (d) on returning along spanning tree edge (1, 2).

Let S be the depth-first spanning tree for G. Initially, for each cross edge (v, w) we compute the highest-numbered common ancestor of v and w. To each vertex v we shall attach a set $L[v]$ of ordered pairs (u, w), where (u, w) represents a request for the highest-numbered ancestor of u and w, $u > w$. Initially, $L[v] = \{(v, w) | \text{there is a cross edge } (v, w), v > w\}$. While traversing S as in part I, we perform the following additional actions.

1. On traversing the tree edge (v, w), $v < w$, delete from $L[v]$ each (x, y) such that $y \geq w$. Vertex v is the highest-numbered common ancestor of x and y.
2. On returning to v by spanning tree edge (v, w), replace $L[v]$ by $L[v] \cup L[w]$.

Computing the highest-numbered ancestors can be done in at most $O(eG(e))$ steps, where e is the number of edges in G, by use of a generalization of the off-line MIN algorithm of Exercise 4.21.

The cross edges are handled by converting them to forward edges by Lemma 5.14. The process of converting cross edges into forward edges must be done while the forward edges are being processed. With each vertex v we shall associate a set $C[v]$ of composite edges. Initially, $C[v] = \{(v, \{h_1, \ldots, h_m\}) | (v, h_i)$ is a cross edge for $1 \leq i \leq m\}$. On returning to vertex v by tree edge (v, w), we perform the following steps, in addition to those for handling forward edges.

1. Replace $C[v]$ by $C[v] \cup C[w]$.
2. Delete each cross edge (x, y) such that v is the highest-numbered ancestor of x and y from the composite edge currently representing it. If the composite edge has tail t, replace the cross edge (x, y) by the forward edge (t, y). If there is already a forward edge into y, retain only the forward edge with lower tail.
3. Let (u, v) be the forward edge, if any, with head v. Otherwise, let (u, v) be the tree edge into v. After searching all descendants of v, delete from

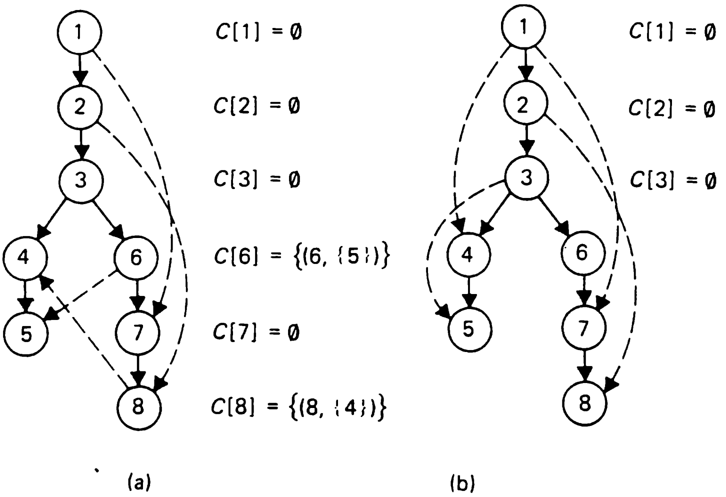


Fig. 5.33 Removing cross edges: (a) initially; (b) after considering edge (3,6).

$C[v]$ any composite cross edge with tail above u. Union the sets of heads of the deleted composite cross edges together and form a new composite edge with tail u. Add the new composite edge to $C[v]$.

Example 5.16. Consider the depth-first spanning tree shown in Fig. 5.33(a). Values for $C[v]$ are shown for selected vertices. Since there is a forward edge from vertex 2 into vertex 8, we change composite edge (8, {4}) in $C[8]$ to (2, {4}). We then union $C[8]$ into $C[7]$. Since (1, 7) is a forward edge, we change composite edge (2, {4}) into (1, {4}). On our return to 6, $C[6]$ becomes {(1, {4}), (6, {5})}.

When we return from vertex 6 to vertex 3, $C(3)$ becomes {(3, {5}), (1, {4})}. Vertex 3 is the highest-numbered ancestor of vertices 6 and 5, and 8 and 4, so we delete the composite edges (3, {5}) and (1, {4}) from $C[3]$ and add the forward edge (3, 5) and (1, 4) to G. The result is shown in Fig. 5.33(b). The remainder of the search produces no further changes. □

The composite cross edges can be represented by 2–3 trees. We leave as an interesting exercise the formal description of the dominator algorithm. If you can combine the appropriate structures you have mastered the techniques of Chapter 4 and 5.

EXERCISES

5.1 Find the minimum-cost spanning tree for the graph of Fig. 5.34 on the assumption that edges shown are undirected.

5.2 Let $S = (V, T)$ be a minimum-cost spanning tree constructed by Algorithm 5.1. Let $c_1 \leq c_2 \leq \cdots \leq c_n$ be the costs of the edges in T. Let S' be an arbitrary spanning tree with edge costs $d_1 \leq d_2 \leq \cdots \leq d_n$. Show that $c_i \leq d_i$ for $1 \leq i \leq n$.

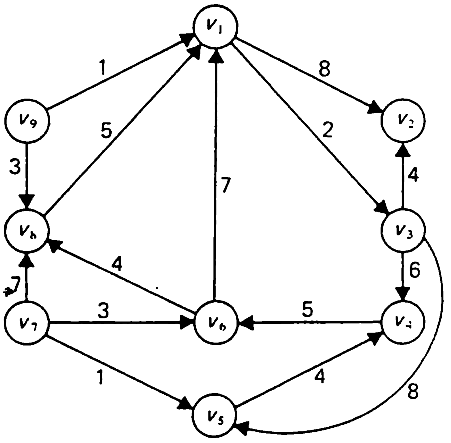


Figure 5.34

****5.3** The following strategy can be used to find a minimum-cost spanning tree for a graph with n vertices and e edges in time $O(e)$, provided $e \geq f(n)$ for some function f which you must find. At various times, vertices will be grouped into sets which are connected by the tree edges found so far. All edges incident upon one or two vertices in the set will be kept in a priority queue for the set. Initially, each vertex is in a set by itself.

The iterative step is to find a smallest set S and the lowest-cost edge leading out of S, say to set T. Then add that edge to the spanning tree and merge sets S and T. However, if all sets are at least of size $g(n)$, where g is another function you must find, then instead make a new graph with one vertex for each set. The vertices in the new graph corresponding to sets S_1 and S_2 are adjacent if some vertex in S_1 was originally adjacent to some vertex in S_2. The cost of the edge joining S_1 and S_2 in the new graph is the minimum cost of any edge originally between any vertex in S_1 and a vertex in S_2. Then apply the algorithm recursively to the new graph.

Your problem is to select $g(n)$ so that $f(n)$ is minimized.

5.4 Find a depth-first spanning forest for the undirected graph of Fig. 5.35. Choose any vertex you like to begin each tree. Find the biconnected components of the graph.

5.5 Find a depth-first spanning forest for the directed graph of Fig. 5.34. Then find the strongly connected components.

5.6 Find the biconnected components of the graph of Fig. 5.36.

5.7 Use depth-first search to help design efficient algorithms to do the following.

a) Divide an undirected graph into its connected components.

b) Find a path in an undirected connected graph which goes through each edge exactly once in each direction.

c) Test whether a directed graph is acyclic.

d) Find an order for the vertices of an acyclic directed graph such that $v < w$ if there is a path from v to w of length greater than zero.

e) Determine whether the edges of a connected, undirected graph can be directed to produce a strongly connected, directed graph. [*Hint:* Show that this can be done if and only if removing any edge from G leaves a connected graph.]

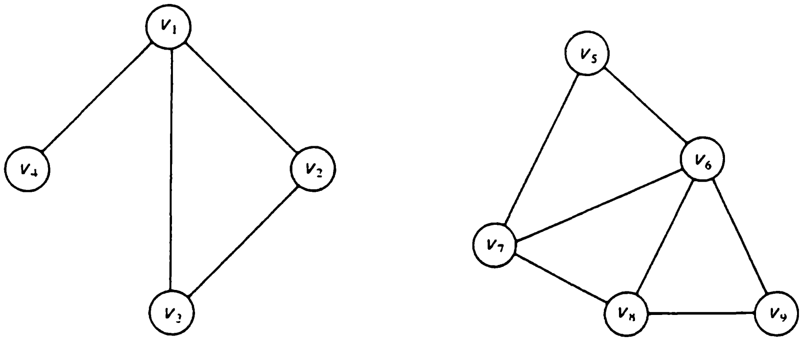


Figure 5.35

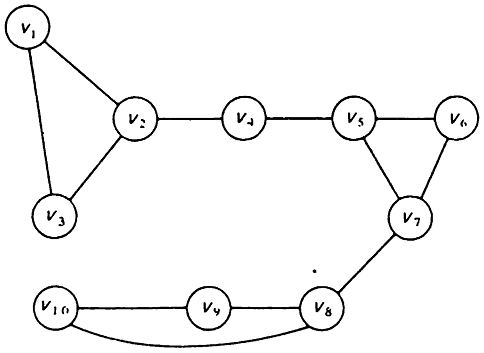


Figure 5.36

5.8 Let $G = (V, E)$ be a *multigraph* (i.e., an undirected graph G which may have more than one edge between a pair of vertices). Write an $O(\|E\|)$ algorithm to delete each degree 2 vertex v [by replacing edges (u, v) and (v, w) by an edge (u, w)] and eliminate multiple copies of edges by replacing them with a single edge. Note that replacing multiple copies of an edge by a single edge may create a degree 2 vertex which must then be removed. Similarly, removing a degree 2 vertex may create a multiple edge which must then be removed.

5.9 An *Euler circuit* for an undirected graph is a path which starts and ends at the same vertex and uses each edge exactly once. A connected, undirected graph G has an Euler circuit if and only if every vertex is of even degree. Give an $O(e)$ algorithm to find an Euler circuit in a graph with e edges provided one exists.

*5.10 Let $G = (V, E)$ be a biconnected undirected graph. Let (v, w) be an edge of G. Let $G' = (\{v, w\}, \{(v, w)\})$. Find a technique for executing on-line a sequence of instructions of the form FINDPATH(s, t), where s is a vertex of G' and (s, t) is an edge of G not in G'. FINDPATH(s, t) is executed by finding a simple path in G starting with edge (s, t) and ending at a vertex in G' other than s and adding the vertices and edges of the path to G'. The execution time of any sequence should be $O(\|E\|)$.

5.11 Consider the directed graph G of Fig. 5.37.

a) Find the transitive closure of G.

b) Find the length of the shortest path between each pair of vertices of G. The cost of each edge is shown in Fig. 5.37.

*5.12 Find a closed semiring with four elements.

*5.13 A *transitive reduction* of a directed graph $G = (V, E)$ is defined to be any graph $G' = (V, E')$ with as few edges as possible, such that the transitive closure of G' is equal to the transitive closure of G. Show that if G is acyclic, then the transitive reduction of G is unique.

**5.14 Show that the time $R(n)$ to compute the transitive reduction of an n-vertex acyclic graph is within a constant factor of the time $T(n)$ to compute transitive closures, on the (reasonable) assumption that $8R(n) \geq R(2n) \geq 4R(n)$ and $8T(n) \geq T(2n) \geq 4T(n)$.

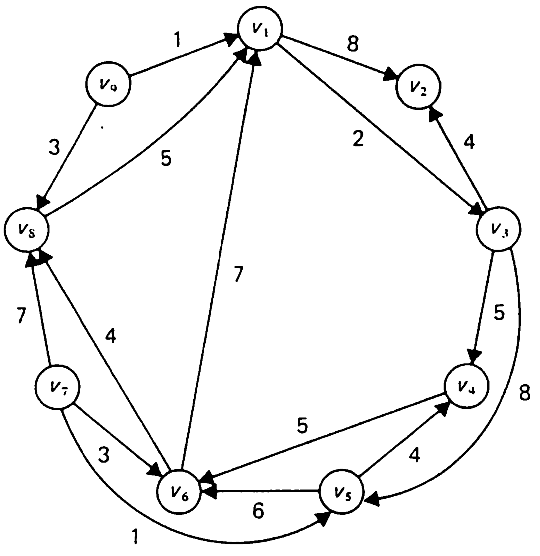


Figure 5.37

***5.15** Show that the time necessary to compute the transitive reduction of an acyclic graph is within a constant factor of the time necessary to find the transitive reduction of an arbitrary graph, on the assumption of Exercise 5.14.

***5.16** Prove Exercise 5.15 for transitive closure.

***5.17** Let A be an $n \times n$ matrix over the closed semiring $\{0, 1\}$. Without using a graph interpretation, prove the following.

a) $A^* = I_n + A + A^2 + \cdots + A^n$

b) $\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^* = \begin{pmatrix} A^* & A^*BC^* \\ 0 & C^* \end{pmatrix}$

[*Hint:* Show that

$$\begin{pmatrix} A & B \\ 0 & C \end{pmatrix}^i = \begin{pmatrix} A^i & A^{i-1}B + A^{i-2}BC + \cdots + BC^{i-1} \\ 0 & C^i \end{pmatrix}.]$$

***5.18** Show that the shortest-path algorithm of Section 5.8 still works if some of the edges have negative cost but no cycle has negative cost. What happens if there are cycles of negative cost?

****5.19** Show that the positive and negative reals with $+\infty$ and $-\infty$ is not a closed semiring. How do you explain Exercise 5.18 in this light? [*Hint:* What properties of a closed semiring are actually used in Algorithm 5.5?]

5.20 Use Algorithm 5.6 to find the shortest distance from vertex v_8 to each vertex v in the graph G of Fig. 5.37.

***5.21** Show that the single-source shortest-path problem for nonnegative edges can be solved in time $O(e \log n)$ for a graph with e edges and n vertices. [*Hint:* Use the proper data structure so that lines 5 and 8 of Algorithm 5.6 can be done efficiently when $e << n^2$.]

```
begin
    $S \leftarrow \{v_0\}$;
    $D[v_0] \leftarrow 0$;
    for each $v$ in $V - \{v_0\}$ do $D[v] \leftarrow l(v_0, v)$;
    while $S \neq V$ do
        begin
            choose a vertex $w$ in $V - S$ such that $D[w]$ is a minimum;
            $S \leftarrow S \cup \{w\}$;
            for $v \in V$ such that $D[v] > D[w] + l(w, v)$ do
                begin
                    $D[v] = D[w] + l(w, v)$;
                    $S \leftarrow S - \{v\}$
                end
        end
end
```

Fig. 5.38. A single-source shortest-path algorithm.

***5.22** Show that the single-source shortest-path problem for nonnegative edges can be solved in time $O(ke + kn^{1+1/k})$ for any fixed constant k, on a graph with e edges and n vertices.

***5.23** Prove that the single-source shortest path algorithm of Fig. 5.38 computes the shortest path from v_0 to each v in an arbitrary graph with negative cost edges but no negative cost cycles.

***5.24** What is the order of the execution time of the algorithm of Fig. 5.38? [*Hint:* Execute the algorithm on a five-vertex graph with the edge costs shown in Fig. 5.39.]

5.25 Give an algorithm to determine whether a directed graph with positive and negative cost edges has a negative cost cycle.

5.26 Change the selection rule for w in the algorithm of Fig. 5.38 so as to guarantee for arbitrary cost edges that the time bound is $O(n^3)$.

5.27 Write an algorithm which given an $n \times n$ matrix M of positive integers will find a sequence of adjacent entries starting from $M[n, 1]$ and ending at $M[1, n]$ such that the sum of the absolute values of differences between adjacent entries is minimized. Two entries $M[i, j]$ and $M[k, l]$ are *adjacent* if (a) $i = k \pm 1$ and $j = l$, or (b) $i = k$

	1	2	3	4	5
1	0	7	8	9	10
2	0	0	8	9	10
3	0	−2	0	9	10
4	0	−4	−3	0	10
5	0	−7	−6	−5	0

Fig. 5.39. Edge cost for a five-vertex graph.

1	9	6	12
8	7	3	5
5	9	11	4
7	3	2	6

Fig. 5.40 Matrix of positive integers.

and $j = l \pm 1$. For example, in Fig. 5.40, the sequence 7, 5, 8, 7, 9, 6, 12 is a solution.

5.28 The algorithm of Fig. 5.24 (p. 208) computes for each $v \in V$ the minimum over all paths P from v_0 to v of the cost of P. Modify the algorithm to produce for each v in V a path of minimum cost.

**5.29 Write a program to implement the dominator algorithm in Section 5.11.

Research Problems

5.30 There are numerous graph problems in which depth-first search might be of some utility. One that stands out concerns *k-connectedness*. An undirected graph is *k-connected* if, for every pair of vertices v and w, there are k paths between v and w such that no vertex (except v and w) appears on more than one path. Thus biconnected means 2-connected. Hopcroft and Tarjan [1973b] have given a linear time algorithm to find 3-connected components. It is natural to conjecture that there exist linear (in numbers of vertices and edges) time algorithms to find k-connected components for each k. Can you find one?

5.31 Another interesting prospect, which may or may not involve depth-first search, is to find a linear (in number of edges) algorithm for finding minimum-cost spanning trees.

5.32 A third problem worth considering is the single-source shortest-path problem when $e << n^2$. Does there exist an $O(e)$ algorithm to find even the shortest distance between two particular points? The reader should be aware of Exercises 5.21 and 5.22 from Johnson [1973], and also of Spira and Pan [1973], which shows that $n^2/4$ comparisons are necessary in general for those algorithms which use only comparisons between sums of edge costs. Wagner [1974] has used a bucket sort technique to obtain an $O(e)$ algorithm in the case where the edge weights are small integers.

5.33 The problem of finding shortest paths between all pairs of points has been shown to require kn^3 steps for some constant $k > 0$ (Kerr, [1972]) provided that the only permissible operations are MIN and +. M. O. Rabin has strengthened this result to $n^3/6$. However, it is possible that we can do better than $O(n^3)$ if we permit other operations. For example, transitive closure (equivalently, Boolean matrix multiplication) can be done in less than $O(n^3)$ steps if we use operations other than the Boolean ones, as we shall see in the next chapter.

BIBLIOGRAPHIC NOTES

Two sources on graph theory are Berge [1958] and Harary [1969]. Algorithm 5.1 on minimum-cost spanning trees is from Kruskal [1956]. Prim [1957] provides another approach to the problem. The algorithm proposed in Exercise 5.3 was pointed out to us by P. M. Spira.

The biconnectedness algorithm using depth-first search is due to J. E. Hopcroft. The strongly connected components algorithm is from Tarjan [1972]. Numerous applications of depth-first search to produce optimal or best known algorithms appear in the literature. Hopcroft and Tarjan [1973a] gives a linear planarity test. Hopcroft and Tarjan [1973b] describes a linear algorithm for triply connected components. Tarjan [1973a] uses the concept to produce the best algorithm known for finding dominators and Tarjan [1973b] gives a test for "flow graph reducibility."

Algorithm 5.5, the general path-finding algorithm, is essentially due to Kleene [1956], who used it in connection with "regular expressions" (see Section 9.1). The form of the algorithm given here is from McNaughton and Yamada [1960]. The $O(n^3)$ transitive closure algorithm is due to Warshall [1962]. The analogous shortest-path algorithm for all pairs of points is from Floyd [1962]. (See also Hu [1968]). The single-source algorithm is by Dijkstra [1959]. Spira and Pan [1973] show that Dijkstra's algorithm is essentially optimal under the decision tree model.

In Dantzig, Blattner, and Rao [1967] it was observed that the presence of negative edges does not affect the all-points shortest-path problem if no negative cycles are present. Johnson [1973] discusses the single-source problem with negative edges, and the solutions to Exercises 5.21 and 5.22 can be found there. Spira [1973] gives an $O(n^2 \log^2 n)$ expected time algorithm for finding shortest paths.

The relation between path problems and matrix multiplication is due to Munro [1971] and Furman [1970] (Theorem 5.7), and Fischer and Meyer [1971] (Theorem 5.6). The relation with transitive reduction (Exercises 5.13–5.15) is from Aho, Garey, and Ullman [1972]. Even [1973] discusses the k-connectivity of graphs.

MATRIX MULTIPLICATION AND RELATED OPERATIONS

CHAPTER 6

In this chapter we investigate the asymptotic computational complexity of matrix multiplication with elements chosen from an arbitrary ring. We shall see that the $O(n^3)$ "ordinary" matrix multiplication algorithm can be asymptotically improved; $O(n^{2.81})$ time is sufficient to multiply two $n \times n$ matrices. Moreover, we shall see that other operations such as LUP decomposition, matrix inversion, and evaluation of a determinant are reducible to matrix multiplication and thus can be performed as fast as matrix multiplication. We shall further show that matrix multiplication is reducible to matrix inversion and thus any improvement in the asymptotic time for one problem would result in a similar improvement for the other. We conclude the chapter with two algorithms for Boolean matrix multiplication whose asymptotic time complexities are less than $O(n^3)$.

To a large extent the algorithms in this chapter are not to be regarded as being practical with current computer hardware. For one thing, because of the hidden constant factors, it is only for quite large values of n that the asymptotically faster algorithms actually outperform the usual $O(n^3)$ algorithms. Furthermore, the problem of numerical error control for this family of algorithms is not sufficiently well understood. Nevertheless, we feel that the ideas of this chapter are worth considering because they remind us that the obvious algorithms are not always best, and they hopefully lay the groundwork for the development of even more efficient and genuinely practical algorithms for this important family of problems.

6.1 BASICS

This section presents basic definitions of some algebraic concepts that are useful in dealing with matrix multiplication problems. The reader familiar with rings and linear algebra should proceed directly to Section 6.2.

Definition. A *ring* is an algebraic structure $(S, +, \cdot, 0, 1)$ in which S is a set of elements, and $+$ and $\cdot$ are binary operations on S. For each a, b, and c in S, the following properties hold.

1. $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b + c)$ ($+$ and $\cdot$ are associative).
2. $a + b = b + a$ ($+$ is commutative).
3. $(a + b) \cdot c = a \cdot c + b \cdot c$ and $a \cdot (b + c) = a \cdot b + a \cdot c$ ($\cdot$ distributes over $+$).
4. $a + 0 = 0 + a = a$ (0 is the $+$ identity).
5. $a \cdot 1 = 1 \cdot a = a$ (1 is the $\cdot$ identity).
6. For each a in S there is an *inverse* $-a$ such that $a + (-a) = (-a) + a = 0$.

Note that the last property, the existence of an additive inverse, is not necessarily applicable to every closed semiring (see Section 5.6). Also, the fourth property for a closed semiring, namely that infinite sums exist and are unique, is not always applicable to a ring. If $\cdot$ is commutative, we say the ring is *commutative*.

If, in a commutative ring, for each element a there is a multiplicative inverse a^{-1} such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$, then the ring is called a *field.*

Example 6.1. The real numbers form a ring with + and $\cdot$ standing for ordinary addition and multiplication. The reals do not, however, form a closed semiring.

The system $(\{0, 1\}, +, \cdot, 0, 1)$, where + is addition modulo 2 and $\cdot$ is ordinary multiplication, forms a ring but not a closed semiring since $1 + 1 + \cdots$ is not well defined. However, if + is redefined so that $a + b$ is 0 if both a and b are 0, and $a + b$ is 1 otherwise, then we have the closed semiring S_2 of Example 5.1. S_2 is not a ring, since 1 has no inverse. □

An important class of rings formed from matrices is introduced in the following definition and lemma.

Definition. Let $R = (S, +, \cdot, 0, 1)$ be a ring, and let M_n be the set of $n \times n$ matrices whose elements are chosen from R. Let 0_n be the $n \times n$ matrix of 0's and let I_n be the $n \times n$ *identity matrix* with 1's on the main diagonal and 0's elsewhere. For A and B in M_n let $A +_n B$ be the $n \times n$ matrix C, where $C[i,j] = A[i,j] + B[i,j]$,† and let $A \cdot_n B$ be the $n \times n$ matrix D, where $D[i,j] = \Sigma_{k=1}^{n} A[i,k] \cdot B[k,j]$.

Lemma 6.1. $(M_n, +_n, \cdot_n, 0_n, I_n)$ is a ring.

Proof. Elementary exercise. □

Note that $\cdot_n$, the matrix multiplication operation as defined above, is not commutative for $n > 1$, even if $\cdot$, the multiplicative operation in the underlying ring R, is commutative. We shall use + and $\cdot$ instead of $+_n$ and $\cdot_n$ when there is no possibility of confusion with the addition and multiplication operators in the underlying ring R. Also, we shall often omit the multiplication operator when its presence should be apparent.

Let R be a ring and let M_n be the ring of $n \times n$ matrices with elements from R. Let us assume n is even. An $n \times n$ matrix in M_n can be partitioned into four $(n/2) \times (n/2)$ matrices. Let $R_{2,n/2}$ be the ring of all 2×2 matrices with elements from $M_{n/2}$. It is straightforward to verify that multiplication and addition of $n \times n$ matrices in M_n is equivalent to multiplication and addition, respectively, of the equivalent 2×2 matrices in $R_{2,n/2}$ whose elements are $(n/2) \times (n/2)$ matrices.

Lemma 6.2. Let f be the mapping from M_n to $R_{2,n/2}$ such that $f(A)$ is the matrix

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

† We use $M[i,j]$ to denote the element in the ith row and jth column of matrix M.

in $R_{2,n/2}$, where A_1 is the upper left quadrant of A, A_2 the upper right, A_3 the lower left, and A_4 the lower right. Then

$$\text{i) } f(A + B) = f(A) + f(B),$$
$$\text{ii) } f(AB) = f(A) \cdot f(B).$$

Proof. It is an elementary exercise to substitute the definitions of + and · for $M_{n/2}$ into the definitions of + and · for $R_{2,n/2}$. □

The importance of Lemma 6.2 is that we can construct an algorithm for $n \times n$ matrix multiplication out of algorithms for 2×2 and $(n/2) \times (n/2)$ matrix multiplication. We shall make use of this fact in the next section tc develop an asymptotically fast matrix-multiplication algorithm. For the moment we stress the fact that the algorithm for 2×2 matrix multiplication is not an arbitrary algorithm but one specifically designed for $R_{2,n/2}$. Since $R_{2,n/2}$ is not necessarily commutative even though R is, the 2×2 matrix multiplication algorithm cannot assume commutativity of the $(n/2) \times (n/2)$ multiplicative operation. It can, of course, make use of any ring property.

Definition. Let A be an $n \times n$ matrix with elements chosen from some field. The *inverse* of A, denoted A^{-1}, is that $n \times n$ matrix, if it exists, such that $AA^{-1} = I_n$.

It is easy to show that if A^{-1} exists, it is unique and that $AA^{-1} = A^{-1}A = I_n$. Furthermore, $(AB)^{-1} = B^{-1}A^{-1}$.

Definition. Let A be an $n \times n$ matrix. The *determinant* of A, denoted **det**(A), is the sum over all permutations $p = (i_1, i_2, \ldots, i_n)$ of the integers 1 through n of the product

$$(-1)^{k_p} \prod_{j=1}^{n} A[j, i_j],$$

where k_p is 0 if p is *even* [constructible from $(1, 2, \ldots, n)$ by an even number of interchanges] and k_p is 1 if p is *odd* (constructible by an odd number of interchanges).

It is easily shown that a permutation is even or odd but not both.

Example 6.2. Let

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

The six permutations of 1, 2, 3 are (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). Surely the permutation (1, 2, 3) is even, since it is constructed by 0 interchanges from itself. The permutation (2, 3, 1) is even, since we can begin with (1, 2, 3), interchange 2 and 3 to get (1, 3, 2), then interchange 1 and 2 to get (2, 3, 1). Similarly, (3, 1, 2) is even and the remaining three permutations are odd. Thus $\mathbf{det}(A) = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$. □

Let A be an $n \times n$ matrix with elements chosen from some field. We can show that A^{-1} exists if and only if $\mathbf{det}(A) \neq 0$ and that $\mathbf{det}(AB) = \mathbf{det}(A)\,\mathbf{det}(B)$. In the case that $\mathbf{det}(A) \neq 0$ we say that the matrix A is *nonsingular.*

Definition. An $m \times n$ matrix A is *upper triangular* if $A[i, j] = 0$ whenever $1 \leq j < i \leq m$. An $m \times n$ matrix A is *lower triangular* if $A[i, j] = 0$ whenever $1 \leq i < j \leq n$.

The product of two upper triangular matrices is an upper triangular matrix. The analogous result is true for lower triangular matrices.

Lemma 6.3. If A is a square upper or lower triangular matrix, then

a) $\mathbf{det}(A)$ is the product of the elements on the main diagonal (i.e., $\Pi_i A[i, i]$), and

b) A is nonsingular if and only if no element on the main diagonal is zero.

Proof. (a) Every permutation $(i_1, i_2, \ldots, i_n)$ except $(1, 2, \ldots, n)$ has some component i_j such that $i_j < j$ and another component i_k such that $i_k > k$. Thus every term in $\mathbf{det}(A)$ except the one for the permutation $(1, 2, \ldots, n)$ is zero. b) Follows immediately from (a). □

Definition. A *unit* matrix is a matrix with 1's along the main diagonal (off-diagonal elements are arbitrary).

Observe that the determinant of a unit upper triangular matrix or unit lower triangular matrix is unity.

Definition. A *permutation matrix* is a matrix of 0's and 1's such that each row and each column has exactly one 1.

Definition. A *submatrix* of a matrix A is a matrix obtained by deleting some rows and columns of A. A *principal submatrix* of an $n \times n$ matrix A is a square submatrix of A that consists of the first k rows of the first k columns of A, $1 \leq k \leq n$.

The *rank* of A, denoted $\mathbf{rank}(A)$, is the size of the largest square nonsingular submatrix of A. For example, the rank of an $n \times n$ permutation matrix is n.

We observe that if $A = BC$, $\mathbf{rank}(A)$ is less than or equal to $\text{MIN}(\mathbf{rank}(B), \mathbf{rank}(C))$. Also, if A has m rows and is of rank m, then any k rows of A form a matrix of rank k.

Definition. The *transpose* of a matrix A, denoted A^T, is the matrix formed by exchanging $A[i, j]$ and $A[j, i]$ for each i and j.

6.2 STRASSEN'S MATRIX-MULTIPLICATION ALGORITHM

Let A and B be two $n \times n$ matrices, where n is a power of 2. By Lemma 6.2, we can partition each of A and B into four $(n/2) \times (n/2)$ matrices and express the product of A and B in terms of these $(n/2) \times (n/2)$ matrices as:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned} \tag{6.1}$$

If we treat A and B as 2×2 matrices, each of whose elements are $(n/2) \times (n/2)$ matrices, then the product of A and B can be expressed in terms of sums and products of $(n/2) \times (n/2)$ matrices as given in (6.1). Suppose we can compute the C_{ij}'s using m multiplications and a additions (or subtractions) of the $(n/2) \times (n/2)$ matrices. By applying the algorithm recursively, we can compute the product of two $n \times n$ matrices in time $T(n)$, where for n a power of 2, $T(n)$ satisfies

$$T(n) \le mT\left(\frac{n}{2}\right) + \frac{an^2}{4}, \qquad n > 2. \tag{6.2}$$

The first term on the right of (6.2) is the cost of multiplying m pairs of $(n/2) \times (n/2)$ matrices, and the second is the cost of doing a additions, assuming $n^2/4$ time is required for each addition or subtraction of two $(n/2) \times (n/2)$ matrices. By an analysis similar to that of Theorem 2.1 with $c = 2$, we can show that as long as $m > 4$, the solution to (6.2) is bounded from above by $kn^{\log m}$ for some constant k. The form of the solution is independent of a, the number of additions. Thus if $m < 8$ we have a method asymptotically superior to the usual $O(n^3)$ method.

V. Strassen discovered a clever method of multiplying two 2×2 matrices with elements from an arbitrary ring using only seven multiplications. By using the method recursively, he was able to multiply two $n \times n$ matrices in time $O(n^{\log 7})$, which is of order approximately $n^{2.81}$.

Lemma 6.4. The product of two 2×2 matrices with elements chosen from an arbitrary ring can be computed with 7 multiplications and 18 additions/subtractions.

Proof. To compute the matrix product

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

first compute the following products.

$$\begin{aligned}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}),\\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}),\\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}),\\
m_4 &= (a_{11} + a_{12})b_{22},\\
m_5 &= a_{11}(b_{12} - b_{22}),\\
m_6 &= a_{22}(b_{21} - b_{11}),\\
m_7 &= (a_{21} + a_{22})b_{11}.
\end{aligned}$$

Then compute the c_{ij}'s, using the formulas

$$\begin{aligned}
c_{11} &= m_1 + m_2 - m_4 + m_6,\\
c_{12} &= m_4 + m_5,\\
c_{21} &= m_6 + m_7,\\
c_{22} &= m_2 - m_3 + m_5 - m_7.
\end{aligned}$$

The count of operations is straightforward. The proof that the desired c_{ij}'s are obtained is a simple algebraic exercise using the laws of a ring. □

Exercise 6.5 gives a technique for computing the product of two 2 × 2 matrices with 7 multiplications and 15 additions.

Theorem 6.1. Two $n \times n$ matrices with elements from an arbitrary ring can be multiplied in $O(n^{\log 7})$ arithmetic operations.

Proof. First consider the case in which $n = 2^k$. Let $T(n)$ be the number of arithmetic operations needed to multiply two $n \times n$ matrices. By application of Lemma 6.4,

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \qquad \text{for} \quad n \geq 2.$$

Thus $T(n)$ is $O(7^{\log n})$ or $O(n^{\log 7})$ by an easy modification of Theorem 2.1.

If n is not a power of 2, then we embed each matrix in a matrix whose dimension is the next-higher power of 2. This at most doubles the dimension and thus increases the constant by at most a factor of 7. Thus $T(n)$ is $O(n^{\log 7})$ for all $n \geq 1$. □

Theorem 6.1 is concerned only with the functional growth rate of $T(n)$. But to know for what value of n Strassen's algorithm outperforms the usual method we must determine the constant of proportionality. However, for n not a power of 2, simply embedding each matrix in a matrix whose dimension s the next-higher power of 2 will give too large a constant. Instead, we can mbed each matrix in a matrix of dimension $2^p r$ for some small value of r, nd use p applications of Lemma 6.4, multiplying the $r \times r$ matrices by the onventional $O(r^3)$ method. Alternatively, we could write a more general ecursive algorithm which, when n is even, would split each matrix into four ubmatrices as before and which, when n is odd, would first augment the dimension of the matrices by 1.

We should point out that determining the constant of proportionality onl bounds the number of arithmetic operations. To compare Strassen's metho with the ordinary method for matrix multiplication we must also consider th additional complexity of the functions needed to access a matrix elemen

6.3 INVERSION OF MATRICES

In this section we shall show that the problem of inverting matrices in a certain class can be reduced to the problem of matrix multiplication. The clas includes all triangular matrices which are invertible but does not include al invertible matrices. Later, we extend the result to arbitrary invertible matrices In this section and Sections 6.4 and 6.5 we assume all matrices have element chosen from some field.

Lemma 6.5. Let A be partitioned as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.$$

Suppose A_{11}^{-1} exists. Define $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12}$ and assume Δ^{-1} exists Then

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}\,\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\,\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{bmatrix}. \qquad (6.3$$

Proof. By straightforward algebraic manipulation one can show that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & \Delta \end{bmatrix} \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix},$$

where $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12}$. Thus

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & \Delta^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12}\,\Delta^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}\,\Delta^{-1} \\ -\Delta^{-1}A_{21}A_{11}^{-1} & \Delta^{-1} \end{bmatrix}. \;\square$$

Lemma 6.5 does not apply to all nonsingular matrices. For example, th $n \times n$ permutation matrix A with $A[i, j]$ equal to 1 if $j = n - i + 1$ and 0 otherwise is nonsingular, but $\det(A_{11}) = 0$ for each principal submatrix A_{11}. However, Lemma 6.5 does apply to all nonsingular lower or upper triangula matrices.

Lemma 6.6. If A is a nonsingular upper (lower) triangular matrix, the the matrices A_{11} and Δ of Lemma 6.5 have inverses and are nonsingula upper (lower) triangular.

Proof. Assume that A is upper triangular. The proof for the lower triangula case is analogous. Clearly, A_{11} is nonsingular upper triangular, hence A_{11}^{-1} exists. Next observe that $A_{21} = 0$. Thus $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12} = A_{22}$ and is nonsingular upper triangular. □

Theorem 6.2. Let $M(n)$ be the time required to multiply two $n \times n$ matrices over a ring. If for all m, $8M(m) \geq M(2m) \geq 4M(m)$, then there exists a constant c such that the inverse of any $n \times n$ nonsingular upper (lower) triangular matrix A can be computed in time $cM(n)$.

Proof. We prove the result for n a power of 2. Clearly, if n is not a power of 2, we can embed A in a matrix of the form

$$\begin{bmatrix} A & 0 \\ 0 & I_m \end{bmatrix},$$

where $m + n \leq 2n$ is a power of 2. Thus by increasing the constant c by at most a factor of 8, the result is established for arbitrary n.†

Assuming n is a power of 2, we can partition A into four $(n/2) \times (n/2)$ submatrices and apply (6.3) recursively. Note $A_{21} = 0$, so $\Delta = A_{22}$. Thus we require $2T(n/2)$ time for the inverses of triangular matrices A_{11} and Δ, $2M(n/2)$ time for the two nontrivial multiplications, and $n^2/4$ time for the negation on the upper right. From the theorem hypothesis and the observation $M(1) \geq 1$, we have $n^2/4 \leq M(n/2)$. Thus

$$\begin{aligned} T(1) &= 1 \\ T(n) &\leq 2T\left(\frac{n}{2}\right) + 3M\left(\frac{n}{2}\right), \qquad \text{for } n \geq 2. \end{aligned} \tag{6.4}$$

It is an elementary exercise to show that (6.4) implies $T(n) \leq 3M(n)/2$. □

6.4 LUP DECOMPOSITION OF MATRICES

An efficient method for solving simultaneous linear equations is to make use of what is known as LUP decomposition.

Definition. The *LU decomposition* of an $m \times n$ matrix A, $m \leq n$, is a pair of matrices L and U such that $A = LU$, L is $m \times m$ unit lower triangular, and U is $m \times n$ upper triangular.

We can solve the equation $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, where A is an $n \times n$ matrix, $\mathbf{x}$ an n-dimensional column vector of unknowns, and $\mathbf{b}$ an n-dimensional column vector, by factoring A into the product of a unit lower triangular matrix L and an upper triangular matrix U, provided such factors exist. Then we can write $A\mathbf{x} = \mathbf{b}$ as $LU\mathbf{x} = \mathbf{b}$. To obtain the solution $\mathbf{x}$, we first solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ and then solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

The difficulty with this method is that A may not have an LU decomposition even if A is nonsingular. For example, the matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

† Again, more careful analysis produces a constant c for arbitrary n that is not much different from the best constant known for the case where n is a power of 2.

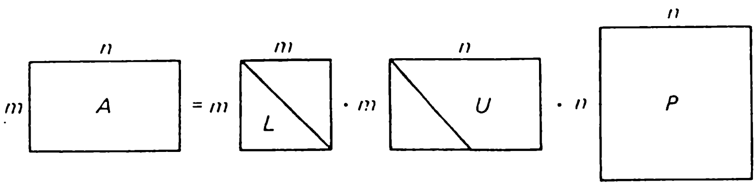
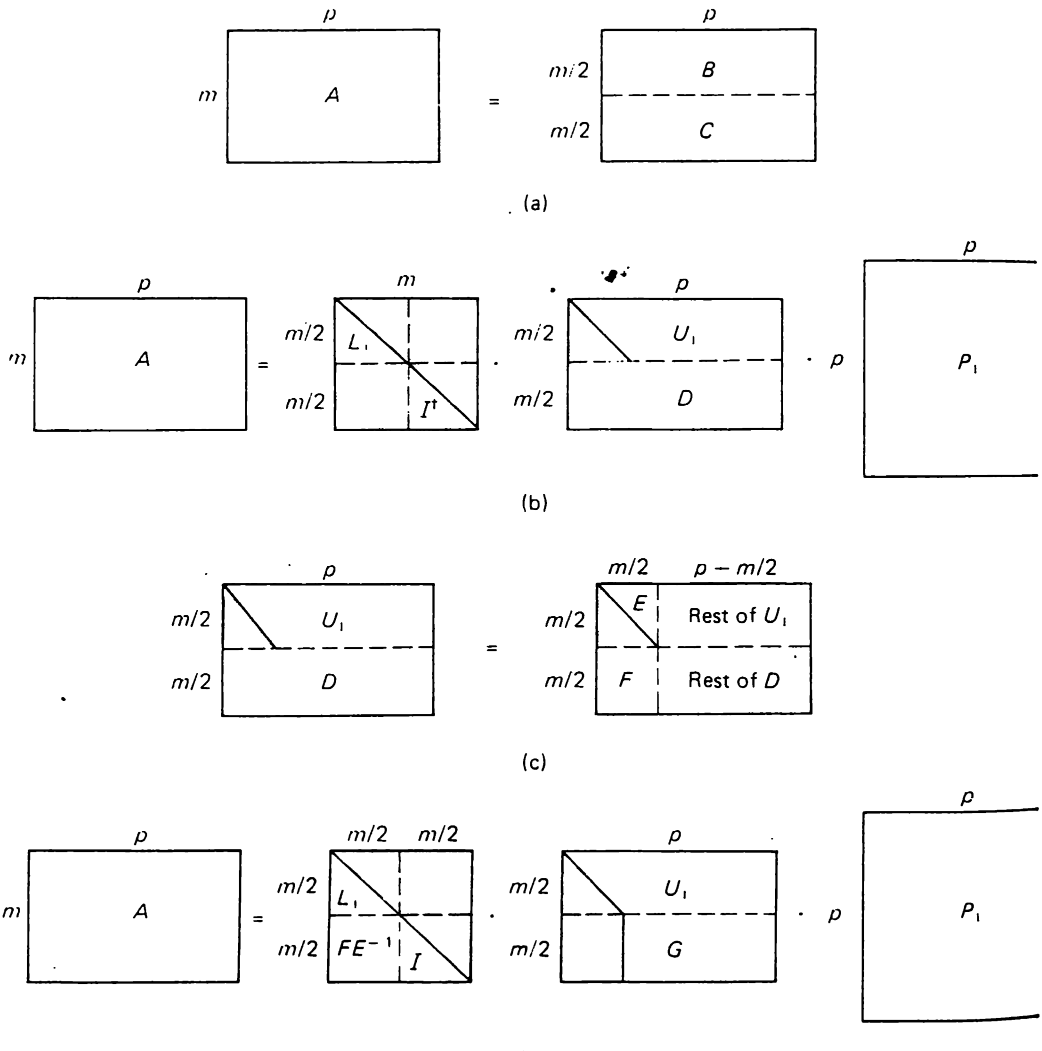


Fig. 6.1 Desired output of FACTOR.



(a)

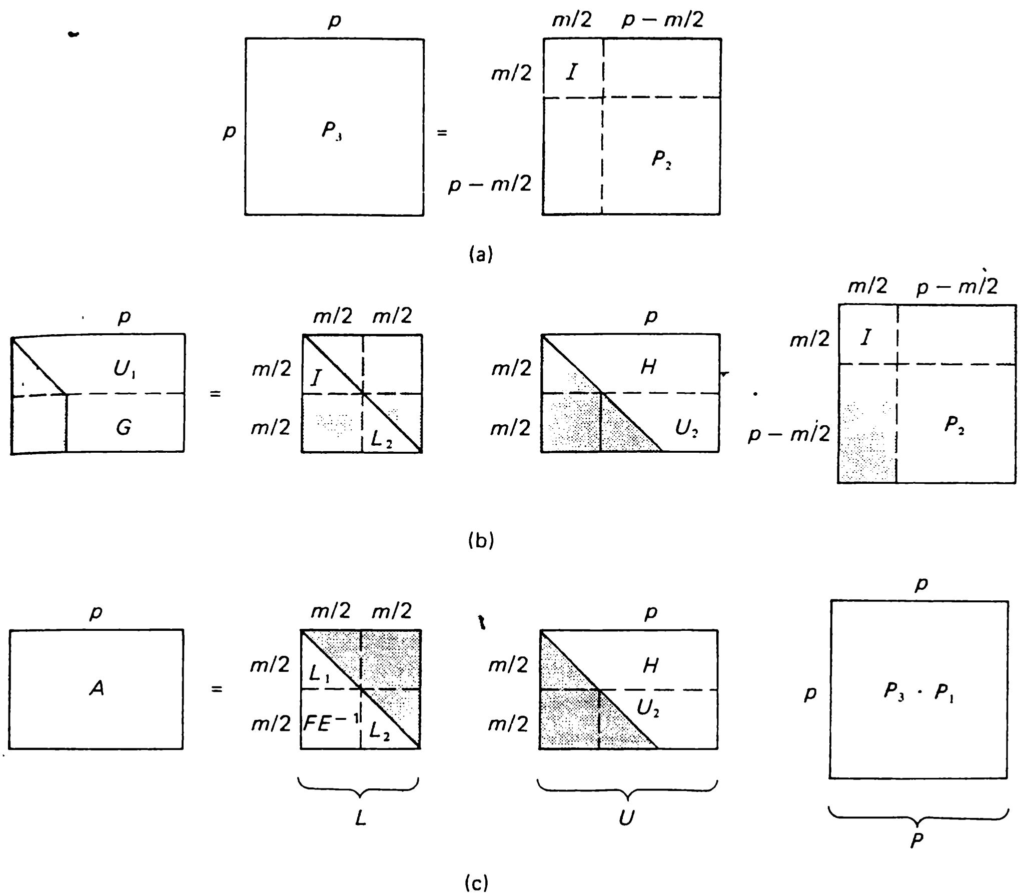
(b)

(c)

(d)

Fig. 6.2 Steps in the procedure FACTOR: (a) initial partition of A; (b) factorization after first call to FACTOR; (c) partition of U_1 and D; (d) zeroing of the lower left c of D.

[†] Note that I can be regarded as a unit lower (or upper) triangular matrix.



6.3 Completion of FACTOR: (a) construction of P_3; (b) decomposition of U_1 and G; omposition of A.

s nonsingular but has no LU decomposition. However, if A is nonsingular, here exists a permutation matrix P such that AP^{-1} has an LU decomposition. Ve now give an algorithm to find, for any nonsingular matrix A, factors L, J, and P such that $A = LUP$. The matrices L, U, and P are called an *LUP lecomposition* of A.

lgorithm 6.1. LUP decomposition.

nput. A nonsingular $n \times n$ matrix M, where n is a power of 2.

)utput. Matrices L, U,and P such that $M = LUP$, L is unit lower triangular, J is upper triangular, and P is a permutation matrix.

1ethod. We call FACTOR(M, n, n), where FACTOR is the recursive procedure shown in Fig. 6.4. The specification of FACTOR makes use of the liagrams of Figs. 6.1, 6.2, and 6.3, in which the shaded area of a matrix repesents a portion of the matrix known to be all zeros.

```
        procedure FACTOR($A$, $m$, $p$):
        if $m = 1$ then
            begin
 1.             set $L = [1]$ (i.e., $L$ is a unit $1 \times 1$ matrix);
 2.             find, if possible, a column $c$ of $A$ having a nonzero element and
                  let $P$ be the $p \times p$ permutation matrix which interchanges
                  columns 1 and $c$;
                comment Note $P = P^{-1}$;
 3.             let $U = AP$;
 4.             return ($L$, $U$, $P$)
            end
        else
            begin
 5.             partition $A$ into $(m/2) \times p$ matrices $B$ and $C$ as shown in
                  Fig. 6.2(a);
 6.             call FACTOR($B$, $m/2$, $p$) to produce $L_1$, $U_1$, $P_1$;
 7.             compute $D = CP_1^{-1}$;
                comment At this point, $A$ can be written as the product of the
                  three matrices shown in Fig. 6.2(b);
 8.             let $E$ and $F$ be, respectively, the first $m/2$ columns of $U_1$ and
                  $D$, as shown in Fig. 6.2(c);
 9.             compute $G = D - FE^{-1}U_1$;
                comment Note that the first $m/2$ columns of $G$ are all 0. $A$
                  may be written as the product of the matrices shown in
                  Fig. 6.2(d);
10.             let $G'$ be the rightmost $p - m/2$ columns of $G$;
11.             call FACTOR($G'$, $m/2$, $p - m/2$) to produce $L_2$, $U_2$, and $P_2$;
12.             let $P_3$ be the $p \times p$ permutation matrix with $I_{m/2}$ in the upper
                  left and $P_2$ in the lower right, as in Fig. 6.3(a);
13.             compute $H = U_1 P_3^{-1}$;
                comment At this time, the matrix composed of $U_1$ and $G$ can
                  be expressed as shown in Fig. 6.3(b). Substituting the right
                  side of Fig. 6.3(b) into Fig. 6.2(d) expresses $A$ as the product
                  of five matrices. The first two are unit lower triangular, the
                  third is upper triangular, and the last two are permutation
                  matrices. We shall multiply the first two together and the
                  last two together to get the desired decomposition of $A$:
14.             let $L$ be the $m \times m$ matrix consisting of $L_1$, $O_{m/2}$, $FE^{-1}$, and
                  $L_2$ as shown in Fig. 6.3(c);
15.             let $U$ be the $m \times p$ matrix with $H$ in the upper half and $O_{m/2}$
                  and $U_2$ in the lower half as shown in Fig. 6.3(c);
16.             let $P$ be the product $P_3 P_1$;
17.             return ($L$, $U$, $P$)
            end
```

Fig. 6.4. The procedure FACTOR.

Each recursive call of FACTOR is made on an $m \times p$ submatrix A of the $n \times n$ matrix M. At each call m is a power of 2 and $m \leq p \leq n$. The outputs of FACTOR are the three matrices L, U, and P shown in Fig. 6.1. □

Example 6.3. Let us find the LUP decomposition of the matrix

$$M = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix}.$$

To begin, we call FACTOR(M, 4, 4), which immediately calls

$$\text{FACTOR}\left(\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix}, 2, 4\right).$$

Letting this matrix be A, we call FACTOR([0 0 0 1], 1, 4), which returns

$$L_1 = [1],\ U_1 = [1 \quad 0 \quad 0 \quad 0], \quad \text{and} \quad P_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

which interchanges columns 1 and 4.

At line 7 we compute

$$D = CP_1^{-1} = [0 \quad 0 \quad 2 \quad 0] \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{\dagger} = [0 \quad 0 \quad 2 \quad 0].$$

Then at line 8 we have $E = [1]$ and $F = [0]$, so $G = D = [0 \quad 0 \quad 2 \quad 0]$ at line 9. At line 10 we have $G' = [0 \quad 2 \quad 0]$, so at line 11 we have

$$L_2 = [1], \qquad U_2 = [2 \quad 0 \quad 0], \qquad \text{and} \qquad P_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Next, at line 12

$$P_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

At line 13

$$H = U_1 P_3^{-1} = [1 \quad 0 \quad 0 \quad 0] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [1 \quad 0 \quad 0 \quad 0].$$

[†] In this example, all permutation matrices happen to be their own inverses.

Thus

$$\mathrm{FACTOR}\left(\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{bmatrix}, 2, 4\right)$$

returns

$$L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \quad \text{and} \quad P = P_3P_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

We now return to FACTOR(M, 4, 4) at line 6, with L, U, and P above becoming L_1, U_1, and P_1, respectively. Then at line 7 we compute

$$D = CP_1^{-1} = \begin{bmatrix} 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}.$$

At line 8 we find

$$E = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \quad \text{and} \quad F = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Thus

$$G = D = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

at line 9, and

$$G' = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix}$$

at line 10. It is left to the reader to check that the call to FACTOR(G', 2, 2) produces

$$L_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad U_2 = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix} \quad \text{and} \quad P_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Thus P_3 is I_4, an identity matrix at line 12, and

$$H = U_1P_3^{-1} = U_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

at line 13.

We thus find at lines 14–16 that

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}, \quad \text{and} \quad P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \quad \Box$$

Next we proceed to analyze and prove the correctness of Algorithm 6.1.

Theorem 6.3. Algorithm 6.1 calculates L, U, and P such that $A = LUP$ for any nonsingular matrix A.

Proof. We leave as exercises the details needed to show that the various decompositions depicted in Figs. 6.2 and 6.3 are correct. It is necessary only to show that

1. a nonzero column can always be found on line 2 of FACTOR, and
2. E^{-1} always exists on line 9.

Let A be an $m \times n$ matrix. We show by induction on m, where m is a power of 2, that if A has rank m, FACTOR will compute L, U, P such that $A = LUP$, L, U, and P are lower triangular, upper triangular, and permutation matrices of ranks m, m, and n, respectively. Furthermore, the first m columns of U are of rank m. If $m = 1$, then A must have a nonzero element and thus the induction hypothesis is clearly true. Assume $m = 2^k$, $k \geq 1$. Since A has n columns and is of rank m, B and C on line 5 each have $m/2$ columns, and each are of rank $m/2$. By the induction hypothesis the call to FACTOR on line 6 produces the desired L_1, U_1, and P_1, and the first $m/2$ columns of U_1 are of rank $m/2$. Thus E^{-1} exists on line 9.

From Fig. 6.2(d), we see that A is the product of three matrices, one of which is U_1 above G. Since A is of rank m, so must that matrix be of rank m. Therefore, G is of rank $m/2$. Since the first $m/2$ columns of G are zero and since G' is G minus the first $m/2$ columns, G' is also of rank $m/2$. Thus by the induction hypothesis the call to FACTOR on line 11 produces the desired L_2, U_2, P_2. The induction hypothesis follows in a straightforward manner. □

Before proceeding with a timing analysis, we observe that a permutation matrix can be represented by an array P such that $P[i] = j$ if and only if column i has its 1 in row j. Thus two $n \times n$ permutation matrices can be multiplied in time $O(n)$ by setting $P_1P_2[i] = P_1[P_2[i]]$. In this representation, the inverse of a permutation matrix may also be computed in time $O(n)$.

Theorem 6.4. Suppose for each n we can multiply two $n \times n$ matrices in time $M(n)$, where for all m and some $\epsilon > 0$ we have $M(2m) \geq 2^{2+\epsilon}M(m)$.† Then there is a constant k such that Algorithm 6.1 requires at most $kM(n)$ time on any nonsingular matrix.

Proof. Let Algorithm 6.1 be applied to an $n \times n$ matrix. Let $T(m)$ be the time required for a call FACTOR(A, m, p), where A is an $m \times p$ matrix, $m \leq p \leq n$. By lines 1–4 of FACTOR, we have $T(1) = bn$ for some constant b. For the recurrence, lines 6 and 11 each require time $T(m/2)$. Lines 7 and 13 each require the computation of the inverse of a permutation matrix which is $O(n)$ and the product of an arbitrary matrix by a permutation matrix. This product simply permutes the columns of the first matrix. Using the array representation P for a permutation matrix, we easily see that the $P[i]$th col-

† Intuitively this condition requires that $M(n)$ be in the range $n^{2+\epsilon}$ to n^3. It is possible, say, that $M(n) = kn^2 \log n$ for some constant k, in which case the hypothesis of the theorem is not satisfied.

umn of the first matrix will become the ith column of the product. Thus th product can be found in $O(mn)$ time, and lines 7 and 13 are $O(mn)$.

Line 9 requires $O(M(m/2))$ time to compute E^{-1} by Theorem 6.2, an the same time to compute the product FE^{-1}. Since U_1 is at most $(m/2) \times n$ the product $(FE^{-1})U_1$ can be computed in time

$$O\left(\frac{n}{m} M\left(\frac{m}{2}\right)\right).$$

Note that m divides n, since both are powers of 2 and $m \leq n$. The remaining steps are easily seen to be $O(mn)$ at worst. Thus we have the following recurrence.

$$T(m) \leq \begin{cases} 2T\left(\frac{m}{2}\right) + \frac{cn}{m} M\left(\frac{m}{2}\right) + dmn, & \text{if } m > 1, \\ bn, & \text{if } m = 1, \end{cases} \tag{6.5}$$

for constants b, c, and d.

By the theorem hypothesis and $M(1) \geq 1$, we have $M(m/2) \geq (m/2)^2$. Thus we can combine the second and third terms of (6.5). For some constant e

$$T(m) \leq \begin{cases} 2T\left(\frac{m}{2}\right) + \frac{en}{m} M\left(\frac{m}{2}\right), & \text{if } m > 1, \\ bn, & \text{if } m = 1. \end{cases} \tag{6.6}$$

From (6.6) we can obtain

$$\begin{aligned} T(m) &\leq \frac{en}{4m}\left[4M\left(\frac{m}{2}\right) + 4^2M\left(\frac{m}{2^2}\right) + \cdots + 4^{\log m}M(1)\right] + bnm \\ &\leq \frac{en}{4m} \sum_{i=1}^{\log m} 4^iM\left(\frac{m}{2^i}\right) + bnm. \end{aligned}$$

It follows from the theorem hypothesis that $4^iM(m/2^i) \leq (1/2^\epsilon)^iM(m)$. Thus

$$T(m) \leq \frac{en}{4m} M(m) \sum_{i=1}^{\infty} \left(\frac{1}{2^\epsilon}\right)^i + bnm.$$

Since the sum converges and $M(m) \geq m^2$, there exists a constant k such that $T(m) \leq (kn/m)M(m)$. For Algorithm 6.1 $n = m$ and hence $T(n) \leq kM(n)$. □

Corollary. Given any nonsingular matrix A we can find an LUP decomposition of A in $O(n^{2.81})$ steps.

Proof. By Theorems 6.1, 6.3, and 6.4. □

6.5 APPLICATIONS OF LUP DECOMPOSITION

In this section we shall show how LUP decomposition can be used to compute matrix inverses and determinants and to solve simultaneous linear equations. We shall see that each of these problems is reduced to the problem of multiplying two matrices, and thus any improvement in the asymptotic time complexity of matrix multiplication results in an improvement in the asymptotic time complexity of these problems. Conversely we shall show that matrix multiplication is reducible to matrix inversion, and thus matrix multiplication and matrix inversion are computationally equivalent problems.

Theorem 6.5. Let $\epsilon > 0$ and $a \geq 1$. Let $M(n)$ be the time required to multiply two matrices over some ring, and assume $M(2m) \geq 2^{2+\epsilon}M(m)$ for some $\epsilon > 0$. Then the inverse of any nonsingular matrix can be computed in $O(M(n))$ time.

Proof. Let A be any nonsingular $n \times n$ matrix. By Theorems 6.3 and 6.4 we can find $A = LUP$ in time $O(M(n))$. Thus $A^{-1} = P^{-1}U^{-1}L^{-1}$. P^{-1} is easy to compute in $O(n)$ steps. U^{-1} and L^{-1} exist and can be computed in $O(M(n))$ steps by Theorem 6.2. The product $P^{-1}U^{-1}L^{-1}$ can likewise be computed in $O(M(n))$ steps. □

Corollary. The inverse of an $n \times n$ matrix can be obtained in $O(n^{2.81})$ steps.

Theorem 6.6. If $M(n)$ is as in Theorem 6.5 and A is an $n \times n$ matrix, then we can compute $\mathbf{det}(A)$ in $O(M(n))$ steps.

Proof. Apply Algorithm 6.1 to find the LUP decomposition of A. If the algorithm fails to work because a nonzero column cannot be found at line 2 or E^{-1} does not exist at line 9, then A is singular and $\mathbf{det}(A) = 0$. Otherwise, let $A = LUP$. Then $\mathbf{det}(A) = \mathbf{det}(L)\ \mathbf{det}(U)\ \mathbf{det}(P)$. We can find $\mathbf{det}(L)$ and $\mathbf{det}(U)$ by taking the product of the main diagonal elements. Since L is unit lower triangular, $\mathbf{det}(L) = 1$. Since U is upper triangular, we can calculate $\mathbf{det}(U)$ in $O(n)$ steps. Since P is a permutation matrix, $\mathbf{det}(P) = \pm 1$, depending on whether P represents an odd or an even permutation. We can tell whether a permutation is odd or even by actually constructing the permutation from $(1, 2, \ldots, n)$ by interchanges. At most $n - 1$ interchanges are needed, and the number of permutations can be counted as they are performed. □

Corollary. The determinant of an $n \times n$ matrix can be computed in $O(n^{2.81})$ steps.

Example 6.4. Let us compute the determinant of the matrix M of Example 6.3. There, we determined the LUP decomposition

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

The determinants of the first and second of these matrices are the products of the diagonal terms, and thus are $+1$ and $+24$, respectively. We must determine only whether the last matrix P represents an odd or even permutation.

Since P represents the permutation (4, 3, 2, 1), and this permutation can b constructed by the sequence of two interchanges (1, 2, 3, 4) $\Rightarrow$ (4, 2, 3, 1) = (4, 3, 2, 1), we find that the permutation is even and $\mathbf{det}(P) = +1$. Thu $\mathbf{det}(M) = +24$. □

Theorem 6.7. Let $M(n)$ be as in Theorem 6.5, let A be a nonsingula $n \times n$ matrix, and let $\mathbf{b}$ be a column vector of length n. Let $\mathbf{x}$ be the co umn vector of unknowns $[x_1, x_2, \ldots, x_n]^T$. Then the solution to the se of simultaneous linear equations $A\mathbf{x} = \mathbf{b}$ can be obtained in $O(M(n))$ step

Proof. Write $A = LUP$ by Algorithm 6.1. Then $LUP\mathbf{x} = \mathbf{b}$ is solved in tw steps. First $L\mathbf{y} = \mathbf{b}$ is solved for the vector of unknowns $\mathbf{y}$, and then $UP\mathbf{x} =$ is solved for $\mathbf{x}$. Each subproblem can be solved by back substitution in $O(n^2$ steps, i.e., solve for y_1, substitute the value of y_1 for unknown y_1, and solv for y_2, etc. The LUP decomposition can be done in $O(M(n))$ steps b Theorem 6.4, and the soiution of $LUP\mathbf{x} = \mathbf{b}$ can then be obtained in $O(n^2$ steps. □

Corollary. n simultaneous equations in n unknowns can be solved i $O(n^{2.81})$ steps.

Finally we show that matrix multiplication and matrix inversion are o the same computational complexity.

Theorem 6.8. Let $M(n)$ and $I(n)$ be the amounts of time required t multiply two $n \times n$ matrices and to invert an $n \times n$ matrix, respectively Assume $8M(m) \geq M(2m) \geq 2^{2+\epsilon}M(m)$ for some $\epsilon > 0$, and analogousl for $I(n)$. Then $M(n)$ and $I(n)$ are, to within a constant factor, the same.

Proof. Theorem 6.5 shows that $I(n) \leq c_1M(n)$ for some c_1. To establish the relationship $M(n) \leq c_2I(n)$ for some c_2, let A and B be $n \times n$ matrices. Then

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}.$$

Thus we can obtain the product AB by inverting a $3n \times 3n$ matrix. It follows that $M(n) \leq I(3n) \leq I(4n) \leq 64I(n)$. □

6.6 BOOLEAN MATRIX MULTIPLICATION

In Section 5.9 we considered the problem of multiplying two Boolean matrices whose elements were chosen from the closed semiring {0, 1} with sum and product defined by

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

.

We showed that the problem of multiplying two Boolean matrices is equivalent to the computation of the transitive closure of a graph. Unfortunately, the closed semiring $\{0, 1\}$ is not a ring, and therefore Strassen's matrix-multiplication algorithm and the rest of the results presented so far in this chapter do not apply directly to Boolean matrix multiplication.

Clearly, the ordinary matrix-multiplication algorithm requires $O(n^3)$ steps.† However, there are at least two ways of multiplying Boolean matrices in less than $O(n^3)$ steps. The first is asymptotically better, but the second is likely to be more practical for moderate n. We present the first method in the next theorem.

Theorem 6.9. The product of two Boolean $n \times n$ matrices A and B can be computed in $O_A(n^{2.81})$ steps.

Proof. The integers modulo $n + 1$ form a ring $\mathbf{Z}_{n+1}$. We may use Strassen's matrix-multiplication algorithm to multiply matrices A and B in $\mathbf{Z}_{n+1}$. Let C be the product of A and B in $\mathbf{Z}_{n+1}$ and D be their product treated as Boolean matrices. Then it is easy to show that if $D[i, j] = 0$ then $C[i, j] = 0$, and if $D[i, j] = 1$ then $1 \leq C[i, j] \leq n$. Thus D can be immediately obtained from C. □

Corollary 1. If multiplication of two k-bit integers requires $m(k)$ bitwise operations, then Boolean matrix multiplication can be done in $O_B(n^{2.81}m(\log n))$ steps.

Proof. Since all arithmetic may be done in $\mathbf{Z}_{n+1}$, we need at most $\lfloor \log n \rfloor + 1$ bits to represent numbers. Multiplication of two such integers uses at most $O_B(m(\log n))$ time and addition or subtraction uses at most $O_B(\log n)$, which is surely no greater. □

In Chapter 7, we shall discuss a multiplication algorithm for which $m(k)$ is $O_B(k \log k \operatorname{loglog} k)$. Using this value we have the following corollary.

Corollary 2. Boolean matrix multiplication requires at most

$$O_B(n^{2.81} \log n \operatorname{loglog} n \operatorname{logloglog} n)$$

steps.

The second method, often called the "Four Russians'" algorithm, after the cardinality and nationality of its inventors, is somewhat more "practical" than the algorithm in Theorem 6.9. Moreover, the algorithm lends itself readily to bit vector calculations, which the algorithm in Theorem 6.9 does not.

† If almost all elements of the product are 1, then we can multiply two Boolean matrices in $O(n^2)$ expected time by computing each sum $\Sigma_{k=1}^{n} a_{ik}b_{kj}$ only until a term which is 1 is found. Then we know the sum is 1. If, for example, each element a_{ik} or b_{kj} has probability p of being 1, independently, then the expected number of terms we have to look at is at most $1/p^2$, independent of n.

Let us suppose we wish to multiply A and B, two $n \times n$ Boolean matrice: For convenience assume that $\log n$ divides n evenly. We can partition A int $n \times (\log n)$ submatrices and B into $(\log n) \times n$ submatrices as in Fig. 6.5. The we can write

$$AB = \sum_{i=1}^{n/\log n} A_iB_i.$$

Note that each product A_iB_i is an $n \times n$ matrix in itself. If we can comput each product A_iB_i in $O(n^2)$ steps, then we can compute AB in $O(n^3/\log n$ steps, since there are $n/\log n$ such products.

We now focus our attention on computing the products A_iB_i. We coul compute each product A_iB_i in $O(n^2 \log n)$ steps as follows. To compute A_iB_i we evaluate $\mathbf{a}_jB_i$ for each row $\mathbf{a}_j$ of A_i. To determine $\mathbf{a}_jB_i$, we find each rov k of B_i for which $\mathbf{a}_j$ has a 1 in column k. We then add together these rows o B_i, treating the rows as bit vectors of length n.

For example, suppose

$$A_i = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad \text{and} \quad B_i = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Then

$$C_i = A_iB_i = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

The first row of C_i is merely the third row of B_i, since the first row of A_i has a 1 in column 3 only. The second row of C_i is the sum of the first and third rows of B_i, since the second row of A_i has a 1 in columns 1 and 3, and so on.

To compute $\mathbf{a}_jB_i$ requires $O(n \log n)$ time for each row $\mathbf{a}_j$ of A_i. Since A_i has n rows, the total amount of work to compute A_iB_i is $O(n^2 \log n)$.

To compute the product A_iB_i faster, we observe that each row of A_i has $\log n$ elements, each of which is either 0 or 1. Thus there can be at most $2^{\log n} = n$ distinct rows among all the A_i's.

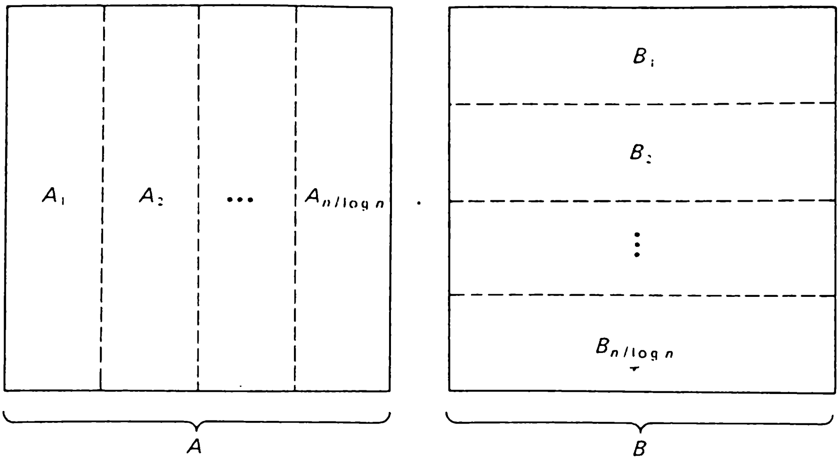


Fig. 6.5 Partition of Boolean matrices A and B.

Thus there are only n distinct possible sums of rows of B_i. We can precompute all possible sums of rows of B_i, and instead of computing $\mathbf{a}_j B_i$ we simply index into a table by $\mathbf{a}_j$ to look up the answer.

This method requires only $O(n^2)$ time to compute A_iB_i. The reasoning is as follows. Any subset of rows of B_i is empty, a singleton set, or the union of a singleton set and a smaller set. We can, if we choose the right order, compute each sum of rows by adding one row of B_i to a row sum already computed. We can then obtain all n sums of rows of B_i in $O(n^2)$ steps. After computing the sums and placing them in an array, we can select the appropriate sum for each of the n rows of A_i.

Algorithm 6.2. Four Russians' Boolean matrix multiplication algorithm.

Input. Two $n \times n$ Boolean matrices, A and B.

Output. The product $C = AB$.

Method. Define m to be $\lfloor \log n \rfloor$. Partition A into matrices $A_1, A_2, \ldots, A_{\lceil n/m \rceil}$, where A_i, $1 \le i < \lceil n/m \rceil$, consists of columns $m(i-1)+1$ through mi of A, and $A_{\lceil n/m \rceil}$ consists of the remaining columns, with extra columns of 0's if necessary to make m columns. Partition B into matrices $B_1, B_2, \ldots, B_{\lceil n/m \rceil}$, where B_i, $1 \le i < \lceil n/m \rceil$, consists of rows $m(i-1)+1$ through mi of B, and $B_{\lceil n/m \rceil}$ consists of the remaining rows with extra rows of 0's if necessary to make m rows. The situation is essentially that of Fig. 6.5. The computation is shown in Fig. 6.6. We use NUM(v) for the integer represented by the reverse of a vector v of 0's and 1's. For example, NUM([0, 1, 1]) = 6. □

Theorem 6.10. Algorithm 6.2 computes $C = AB$ and takes $O(n^3/\log n)$ steps.

```
     begin
1.       for $i \leftarrow 1$ until $\lceil n/m \rceil$ do
             begin
                 comment We compute the sums of the rows $\mathbf{b}_1^{(i)}, \ldots, \mathbf{b}($
                   of $B_i$:
2.               ROWSUM[0] $\leftarrow \underbrace{[0, 0, \ldots, 0]}_{n}$;
3.               for $j \leftarrow 1$ until $2^m - 1$ do
                     begin
4.                       let $k$ be such that $2^k \le j < 2^{k+1}$;
5.                       ROWSUM[$j$] $\leftarrow$ ROWSUM[$j - 2^k$] + $\mathbf{b}_{k+1}^{(i)}$†
                     end;
6.               let $C_i$ be the matrix whose $j$th row, $1 \le j \le n$, i
                   ROWSUM[NUM($\mathbf{a}_j$)], where $\mathbf{a}_j$ is the $j$th row of $A_i$
             end;
7.       let $C$ be $\sum_{i=1}^{\lceil n/m \rceil} C_i$
     end
```

† Bitwise Boolean sum is meant here, of course.

Fig. 6.6. Four Russians' algorithm.

Proof. An easy induction on j shows that at lines 2–5 ROWSUM[j] is se equal to the bitwise Boolean sum of those rows $\mathbf{b}_k$ of B_i such that the binar representation of j has 1 in the kth place from the right. It then follows tha $C_i = A_iB_i$ at line 6 and hence $C = AB$ at line 7.

For the time complexity of the algorithm, first consider the loop of line: 3–5. The assignment statement of line 5 clearly requires $O(n)$ steps. Th computation to evaluate k at line 4 is $O(m)$, which is less than $O(n)$, so th whole body of the loop, lines 4–5, is $O(n)$. The loop is repeated $2^m - 1$ times so the loop is $O(n2^m)$. Since $m \le \log n$, the loop of lines 3–5 is $O(n^2)$.

For line 6, computation of NUM($\mathbf{a}_j$) is $O(m)$, and copying vectoı ROWSUM [NUM($\mathbf{a}_j$)] is $O(n)$, so line 6 is $O(n^2)$. Since $\lceil n/m \rceil \le 2n/\log n$. the loop of lines 1–6, which is executed $\lceil n/m \rceil$ times, is $O(n^3/\log n)$. Similarly, step 7 requires at most $2n/\log n$ sums of $n \times n$ matrices, for a total ol $O(n^3/\log n)$ steps. Thus the entire algorithm requires $O(n^3/\log n)$ steps. □

What is perhaps more interesting is the fact that Algorithm 6.2 can bε implemented in $O_{BV}(n^2/\log n)$ bit vector steps, provided we have logical anc arithmetic operations on bit strings available.

Theorem 6.11. Algorithm 6.2 can be implemented in $O_{BV}(n^2/\log n)$ bil vector steps.

Proof. A counter is used to determine when to increment k. Initially, the counter has value 1 and k has value 0. Each time j is incremented, the counter is decremented by 1 unless it has value 1, in which case the counter is set to the new value of j, and k is incremented by 1.

The assignments of lines 2 and 5 in Fig. 6.6 require a constant number of steps. Thus the loop of lines 3–5 is $O_{BV}(n)$. In the construction of C_i in line 6, note that the computation of NUM($\mathbf{a}_i$) requires no time, since the representation of a bit vector in a RAM is by an integer.† Hence each row of C_i can be found in a constant number of bit vector steps, and line 6 requires $O_{BV}(n)$. Therefore the loop of lines 1–6 is $O_{BV}(n^2/\log n)$ and line 7 is of the same complexity. □

EXERCISES

6.1 Show that the integers modulo n form a ring. That is, $\mathbf{Z}_n$ is the ring $(\{0, 1, \ldots, n-1\}, +, \cdot, 0, 1)$, where $a + b$ and $a \cdot b$ are ordinary addition and multiplication modulo n.

6.2 Show that M_n, the set of $n \times n$ matrices with elements chosen from some ring R, itself forms a ring.

6.3 Give an example to show that the product of matrices is not commutative, even if the elements are chosen from a ring in which multiplication is commutative.

6.4 Use Strassen's algorithm to compute the product

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

6.5 Another version of Strassen's algorithm uses the following identities to help compute the product of two 2×2 matrices.

$$\begin{array}{lll}
s_1 = a_{21} + a_{22} & m_1 = s_2 s_6 & t_1 = m_1 + m_2 \\
s_2 = s_1 - a_{11} & m_2 = a_{11} b_{11} & t_2 = t_1 + m_4 \\
s_3 = a_{11} - a_{21} & m_3 = a_{12} b_{21} & \\
s_4 = a_{12} - s_2 & m_4 = s_3 s_7 & \\
s_5 = b_{12} - b_{11} & m_5 = s_1 s_5 & \\
s_6 = b_{22} - s_5 & m_6 = s_4 b_{22} & \\
s_7 = b_{22} - b_{12} & m_7 = a_{22} s_8 & \\
s_8 = s_6 - b_{21} & &
\end{array}$$

The elements of the product matrix are:

$$\begin{aligned}
c_{11} &= m_2 + m_3, \\
c_{12} &= t_1 + m_5 + m_6, \\
c_{21} &= t_2 - m_7, \\
c_{22} &= t_2 + m_5.
\end{aligned}$$

Show that these elements compute Eq. (6.1). Note that only 7 multiplications and 15 additions have been used.

† We can get around the detail that NUM($\mathbf{a}_i$) is the integer representing the reverse of $\mathbf{a}_i$ by taking the "jth row" of B_i to be the jth row from the bottom instead of the top as we have previously done.

6.6 Prove the following for $n \times n$ matrices A, B, and C.

a) $AB = I$ and $AC = I$ implies $B = C$.

b) $A^{-1}A = I$.

c) $(AB)^{-1} = B^{-1}A^{-1}$.

d) $(A^{-1})^{-1} = A$.

e) $\det(AB) = \det(A)\det(B)$.

6.7 Theorem 6.2 shows that the inverse of a nonsingular upper triangular matrix can be taken in $cn^{2.81}$ arithmetic operations for some c. Find the constant c on the assumption that matrix multiplication is by Strassen's algorithm and n is a power of 2.

6.8 Represent a permutation matrix by an array P such that $P[i] = j$ if and only if column i has its 1 in row j. Let P_1 and P_2 be representations of $n \times n$ permutation matrices.

a) Prove $P_1P_2[i] = P_1[P_2[i]]$.

b) Give an $O(n)$ algorithm for computing P_1^{-1}.

c) Change the representation such that $P[i] = j$ if and only if row i has its 1 in column j. Give the correct formula for P_1P_2 and the algorithm for computing P_1^{-1}.

6.9 Use Algorithm 6.1 to find the LUP decomposition of the matrix

$$M = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 3 & 0 \\ 1 & -1 & 0 & 1 \\ 2 & 0 & -1 & 3 \end{bmatrix}.$$

6.10 We have shown that LUP decomposition, matrix inversion, determinant computation, and solution of linear equations are each $O_A(n^{2.81})$. Find the best possible constant factor for each problem, assuming that Strassen's algorithm is used for the multiplications, n is a power of 2, and the techniques of Algorithm 6.1 and Theorems 6.4–6.7 are used.

6.11 Find (a) the inverse and (b) the determinant of the matrix M of Exercise 6.9, using the techniques of this chapter.

6.12 Solve the following set of simultaneous linear equations using LUP decomposition.

$$\begin{aligned} x_3 + 2x_4 &= 7 \\ 3x_3 \qquad &= 9 \\ x_1 - x_2 \qquad + x_4 &= 3 \\ 2x_1 \qquad - x_3 + 3x_4 &= 10 \end{aligned}$$

6.13 Show that every permutation is either even or odd, but not both.

6.14 Compute the following Boolean matrix product using (a) the method of Theorem 6.9, and (b) the Four Russians' Algorithm.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.15 Complete the proof of Theorem 6.3 by showing that the relationships shown in Figs. 6.2 and 6.3 are valid.

****6.16** Consider the field† F_2 of integers modulo 2. Find a multiplication algorithm for $n \times n$ matrices over F_2 with an asymptotic bound of $n^{2.81}/(\log n)^{0.4}$. [*Hint:* Partition the matrices into blocks of size $\sqrt{\log n} \times \sqrt{\log n}$.]

6.17 Estimate the value of n beyond which $n^{2.81}$ is less than $n^3/\log n$.

***6.18** Let $L(n)$ be the time to multiply two $n \times n$ lower triangular matrices. Let $T(n)$ be the time to multiply two arbitrary matrices. Prove that there exists a constant c such that $T(n) \leq cL(n)$.

6.19 Prove that the inverse of an upper (lower) triangular matrix is upper (lower) triangular.

***6.20** Let $I(n)$ and $U(n)$ be the number of steps needed to invert an $n \times n$ matrix and an $n \times n$ upper triangular matrix, respectively. Prove there exists a constant c such that $I(n) \leq cU(n)$ for all n.

****5.21** To compute the matrix product $C = AB$ one could compute the product $D = (PAQ)(Q^{-1}BR)$ and then compute $C = P^{-1}DR^{-1}$. If P, Q, and R are certain matrices, e.g., permutationmatrices, the multiplications PAQ, $Q^{-1}BR$, and $P^{-1}DR^{-1}$ require no multiplications of ring elements. Use this idea to find another method of multiplying 2×2 matrices in seven multiplications.

6.22 Prove that the LU decomposition of a nonsingular matrix A is unique whenever it exists. [*Hint:* Assume $A = L_1U_1 = L_2U_2$. Show that $L_2^{-1}L_1 = U_2U_1^{-1} = I$.]

6.23 Prove that if A is nonsingular and every principal submatrix of A is nonsingular, then A has an LUP decomposition.

6.24 Can a singular matrix have an LUP decomposition?

****6.25** Let A be an $n \times n$ matrix over the reals. A is *positive definite* if for each nonzero column vector $\mathbf{x}$, $\mathbf{x}^TA\mathbf{x} > 0$.

a) Show that Lemma 6.5 can be used to invert any nonsingular symmetric positive definite matrix.
b) Prove that AA^T is always positive definite and symmetric.
c) Using (a) and (b) give an $O(M(n))$ algorithm for matrix inversion of arbitrary nonsingular matrices over the reals.
d) Does your algorithm in (c) work for the field of integers modulo 2?

***6.26** An $n \times n$ *Toeplitz* matrix is a matrix A with the property that

$$A[i, j] = A[i-1, j-1], \; 2 \leq i, j \leq n.$$

a) Find a representation for a Toeplitz matrix so that two $n \times n$ Toeplitz matrices can be added in $O(n)$ operations.
b) Find a divide-and-conquer algorithm for multiplying an $n \times n$ Toeplitz matrix by a column vector. How many arithmetic operations are required? [*Hint:* Using the divide-and-conquer approach one can obtain an $O(n^{1.59})$ algorithm. In Chapter 7 we develop techniques which can be used to improve upon this result.]
c) Find an asymptotically efficient algorithm for multiplying two $n \times n$ Toeplitz

† See Section 12.1 for a definition of "field."

matrices. How many arithmetic operations are required? Note that the product of Toeplitz matrices is not necessarily Toeplitz.

Research Problems

6.27 A natural problem is to improve on Strassen's method directly. It has been shown by Hopcroft and Kerr [1971] that seven multiplications are required for 2×2 matrix multiplication over an arbitrary ring. However, a recursive algorithm could be based on some other small-sized matrix. For example, we could asymptotically improve on Strassen if we did 3×3 matrix multiplication in 21 multiplications or 4×4 in 48.

6.28 Can we do the shortest-path problem in less than $O(n^3)$ steps? Strassen's algorithm does not apply to the closed semiring consisting of the nonnegative reals with $+\infty$, but it may be possible to embed this closed semiring in a ring as we did for Boolean matrices.

BIBLIOGRAPHIC NOTES

Strassen's algorithm is taken from Strassen [1969]. Winograd [1973] reduced the number of additions required to 15, which improved the constant factor but not the order-of-magnitude complexity (see Exercise 6.5).

Strassen [1969] also gave $O(n^{2.81})$ methods for matrix inversion, computation of determinants, and solution of simultaneous linear equations, on the assumption that each matrix encountered during the recursion was nonsingular. Bunch and Hopcroft [1974] showed that LUP decomposition could be done in $O(n^{2.81})$ steps with only the condition that the initial matrix be nonsingular. A. Schönhage independently showed that inversion of any nonsingular matrix over an ordered field could be done in $O(n^{2.81})$ steps (Exercise 6.25).

The result that matrix multiplication is no harder than inversion is by Winograd [1970c]. The $O_A(n^{2.81})$ algorithm for Boolean matrix multiplication is by Fischer and Meyer [1971]. The "Four Russians" are Arlazarov, Dinic, Kronrod, and Faradzev [1970].

For additional reading on the mathematics of matrices consult Hohn [1958]. Algebraic concepts, such as the theory of rings, can be found in MacLane and Birkhoff [1967]. A solution to Exercise 6.13 can be found in Birkhoff and Bartee [1970]. Exercise 6.16 is by J. Hopcroft.

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS

CHAPTER 7

The Fourier transform arises naturally in many fields of science and engineering, and thus an efficient algorithm for computing the Fourier transform is of interest. The ubiquity of the Fourier transform is further demonstrated by its applicability to the design of efficient algorithms. In many applications it is convenient to transform a problem into another, easier problem. An example occurs in computing the product of two polynomials. It is computationally expedient first to apply a linear transformation to the coefficient vectors of the polynomials, then to perform an operation simpler than convolution on the images of the coefficients, and finally to apply the inverse transformation to the result to get the desired product. An appropriate linear transformation for this situation is the discrete Fourier transform.

In this chapter we shall study the Fourier transform, its inverse, and its role in computing convolutions and products of various types. We shall develop an efficient algorithm called the fast Fourier transform (FFT) for computing the Fourier transform. The algorithm, which is based on techniques of polynomial evaluation by division, makes use of the fact that a polynomial is being evaluated at the roots of unity.

Then we shall prove the Convolution Theorem. We shall interpret convolution as polynomial evaluation at roots of unity, multiplication of sample values, and finally polynomial interpolation. We shall develop an efficient algorithm for convolution using the fast Fourier transform, and apply the convolution algorithm to symbolic multiplication of polynomials and integer multiplication. The resulting algorithm for integer multiplication, the so-called Schönhage-Strassen algorithm, is asymptotically the fastest known way to multiply two integers together.

7.1 THE DISCRETE FOURIER TRANSFORM AND ITS INVERSE

The Fourier transform is usually defined over the complex numbers. For reasons which will become apparent later, we shall define the Fourier transform over an arbitrary commutative ring $(R, +, \cdot, 0, 1)$.† An element ω of R such that

1. $\omega \neq 1$,
2. $\omega^n = 1$, and
3. $\sum_{j=0}^{n-1} \omega^{jp} = 0$, for $1 \leq p < n$,

is said to be a *principal* nth *root of unity*. The elements $\omega^0, \omega^1, \ldots, \omega^{n-1}$ are the nth *roots of unity*.

For example, $e^{2\pi i/n}$, where $i = \sqrt{-1}$, is a principal nth root of unity in the ring of complex numbers.

† Recall that a commutative ring is one in which multiplication (as well as addition) is commutative.

Let $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ be a length-n (column) vector with elements from R. We assume the integer n has a multiplicative inverse in R† and that R has a principal nth root of unity ω. Let A be an $n \times n$ matrix such that $A[i, j] = \omega^{ij}$, for $0 \le i, j < n$. The vector $F(\mathbf{a}) = A\mathbf{a}$ whose ith component b_i, $0 \le i < n$, is $\Sigma_{k=0}^{n-1} a_k \omega^{ik}$ is called the *discrete Fourier transform* of $\mathbf{a}$. The matrix A is nonsingular, and thus A^{-1} exists. A^{-1} has the simple form given in Lemma 7.1.

Lemma 7.1. Let R be a commutative ring having a principal nth root of unity ω, where n has a multiplicative inverse in R. Let A be the $n \times n$ matrix whose ijth element is ω^{ij} for $0 \le i,j < n$. Then A^{-1} exists and the ijth element of A^{-1} is $(1/n)\omega^{-ij}$.

Proof. Let δ_{ij} be 1 if $i = j$ and 0 otherwise. It suffices to show that if A^{-1} is defined as above, then $A \cdot A^{-1} = I_n$, that is, the ijth element in $A \cdot A^{-1}$ is

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega^{ik}\omega^{-kj} = \delta_{ij} \qquad \text{for} \quad 0 \le i, j < n. \tag{7.1}$$

If $i = j$, then the left-hand side of (7.1) reduces to

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega^0 = 1.$$

If $i \ne j$, let $q = i - j$. Then the left-hand side of (7.1) reduces to

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega^{qk}, \qquad -n < q < n, \quad q \ne 0.$$

If $q > 0$, we have

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega^{qk} = 0,$$

since ω is an nth root of unity. If $q < 0$, then multiplying by $\omega^{-q(n-1)}$, rearranging the order of the terms in the summation, and replacing q by $-q$ yields

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega^{qk}, \qquad 0 < q < n,$$

which again has value 0 since ω is a principal nth root of unity. Equation (7.1) follows immediately. □

The vector $F^{-1}(\mathbf{a}) = A^{-1}\mathbf{a}$ whose ith component, $0 \le i < n$, is

$$\frac{1}{n}\sum_{k=0}^{n-1} a_k\omega^{-ik}$$

† The integers appear in any ring, even finite ones. Take n to be $1 + 1 + \cdots + 1$ (n times), where 1 is the multiplicative identity. Note that $\omega\omega^{-1} = 1$ so ω has an inverse and thus it makes sense to talk about negative powers of ω.

is the *inverse discrete Fourier transform* of **a**. Clearly, the inverse transform of the transform of **a** is **a** itself, i.e., $F^{-1}F(\mathbf{a}) = \mathbf{a}$.

There is a close relationship between Fourier transforms and polynomial evaluation and interpolation. Let

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

be an $(n-1)$st-degree polynomial. This polynomial can be uniquely represented in two ways, either by a list of its coefficients $a_0, a_1, \ldots, a_{n-1}$ or by a list of its values at n distinct points $x_0, x_1, \ldots, x_{n-1}$. The process of finding the coefficient representation of a polynomial given its values at x_0, x_1, $\ldots, x_{n-1}$ is called *interpolation.*

Computing the Fourier transform of a vector $[a_0, a_1, \ldots, a_{n-1}]^T$ is equivalent to converting the coefficient representation of the polynomial $\Sigma_{i=0}^{n-1} a_i x^i$ to its value representation at the points $\omega^0, \omega^1, \ldots, \omega^{n-1}$. Likewise, the inverse Fourier transform is equivalent to interpolating a polynomial given its values at the nth roots of unity.

One could define a transform which evaluates a polynomial at a set of points other than the nth roots of unity. For example, one could use the integers $1, 2, \ldots, n$. However, we shall see that by choosing the powers of ω, evaluation and interpolation become particularly straightforward. In Chapter 8 the Fourier transform is used to evaluate and interpolate polynomials at arbitrary points.

One of the principal applications of the Fourier transform is in computing the convolution of two vectors. Let

$$\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T \text{ and } \mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]^T$$

be two column vectors. The *convolution* of **a** and **b**, denoted $\mathbf{a} \circledast \mathbf{b}$, is the vector $\mathbf{c} = [c_0, c_1, \ldots, c_{2n-1}]^T$, where $c_i = \Sigma_{j=0}^{n-1} a_j b_{i-j}$. (Take $a_k = b_k = 0$ if $k < 0$ or $k \geq n$). Thus

$$c_0 = a_0 b_0,\ c_1 = a_0 b_1 + a_1 b_0,\ c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0,$$

and so on. Note that $c_{2n-1} = 0$; this term is included for symmetry only.

To motivate the convolution, consider again the representation of a polynomial by its coefficients. The product of two $(n-1)$st-degree polynomials

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \qquad \text{and} \qquad q(x) = \sum_{j=0}^{n-1} b_j x^j$$

is the $(2n-2)$nd-degree polynomial

$$p(x)q(x) = \sum_{i=0}^{2n-2} \left[\sum_{j=0}^{i} a_j b_{i-j} \right] x^i.$$

Observe that the coefficients of the product polynomial are exactly the components of the convolution of the coefficient vectors $[a_0, a_1, \ldots a_n]^T$ and $[b_0, b_1, \ldots, b_n]^T$ of the original polynomials, if we neglect c_{2n-1}, which is zero.

If the two $(n-1)$st-degree polynomials are represented by their coefficients, then to compute the coefficient representation of their product we can convolve the two coefficient vectors. On the other hand if $p(x)$ and $q(x)$ are represented by their values at the nth roots of unity, then to compute the values representation of their product we can simply multiply pairs of values at corresponding roots. This suggests that the convolution of the two vectors **a** and **b** is the inverse transform of the componentwise product of the transform of the two vectors. Symbolically, $\mathbf{a} \circledast \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$. That is, a convolution may be computed by taking Fourier transforms, computing their pairwise product, and then inverting. The only problem is that the product of two $(n-1)$st-degree polynomials is in general a $(2n-2)$nd-degree polynomial, and to represent it requires values at $2n-2$ distinct points. This technicality is resolved in the following theorem by considering $p(x)$ and $q(x)$ to be $(2n-1)$st-degree polynomials, where the coefficients of the $(n-1)$st highest powers of x are zero [i.e., treating $(n-1)$st-degree polynomials as $(2n-1)$st-degree polynomials].

Theorem 7.1 (Convolution Theorem). Let

$$\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0]^T$$

and

$$\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}, 0, \ldots, 0]^T$$

be column vectors of length $2n$. Let

$$F(\mathbf{a}) = [a'_0, a'_1, \ldots, a'_{2n-1}]^T$$

and

$$F(\mathbf{b}) = [b'_0, b'_1, \ldots, b'_{2n-1}]^T$$

be their Fourier transforms. Then $\mathbf{a} \circledast \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$.

Proof. Since $a_i = b_i = 0$ for $n \le i < 2n$, we note that for $0 \le l < 2n$,

$$a'_l = \sum_{j=0}^{n-1} a_j \omega^{lj} \qquad \text{and} \qquad b'_l = \sum_{k=0}^{n-1} b_k \omega^{lk}.$$

Thus

$$a'_l b'_l = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)}. \tag{7.2}$$

Let $\mathbf{a} \circledast \mathbf{b} = [c_0, c_1, \ldots, c_{2n-1}]^T$ and $F(\mathbf{a} \circledast \mathbf{b}) = [c'_0, c'_1, \ldots, c'_{2n-1}]^T$.

Since $c_p = \sum_{j=0}^{2n-1} a_j b_{p-j}$ we have

$$c'_l = \sum_{p=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{p-j} \omega^{lp}. \tag{7.3}$$

Interchanging the order of summation in (7.3) and substituting k for $p-j$ yields

$$c'_l = \sum_{j=0}^{2n-1} \sum_{k=-j}^{2n-1-j} a_j b_k \omega^{l(j+k)}. \tag{7.4}$$

Since $b_k = 0$ for $k < 0$, we may raise the lower limit on the inner summation to $k = 0$. Likewise, since $a_j = 0$ for $j \geq n$, we may lower the upper limit on the outer summation to $n - 1$. The upper limit on the inner summation is at least n no matter what the value of j is. Thus we may replace the upper limit by $n - 1$ since $b_k = 0$ for $k \geq n$. When we make these changes, (7.4) becomes identical to (7.2), hence $c'_l = a'_l b'_l$. We have now shown $F(\mathbf{a} \circledast \mathbf{b}) = F(\mathbf{a}) \cdot F(\mathbf{b})$, from which it follows that $\mathbf{a} \circledast \mathbf{b} = F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$. □

The convolution of two vectors of length n is a vector of length $2n$. This requires "padding" with n zeros the vectors **a** and **b** in the Convolution Theorem. To avoid this "padding," we can use "wrapped" convolutions.

Definition. Let $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ and $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]^T$ be two vectors of length n. The *positive wrapped convolution of* **a** and **b** is the vector $\mathbf{c} = [c_0, c_1, \ldots, c_{n-1}]^T$, where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}.$$

The *negative wrapped convolution* of **a** and **b** is the vector $\mathbf{d} = [d_0, d_1, \ldots, d_{n-1}]^T$, where

$$d_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}.$$

We shall later make use of the wrapped convolutions in Section 7.5 in the Schönhage–Strassen algorithm, a fast integer-multiplication algorithm. For the moment observe that one can evaluate two $n - 1$st-degree polynomials at the nth roots of unity and multiply the pairs of values at the corresponding roots together. This gives n values through which we can interpolate a unique $n - 1$st-degree polynomial. The vector of coefficients of this unique polynomial is precisely the positive wrapped convolution of the vectors of coefficients of the two original polynomials.

Theorem 7.2. Let $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$ and $\mathbf{b} = [b_0, b_1, \ldots, b_{n-1}]^T$ be two vectors of length n. Let ω be a principal nth root of unity and let $\psi^2 = \omega$. Assume that n has a multiplicative inverse.

1. The positive wrapped convolution of **a** and **b** is given by $F^{-1}(F(\mathbf{a}) \cdot F(\mathbf{b}))$.
2. Let $\mathbf{d} = [d_0, d_1, \ldots, d_{n-1}]^T$ be the negative wrapped convolution of **a** and **b**. Let $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{d}}$ be $[a_0, \psi a_1, \ldots, \psi^{n-1}a_{n-1}]^T$, $[b_0, \psi b_1, \ldots, \psi^{n-1}b_{n-1}]^T$, and $[d_0, \psi d_1, \ldots, \psi^{n-1}d_{n-1}]^T$, respectively. Then $\hat{\mathbf{d}} = F^{-1}(F(\hat{\mathbf{a}}) \cdot F(\hat{\mathbf{b}}))$.

Proof. The proof is analogous to that of Theorem 7.1 with the observation that $\psi^n = -1$. The details are left for an exercise. □

7.2 THE FAST FOURIER TRANSFORM ALGORITHM

It is clear that we can compute the Fourier transform and inverse Fourier transform of a vector **a** in R^n in $O_A(n^2)$ time if we assume that arithmetic operations on arbitrary elements of the ring R require one step each. However, when n is a power of 2, we can do much better; there is an $O_A(n \log n)$ algorithm to evaluate the Fourier transform or the inverse Fourier transform, and we conjecture that this is optimal. We give the algorithm for the Fourier transform itself. The algorithm for the inverse transform is analogous and is left for the reader. The basic idea behind the fast Fourier transform (FFT) is algebraic in nature; we find similarities among portions of the n sums implied by $A\mathbf{a}$. Throughout the section we assume $n = 2^k$ for some integer k.

Recall that evaluating $A\mathbf{a}$ is equivalent to evaluating the polynomial $p(x) = \sum_{j=0}^{n-1} a_j x^j$ for x equal to $\omega^0, \omega^1, \ldots, \omega^{n-1}$. However, evaluation of a polynomial $p(x)$ at a point $x = a$ is equivalent to finding the remainder when $p(x)$ is divided by $x - a$. [To see this, we can write $p(x) = (x - a)q(x) + c$, where c is a constant. Then $p(a) = c$.] Thus evaluation of the Fourier transform reduces to finding the remainder when an $(n - 1)$st-degree polynomial $p(x) = \sum_{j=0}^{n-1} a_j x^j$ is divided by each of $x - \omega^0, x - \omega^1, \ldots, x - \omega^{n-1}$.

Simply dividing $p(x)$ by each of the $x - \omega^i$ in turn is an $O(n^2)$ process. To obtain a faster algorithm we multiply the $x - \omega^i$ together in pairs, then multiply the resulting $n/2$ polynomials together in pairs, and so on until finally we are left with two polynomials, q_1 and q_2, each of which is a product of half of the $x - \omega^i$'s. Next, we divide $p(x)$ by q_1 and by q_2, obtaining remainders $r_1(x)$ and $r_2(x)$, respectively, which are each of degree at most $n/2 - 1$. For each ω^i such that $x - \omega^i$ is a factor of q_1, finding the remainder of $p(x)$ divided by $x - \omega^i$ is equivalent to finding the remainder of $r_1(x)$ divided by $x - \omega^i$. A similar statement is true for each ω^i such that $x - \omega^i$ is a factor of q_2. Thus computing the remainders of $p(x)$ when divided by each of the $x - \omega^i$'s is equivalent to computing the remainders of $r_1(x)$ and $r_2(x)$ when divided by each of the $n/2$ appropriate $x - \omega^i$'s. Recursively applying this divide-and-conquer approach is much more efficient than the straightforward method of dividing $p(x)$ by each $x - \omega^i$.

In multiplying the $x - \omega^i$'s together one would expect the products to have cross product terms. However, by appropriately arranging the order of the $x - \omega^i$'s we can make all resulting products to be of the form $x^j - \omega^i$, further reducing the time spent in multiplying and dividing polynomials. We now make these ideas more precise.

Let $c_0, c_1, \ldots, c_{n-1}$ be a permutation of $\omega^0, \omega^1, \ldots, \omega^{n-1}$ which will be specified later. We define polynomials q_{lm}, for $0 \le m \le k$ and for l an integer multiple of 2^m in the range $0 \le l \le 2^k - 1$, as follows:

$$q_{lm} = \prod_{j=l}^{l+2^m-1} (x - c_j).$$

Thus q_{0k} is the product $(x - c_0)(x - c_1) \cdots (x - c_{n-1})$, q_{l0} is $x - c_l$, and in general

$$q_{lm} = q_{l,m-1} q_{l+2^{m-1},m-1}.$$

There are 2^{k-m} polynomials with second subscript m and each $x - c_l$ is a factor of exactly one of them. The polynomials q_{lm} are illustrated in Fig. 7.1. (Also see Sections 8.4 and 8.5.)

Our goal is to compute the remainder of $p(x)/q_{l0}(x)$ for each l. To do this we compute the remainders of $p(x)/q_{lm}(x)$ for each q_{lm} starting at $m = k - 1$ and ending at $m = 0$.

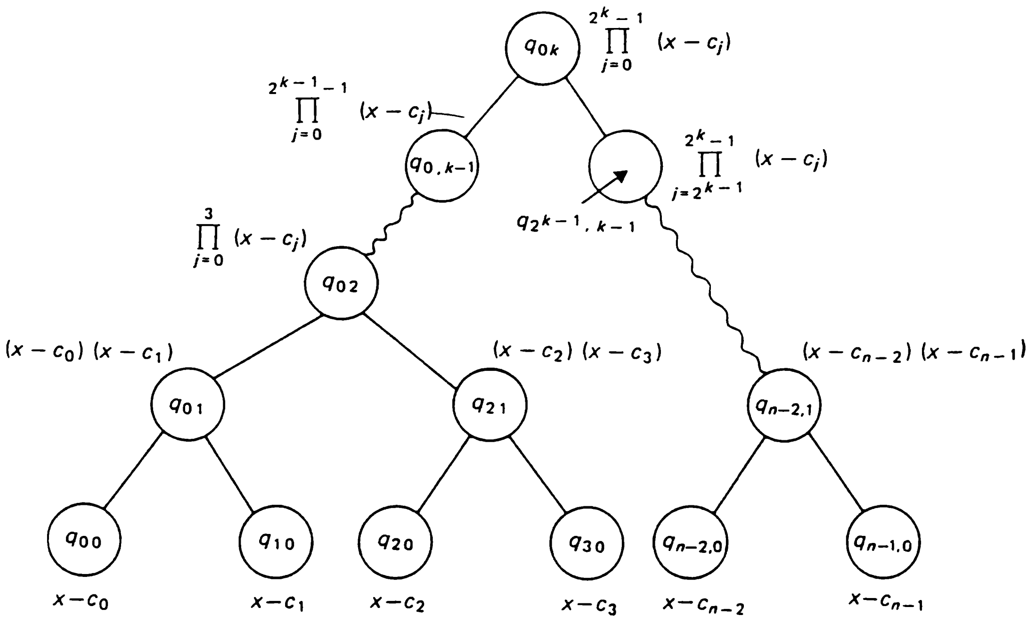


Fig. 7.1 The polynomials q_{lm}.

Suppose we have already computed the $(2^m - 1)$st-degree polynomial r_{lm} which is the remainder of $p(x)/q_{lm}(x)$. [We can assume $r_{0k} = p(x)$.] Since $q_{lm} = q'q''$ where $q' = q_{l,m-1}$ and $q'' = q_{l+2^{m-1},m-1}$, we claim that $p(x)/q'(x)$ has the same remainder as $r_{lm}/q'(x)$ and that a similar statement holds for $q''(x)$. In proof, let

$$p(x) = h_1(x)q'(x) + r_{l,m-1},$$

where the degree of $r_{l,m-1}$ is at most $2^{m-1} - 1$. Since

$$p(x) = h_2(x)q_{lm}(x) + r_{lm}$$

we have

$$h_1(x)q'(x) + r_{l,m-1} = h_2(x)q_{lm}(x) + r_{lm}. \tag{7.5}$$

If both sides of (7.5) are divided by $q'(x)$, we observe that $h_1(x)q'(x)$ and $h_2(x)q_{lm}(x)$ leave no remainder, so the remainder of $r_{lm}/q'(x)$ is $r_{l,m-1}$.

Thus we can obtain the remainders of $p(x)/q'(x)$ and $p(x)/q''(x)$ by dividing the $(2^{k-m} - 1)$st-degree polynomial r_{lm} by $q'(x)$ and $q''(x)$, rather than by dividing the $(2^k - 1)$st-degree polynomial $p(x)$ by $q'(x)$ and $q''(x)$. This method of performing the divisions in itself is a saving. However we can do more. By choosing a judicious ordering of $c_0, c_1, \ldots, c_{n-1}$ for the powers of ω, we can guarantee that each polynomial q_{lm} is of the form $x^{2^m} - \omega^s$ for some s. Division by such a polynomial is especially easy.

Lemma 7.2. Let $n = 2^k$ and let ω be a principal nth root of unity. For $0 \leq j < 2^k$ let $[d_0 d_1 \cdots d_{k-1}]$ be the binary representation of the integer j† and let **rev**(j) be the integer whose binary representation is $[d_{k-1} d_{k-2} \cdots d_0]$. Let c_j be $\omega^{\mathrm{rev}(j)}$ and let $q_{lm} = \Pi_{j=l}^{l+2^m-1}(x - c_j)$. Then $q_{lm} = x^{2^m} - \omega^{\mathrm{rev}(l/2^m)}$

Proof. The proof is by induction on m. The basis, $m = 0$, is trivial, since by definition $q_{l0} = x - c_l = x - \omega^{\mathrm{rev}(l)}$. For the inductive step, we know for $m > 0$,

$$\begin{aligned} q_{lm} &= q_{l,m-1}q_{l+2^{m-1},m-1} \\ &= (x^{2^{m-1}} - \omega^{\mathrm{rev}(l/2^{m-1})})(x^{2^{m-1}} - \omega^{\mathrm{rev}(l/2^{m-1}+1)}), \end{aligned}$$

where $l/2^{m-1}$ is an even integer in the range 0 to $2^k - 1$. Then

$$\omega^{\mathrm{rev}(l/2^{m-1}+1)} = \omega^{2^{k-1}+\mathrm{rev}\,(l/2^{m-1})} = -[\omega^{\mathrm{rev}\,(l/2^{m-1})}],$$

† i.e., $j = \sum_{i=0}^{k-1} d_{k-1-i}2^i$

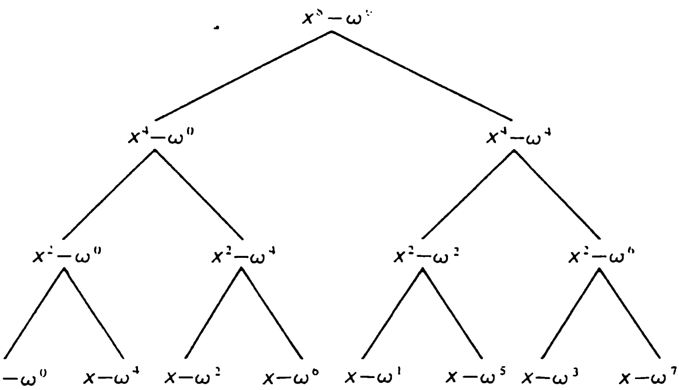


Fig. 7.2 Illustration of q_{lm} of Lemma 7.2.

since $\omega^{2^{k-1}} = \omega^{n/2} = -1$. Thus

$$q_{lm} = x^{2^m} - \omega^{2\text{rev}(l/2^{m-1})} = x^{2^m} - \omega^{\text{rev}(l/2^m)},$$

since $\text{rev}(2t) = \frac{1}{2}\text{rev}(t)$. □

Example 7.1. If $n = 8$, the list $c_0, c_1, \ldots, c_7$ is $\omega^0, \omega^4, \omega^2, \omega^6, \omega^1, \omega^5, \omega^3, \omega^7$. The q_{lm} are illustrated in Fig. 7.2.

The q_{lm} are used to compute the remainders as follows. Initially we evaluate r_{02} and r_{42}, the remainders of $p(x)/(x^4 - \omega^0)$ and $p(x)/(x^4 - \omega^4)$ where $p(x) = \Sigma_{j=0}^7 a_j x^j$. Then we evaluate r_{01} and r_{21} the remainders of $r_{02}/(x^2 - \omega^0)$ and $r_{02}/(x^2 - \omega^4)$, and r_{41} and r_{61}, the remainders of $r_{42}/(x^2 - \omega^2)$ and $r_{42}/(x^2 - \omega^6)$. Finally we evaluate $r_{00}, r_{10}, r_{20}, \ldots, r_{70}$ where r_{00} and r_{10} are the remainders of $r_{01}/(x - \omega^0)$ and$r_{01}/(x - \omega^4)$, r_{20} and r_{30} are the remainders of $r_{21}/(x - \omega^2)$ and $r_{21}/(x - \omega^6)$, and so on.

For more examples of this approach, also see Section 8.5. □

Having shown that the q_{lm} are of the form $x^s - c$, we now show that the remainder of $p(x)$ when divided by $x^s - c$ is easy to compute.

Lemma 7.3. Let

$$p(x) = \sum_{j=0}^{2t-1} a_j x^j$$

and let c be a constant. Then the remainder of $p(x)/(x^t - c)$ is

$$r(x) = \sum_{j=0}^{t-1} (a_j + c a_{j+t}) x^j.$$

Proof. Simply observe that $p(x)$ can be expressed as

$$\left[\sum_{j=0}^{t-1} a_{j+t} x^j\right] (x^t - c) + r(x). \ \square$$

Thus computing the remainder of an arbitrary $(2t - 1)$st-degree polynomial when divided by $x^t - c$ can be done in $O_A(t)$ steps, which is less than that required by any known algorithm for computing the remainder of an arbitrary $(2t - 1)$st-degree polynomial when divided by an arbitrary tth degree polynomial.

The complete FFT algorithm is given below.

Algorithm 7.1. Fast Fourier transform.

Input: A vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$, where $n = 2^k$ for some integer k.

Output. The vector $F(\mathbf{a}) = [b_0, b_1, \ldots, b_{n-1}]^T$, where $b_i = \Sigma_{j=0}^{n-1} a_j\omega^{ji}$ for $0 \leq i < n$.

Method. See Fig. 7.3. □

To modify Algorithm 7.1 to compute inverse transforms, we merely replace ω by ω^{-1} by changing the sign of the exponents of ω in lines 6 and 7. Then we divide $b_{\mathrm{rev}(l)}$ by n in line 8.

Example 7.2. Let $n = 8$ and thus $k = 3$. For $m = 2$ the loop of lines 3–7 is executed only for $l = 0$. At line 4, r_{03} is $\Sigma_{j=0}^{7} a_j x^j$. At line 5, s is 0. At lines 6 and 7,

$$r_{02} = (a_3 + a_7)x^3 + (a_2 + a_6)x^2 + (a_1 + a_5)x + (a_0 + a_4)$$

```
    begin
1.      let $r_{0k}$ be $\Sigma_{j=0}^{n-1} a_j x^j$;
        comment A polynomial is represented by its coefficients so no
          computation is involved at line 1. $r_{lm}$ will represent the re-
          mainder when $\Sigma_{j=0}^{n-1} a_j x^j$ is divided by $q_{lm}$;
2.      for $m \leftarrow k - 1$ step $-1$ until 0 do
3.          for $l \leftarrow 0$ step $2^{m+1}$ until $n - 1$ do
                begin
4.                  let $r_{l,m+1}$ be $\Sigma_{j=0}^{2^{m+1}-1} a_j x^j$;
                    comment We compute the remainders of lower de-
                      gree from the coefficients of $r_{l,m+1}$;
5.                  $s \leftarrow \mathrm{rev}(l/2^m)$:
6.                  $r_{lm}(x) \leftarrow \Sigma_{j=0}^{2^m-1} (a_j + \omega^s a_{j+2^m}) x^j$;
7.                  $r_{l+2^m,m}(x) \leftarrow \Sigma_{j=0}^{2^m-1} (a_j + \omega^{s+n/2} a_{j+2^m}) x^j$
                end;
8.      for $l \leftarrow 0$ until $n - 1$ do $b_{\mathrm{rev}(l)} \leftarrow r_{l0}$
    end
```

Fig. 7.3. The fast Fourier transform.

and

$$r_{42} = (a_3 + \omega^4 a_7)x^3 + (a_2 + \omega^4 a_6)x^2 + (a_1 + \omega^4 a_5)x + (a_0 + \omega^4 a_4).$$

For $m = 1$, l takes on the values 0 and 4. When $l = 0$, we have $s = 0$ on line 5. Thus by lines 6 and 7

$$r_{01} = (a_1 + a_3 + a_5 + a_7)x + (a_0 + a_2 + a_4 + a_6)$$

and

$$r_{21} = (a_1 + \omega^4 a_3 + a_5 + \omega^4 a_7)x + (a_0 + \omega^4 a_2 + a_4 + \omega^4 a_6).$$

When $l = 4$, we compute $s = 2$. We obtain by lines 6 and 7 and the formula for r_{42} above that

$$r_{41} = (a_1 + \omega^2 a_3 + \omega^4 a_5 + \omega^6 a_7)x + (a_0 + \omega^2 a_2 + \omega^4 a_4 + \omega^6 a_6),$$
$$r_{61} = (a_1 + \omega^6 a_3 + \omega^4 a_5 + \omega^2 a_7)x + (a_0 + \omega^6 a_2 + \omega^4 a_4 + \omega^2 a_6).$$

Finally, for $m = 0$, l takes on the values 0, 2, 4, and 6. With $l = 4$, for example, we have $s = 1$, and we compute from r_{41}:

$$r_{40} = a_0 + \omega a_1 + \omega^2 a_2 + \cdots + \omega^7 a_7.$$

On reaching the **for** loop of line 8, r_{l0} will always be a polynomial of degree 0, i.e., a constant. For example, when $l = 4$, **rev**(l) = 1, and r_{40} is assigned to b_1. This formula for b_1 agrees with the definition of b_1. □

We must now show that Algorithm 7.1 is correct.

Theorem 7.3. Algorithm 7.1 computes the discrete Fourier transform.

Proof. At line 6, $r_{lm} = r_{l,m+1}/q_{lm}$ and at line 7, $r_{l+2^m,m} = r_{l,m+1}/q_{l+2^m,m}$. Thus by use of Lemmas 7.2 and 7.3, it is simple to prove by induction on $k - m$ that r_{lm} is the remainder when $\Sigma_{j=0}^{n-1} a_j x^j$ is divided by q_{lm}. Then, for $m = 0$, Lemma 7.3 guarantees that the **for** loop of line 8 assigns the correct (constant) remainder to each of the b_i's. □

Theorem 7.4. Algorithm 7.1 requires $O_A(n \log n)$ time.

Proof. Lines 6 and 7 require $O_A(2^m)$ steps each time they are executed. For fixed m, the loop of lines 3–7 is iterated $n/2^{m+1}$ times for a total cost of $O_A(n)$, independent of m. The outer loop beginning at line 2 is executed $\log n$ times, for a total cost of $O_A(n \log n)$. The loop of line 8 actually requires no arithmetic. □

We have assumed in Theorem 7.4 that n was fixed. Thus powers of ω and the values of s and **rev**(l) computed from l on lines 5 and 8 can be precomputed and used as constants in a straight-line program. If it is desired that n be a parameter, we can still compute the powers of ω and store them in a table in $O(n)$ steps of a RAM. Moreover, even if $O(\log n)$ steps are spent computing s and rev(l) from l on lines 5 and 8, no more than $3n$ such calculations are done, so the entire algorithm is $O(n \log n)$ in time complexity on a RAM.

Corollary 1. We may compute $\mathbf{a} \circledast \mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are vectors of length n, in $O_A(n \log n)$ steps.

Proof. By Theorems 7.1, 7.3, and 7.4. □

Corollary 2. We may compute the positive and negative wrapped convolutions of $\mathbf{a}$ and $\mathbf{b}$ in $O_A(n \log n)$ steps.

Algorithm 7.1 for the FFT was presented to provide intuitive insight into the development of the algorithm. If one were to actually compute the FFT one would work only with the coefficients and simplify the algorithm somewhat. This is done in Algorithm 7.2.

Algorithm 7.2. Simplified FFT algorithm.

Input. A vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$, where $n = 2^k$ for integer k.

Output. The vector $F(\mathbf{a}) = [b_0, b_1, \ldots, b_{n-1}]^T$, where $b_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ for $0 \le i < n$.

Method. We use the program in Fig. 7.4. In this program we have used, for conceptual simplicity, a temporary array S to hold the results of the previous step. In practice, the computation can be done in place. □

```
      begin
1.        for $i \leftarrow 0$ until $2^k - 1$ do $R[i] \leftarrow a_i$;
2.        for $l \leftarrow 0$ until $k - 1$ do
              begin
3.                for $i \leftarrow 0$ until $2^k - 1$ do $S[i] \leftarrow R[i]$;
4.                for $i \leftarrow 0$ until $2^k - 1$ do
                      begin
5.                        let $[d_0 d_1 \cdots d_{k-1}]$ be the binary representation
                            of the integer $i$;
6.                        $R[[d_0 \cdots d_{k-1}]] \leftarrow$
                            $S[[d_0 \cdots d_{l-1} 0 d_{l+1} \cdots d_{k-1}]]$
                            $+ \omega^{[d_l d_{l-1} \cdots d_0 0 \cdots 0]} S[[d_0 \cdots d_{l-1} 1 d_{l+1} \cdots d_{k-1}]]$
                      end
              end;
7.        for $i \leftarrow 0$ until $2^k - 1$ do
              $b_{[d_0 \cdots d_{k-1}]} \leftarrow R[[d_{k-1} \cdots d_0]]$
      end
```

Fig. 7.4. Simplified FFT.

The first time line 3 is executed the coefficients of the polynomia $p(x) = \sum_{i=1}^{n-1} a_i x^i$ are stored in the array S. The first time line 6 is executed $p(x$ is divided by $x^{n/2} - 1$ and $x^{n/2} - \omega^{n/2}$. The remainders are

$$\sum_{i=0}^{n/2-1} (a_i + a_{i+n/2})x^i \qquad \text{and} \qquad \sum_{i=0}^{n/2-1} (a_i + \omega^{n/2} a_{i+n/2})x^i.$$

The coefficients of the two remainders are stored in the array S at the seconc execution of line 3. The coefficients of the first remainder occupy the first halı of S and the coefficients of the second remainder occupy the last half of S.

On the second execution of line 6 each of the two remainder polynomials is divided into two polynomials of the form $x^{n/4} - \omega^s$. This process results in four remainders each of degree $n/4 - 1$. The third execution of line 3 stores the coefficients of these four remainders in S, and so on. Line 7 rearranges the components of the answer into their proper order. This line is needed since the roots of unity were permuted in order to eliminate the cross product terms when computing products of the $x - \omega^i$'s. The process is partially illustrated for $n = 8$ in Fig. 7.5.

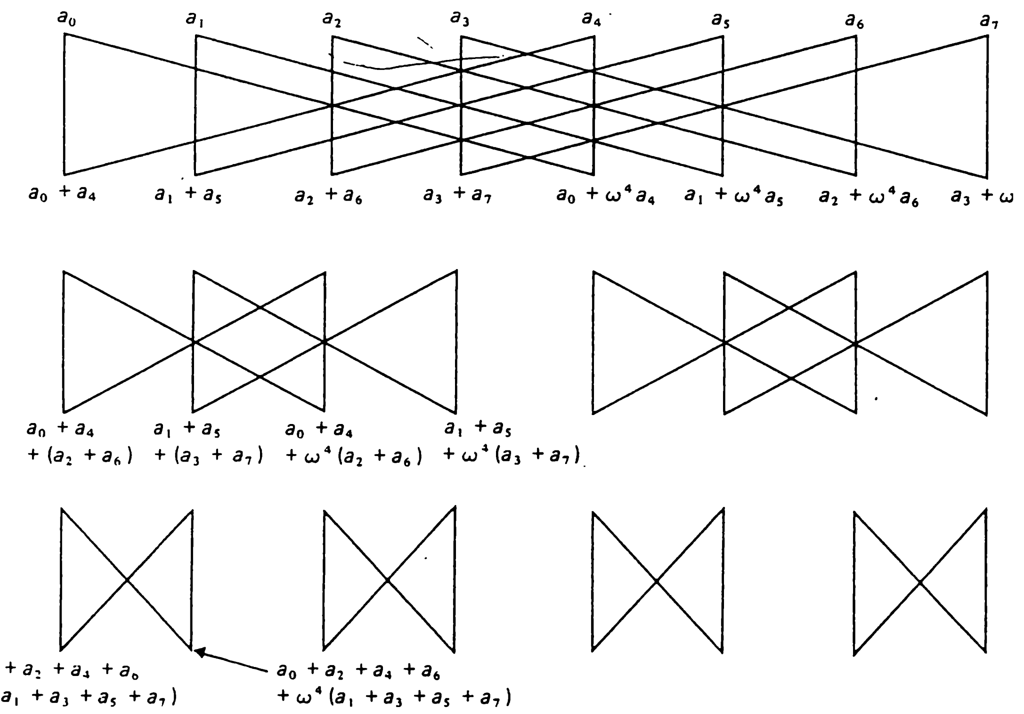

a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
$a_0 + a_4$
$a_1 + a_5$
$a_2 + a_6$
$a_3 + a_7$
$a_0 + \omega^4 a_4$
$a_1 + \omega^4 a_5$
$a_2 + \omega^4 a_6$
$a_3 + \omega$
$a_0 + a_4 + (a_2 + a_6)$
$a_1 + a_5 + (a_3 + a_7)$
$a_0 + a_4 + \omega^4 (a_2 + a_6)$
$a_1 + a_5 + \omega^4 (a_3 + a_7)$
$a_0 + a_2 + a_4 + a_6 + (a_1 + a_3 + a_5 + a_7)$
$a_0 + a_2 + a_4 + a_6 + \omega^4 (a_1 + a_3 + a_5 + a_7)$

Fig. 7.5 Illustration of the computation of FFT by Algorithm 7.2. The coefficients of certain remainder polynomials are omitted for lack of space.

.3 THE FFT USING BIT OPERATIONS

n applications where the Fourier transform is used to simplify the computation f a convolution, we often require an exact result. If we are working in the ring f real numbers, we must approximate real numbers with finite precision umbers, thus introducing errors. These errors can be avoided by performıg the computation in a finite field.† For example, to convolve $\mathbf{a} = a_0, a_1, a_2, 0, 0]^T$ and $\mathbf{b} = [b_0, b_1, b_2, 0, 0]^T$, we can let 2 play the role of the ıfth root of unity and perform computations modulo 31. Then transforming and **b**, performing the pairwise product, and computing the inverse transɔrm results in $\mathbf{a} \circledast \mathbf{b}$ modulo 31 exactly.‡ The difficulty with using a finite eld is in finding a suitable field with an nth root of unity. What we shall do intead is use the ring R_m of integers modulo m, selecting m so that R_m will have n nth root of unity ω.§ It is not immediately clear that given n, we can find ω nd m such that ω is an nth root of unity in the ring of integers modulo m. 1oreɔver, it would not do for m to be too large, since computations modulo m /ould become prohibitive. Fortunately, if n is a power of 2, there always xists a suitable m, where m is approximately 2^n. In particular we show Theorem 7.5) that when n and $\omega > 1$ are powers of 2, we can compute onvolutions in the ring of integers modulo $(\omega^{n/2} + 1)$ by a Fourier transform, omponentwise multiplication, and an inverse transform. First, we establish wo preliminary results in Lemmas 7.4 and 7.5. In these lemmas we assume ıat $R = (S, +, \cdot, 0, 1)$ is a commutative ring and $n = 2^k$, $k \geq 1$.

Lemma 7.4. For all $a \in S$,

$$\sum_{i=0}^{n-1} a^i = \prod_{i=0}^{k-1} (1 + a^{2^i}).$$

roof. The proof is by induction on k. The basis, $k = 1$, is trivial. Now, observe that

$$\sum_{i=0}^{n-1} a^i = (1 + a) \sum_{i=0}^{n/2-1} (a^2)^i. \tag{7.6}$$

y the induction hypothesis and substituting a^2 for a, we obtain

$$\sum_{i=0}^{n/2-1} (a^2)^i = \prod_{i=0}^{k-2} [1 + (a^2)^{2^i}] = \prod_{i=1}^{k-1} [1 + a^{2^i}]. \tag{7.7}$$

See Section 12.1 for a definition of "field."

Of course, we must be sure the elements of the answer are between 0 and 30, else e could not recover them. In general, we must choose our modulus sufficiently large ' recover the answer.

Until now, you have not been led astray if you thought of ω as the complex number $^{\pi i/n}$ and arithmetic as it is done in the field of complex numbers. From here on ough, ω must be thought of as an integer and all arithmetic must be done in a finite ıg of integers modulo m.

The induction hypothesis follows by substituting (7.7) into the right-hand sic of (7.6). □

Lemma 7.5. Let $m = \omega^{n/2} + 1$, where $\omega \in S$, $\omega \neq 0$. Then for $1 \leq p < n$ we have $\sum_{i=0}^{n-1} \omega^{ip} \equiv 0$ modulo m.

Proof. By Lemma 7.4 it suffices to show that $1 + \omega^{2^j p} \equiv 0$ modulo m for som j, $0 \leq j < k$. Let $p = 2^s p'$, where p' is odd. Surely $0 \leq s < k$. Choose such that $j + s = k - 1$. Then $1 + \omega^{2^j p} = 1 + \omega^{2^{k-1} p'} = 1 + (m - 1)^{p'}$. Bu $(m - 1) \equiv -1$ modulo m and p' is odd, so $(m - 1)^{p'} \equiv -1$ modulo m. I follows that $1 + \omega^{2^j p} \equiv 0$ modulo m for $j = k - 1 - s$. □

Theorem 7.5. Let n and ω be positive powers of 2 and let $m = \omega^{n/2} + 1$ Let R_m be the ring of integers modulo m. Then in R_m, n has a multiplicative inverse modulo m and ω is a principal nth root of unity.

Proof. Since n is a power of 2 and m is odd, it follows that m is relatively prime to n. Thus n has a multiplicative inverse modulo m.† Since $\omega \neq 1$ $\omega^n = \omega^{n/2}\omega^{n/2} \equiv (-1)(-1) = 1$ modulo $(\omega^{n/2} + 1)$. It then follows fron Lemma 7.5 that ω is a principal nth root of unity in R_m. □

The importance of Theorem 7.5 is that the convolution theorem is valid for the ring of integers modulo $2^{n/2} + 1$. If we wish to compute the convolution of two n-vectors with integer components, and the components of the convolution are in the range 0 to $2^{n/2}$, then we can be assured that an exact answer will be obtained. If the components of the convolution are not in the range 0 to $2^{n/2}$, then they are correct modulo $2^{n/2} + 1$.

We are almost ready to establish the number of bit operations needed to compute a convolution modulo m. First, however, we consider the number of bit operations used in calculating the residue of an integer modulo m, as this is an essential step in deducing the number of bit operations from the number of arithmetic operations modulo m.

Let $m = \omega^p + 1$ for some integer p. A generalization of "casting out 9's" is used to compute a modulo m. If a is written in base ω^p notation as a sequence of l blocks of p digits, then a modulo m can be calculated by alternately adding and subtracting the l blocks of p digits.

Lemma 7.6. Let $m = \omega^p + 1$ and let $a = \sum_{i=0}^{l-1} a_i \omega^{pi}$, where $0 \leq a_i < \omega^p$ for each i. Then $a \equiv \sum_{i=0}^{l-1} a_i (-1)^i$ modulo m.

Proof. Observe that $\omega^p \equiv -1$ modulo m. □

† This result is a basic theorem of number theory. In Section 8.8, we show that if a and b are relatively prime, there exist integers x and y such that $ax + by = 1$. Then $ax \equiv 1$ modulo b. Letting $b = m$ and a be n gives us our result.

Note that if l (the number of blocks) in Lemma 7.6 is fixed, then the computation of the residue a modulo m can be accomplished in $O_B(p \log \omega)$ bit operations.

Example 7.3. Let $n = 4$, $\omega = 2$, and $m = 2^2 + 1$. Thus $p = 2$ in Lemma 7.6. Consider $a = 101100$ in base 2. Here $a_0 = 00$, $a_1 = 11$, and $a_2 = 10$. We compute $a_0 - a_1 + a_2 = -1$ and then add m to find that $a \equiv 4$ modulo 5. Since a is 44, this result checks. □

Lemma 7.6 provides an efficient method of computing a modulo m. It plays an important role in the following theorem which gives an upper bound on the number of bit operations needed to compute the discrete Fourier transform and its inverse.

Theorem 7.6. Let ω and n be powers of 2 and $m = \omega^{n/2} + 1$. Let $[a_0, a_1, \ldots, a_{n-1}]^T$ be a vector with integer components, where $0 \le a_i < m$ for each i. Then the discrete Fourier transform of $[a_0, a_1, \ldots, a_{n-1}]^T$ and its inverse can be computed modulo m in $O_B(n^2 \log n \log \omega)$ steps.

Proof. Use Algorithm 7.1 or 7.2. In the inverse transform, substitute ω^{-1} for ω and multiply each result by n^{-1}. An integer modulo m can be represented by a string of $b = ((n/2) \log \omega) + 1$ bits. Since $m = 2^{b-1} + 1$, the residues modulo m can be represented by the bit strings $00 \ldots 0$ through $100 \ldots 0$.

Algorithm 7.1 requires integer addition modulo m and multiplication modulo m of an integer by a power of ω. These operations are performed $O(n \log n)$ times. Using Lemma 7.6, addition modulo m requires $O_B(b)$ steps where $b = ((n/2) \log \omega) + 1$. Multiplication by ω^p, $0 \le p < n$, is equivalent to a left shift of $p \log \omega$ places, since ω is a power of 2. The resulting integer has at most $3b - 2$ bits, and thus by Lemma 7.6 the shifting and computation of residues requires $O_B(b)$ steps. Thus the Fourier transform in the forward direction is of time complexity $O_B(bn \log n)$, i.e. $O_B(n^2 \log n \log \omega)$.

The inverse transform requires multiplication by ω^{-p} and by n^{-1}. Since $\omega^p\omega^{n-p} \equiv 1$ modulo m, we have $\omega^{n-p} \equiv \omega^{-p}$ modulo m. Thus the effect of multiplying by ω^{-p} can be achieved by multiplying by ω^{n-p}. The latter is a left shift of $(n - p)\log \omega$ places, and the resulting integer has at most $3b - 2$ bits. Again residues may be found in $O_B(b)$ steps by Lemma 7.6. Finally, we consider multiplication by n^{-1}. If $n = 2^k$, then we shift left $n \log \omega - k$ places, again leaving a number of at most $3b - 2$ bits, and compute the residue by Lemma 7.6. Thus the inverse transform also requires $O_B(n^2 \log n \log \omega)$ steps. □

Example 7.4. Let $\omega = 2$, $n = 4$, and $m = 5$. We shall take the Fourier transform of the vector $[a_0, a_1, a_2, a_3]^T$, where $a_i = i$. As $a_i < 5$ for each i, we

a_0	a_1	a_2	a_3
$a_0 + a_2$	$a_1 + a_3$	$a_0 + 4a_2$	$a_1 + 4a_3$
$a_0 + a_2 + (a_1 + a_3)$ $\equiv a_0 + a_1 + a_2 + a_3$ $\equiv b_0$	$a_0 + a_2 + 4(a_1 + a_3)$ $\equiv a_0 + 4a_1 + a_2 + 4a_3$ $\equiv b_2$	$a_0 + 4a_2 + 2(a_1 + 4a_3)$ $\equiv a_0 + 2a_1 + 4a_2 + 8a_3$ $\equiv b_1$	$a_0 + 4a_2 + 8(a_1 + 4a_3)$ $\equiv a_0 + 8a_1 + 4a_2 + 2$ $\equiv b_3$

Fig. 7.6. Computation of fast Fourier transform for $n = 4$.

may expect to recover this vector if we compute modulo m. We use three bits to represent numbers, but only 000, . . . , 100 will actually be used, except in temporary results.

We must, in Algorithm 7.1, compute the polynomial coefficients shown in Fig. 7.6, where 2 has been substituted for ω.

The actual values for Fig. 7.6 are:

$a_0 = 000$	$a_1 = 001$	$a_2 = 010$	$a_3 = 011$
$a_0 + a_2 = 010$	$a_1 + a_3 = 100$	$a_0 + 4a_2 = 011$	$a_1 + 4a_3 = 011$
$b_0 = 001$	$b_2 = 011$	$b_1 = 100$	$b_3 = 010$

The transform of $[0, 1, 2, 3]^T$ is thus $[1, 4, 3, 2]^T$ modulo 5. Consider the last entry, b_3, in the bottom row. It is computed from the last two entries in the middle-row by the following steps.

Take $a_1 + 4a_3$	011
Shift left three places (multiply by 8)	11000
Split into three blocks of 2	1 10 00
Take the sum of the first and third blocks minus the second	-1
Add $m = 5$	100
Add $a_0 + 4a_2 = 011$	111
Subtract m	010

To invert, we note that $2^{-1} \equiv 8$ modulo 5, $4^{-1} \equiv 4$ modulo 5, and $8^{-1} \equiv 2$ modulo 5. Thus the formulas for inverse transforms may be obtained from Fig. 7.6 by interchanging a_i and b_i and interchanging 2 and 8. The computation is thus:

$b_0 = 001$	$b_1 = 100$	$b_2 = 011$	$b_3 = 010$
$b_0 + b_2 = 100$	$b_1 + b_3 = 001$	$b_0 + 4b_2 = 011$	$b_1 + 4b_3 = 010$
$4a_0 = 000$	$4a_2 = 011$	$4a_1 = 100$	$4a_3 = 010$

Finally, we divide each answer by 4 (multiply by 4 since $4^{-1} \equiv 4$ modulo 5) and obtain $[0, 1, 2, 3]^T$ for $[a_0, a_1, a_2, a_3]^T$. □

Corollary to Theorem 7.6. Let $O_B(M(k))$ be the number of steps needed to compute the product of two k-bit integers. Let **a** and **b** be vectors of length n with integer components in the range 0 to ω^n, where n and ω are powers of 2. Then we may compute **a** ⊛ **b** or the positive or negative wrapped convolutions of **a** and **b** modulo $\omega^n + 1$ in

$$O_B(\text{MAX}[n^2 \log n \log \omega,\ nM(n \log \omega)])$$

time.

The first term in the function of the corollary to Theorem 7.6 is the time) take the transforms. The second is the cost of doing $2n$ multiplications f $(n \log \omega + 1)$-bit integers. The best value we can obtain for $M(k)$ is log k loglog k (see Section 7.5). At this value, the second term dominates ne first, so we require $O_B(n^2 \log n \operatorname{loglog} n \log \omega \operatorname{loglog} \omega \operatorname{logloglog} \omega)$ steps ɔ do this convolution.

.4 PRODUCTS OF POLYNOMIALS

'he problem of computing the product of two polynomials in a single variable , actually the same as computing the convolution of two sequences. That is

$$\left(\sum_{i=0}^{n-1} a_i x^i\right)\left(\sum_{j=0}^{n-1} b_j x^j\right) = \sum_{k=0}^{2n-2} c_k x^k, \qquad \text{where} \qquad c_k = \sum_{m=0}^{n-1} a_m b_{k-m}.$$

.s before, a_p and b_p are taken to be zero if $p < 0$ or $p \geq n$. Also recall that $_{2n-1}$ must be zero. We thus have the following additional corollaries to 'heorem 7.4.

Corollary 3 to Theorem 7.4. The coefficients in the product of two nth-degree polynomials can be computed in $O_A(n \log n)$ steps.

roof. From Corollary 1 to Theorem 7.4 and the above observation. □

Corollary 4 to Theorem 7.4. Suppose we can compute the product of two k-bit integers in $M(k)$ steps. Let

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \qquad \text{and} \qquad q(x) = \sum_{j=0}^{n-1} b_j x^j.$$

Suppose a_i and b_j are integers in the range 0 to $\omega^{n/2}/\sqrt{n}$ for all i and j, where n and ω are powers of 2. Then we may compute the coefficients of $p(x)q(x)$ in $O_B(\text{MAX}[n^2 \log n \log \omega,\ nM(n \log \omega)])$ steps.

roof. From Theorem 7.4 and the corollary to Theorem 7.6. □

Again note that the second term will dominate in Corollary 4.

In fact, Theorem 7.4 has the following interpretation. Suppose $p(x)$ and $x)$ are $(n-1)$st-degree polynomials. We may evaluate p and q at any ı − 1 or more points, say $c_0, c_1, \ldots$, and then multiply the values $p(c_j)q(c_j)$ ' obtain the values of pq at these points. A unique $(2n-2)$nd-degree ›lynomial can then be interpolated through these points. This $(2n-2)$nd-:gree polynomial will be the product $p(x)q(x)$.

When we apply the Fourier transform to convolutions (or equivalently polynomial multiplication), we are choosing $c_j = \omega^j$, where ω is a principa $2n$th root of unity. We take the Fourier transforms of p and q, i.e., evaluate them at the points $c_0, c_1, \ldots$ Next, we take the pairwise products of the transforms, i.e., multiply $p(c_j)$ by $q(c_j)$, to obtain the value of the product at c_j Then, we find pq by applying the inverse transform. Lemma 7.1 guarantees that the inverse is in fact a formula for interpolation. That is, we really do recover the polynomial from its values at the points $\omega^0, \omega^1, \ldots, \omega^{2n-1}$.

7.5 THE SCHÖNHAGE-STRASSEN INTEGER-MULTIPLICATION ALGORITHM

We now turn to an important application of the Convolution Theorem—a fast bitwise integer-multiplication algorithm. In Section 2.6 we saw how to multiply two n-bit integers in $O(n^{\log 3})$ steps by partitioning each binary integer into two $(n/2)$-bit integers. The method can be generalized by partitioning each integer into b blocks of l bits each. Expressions analogous to those in (2.4) are developed by treating the b blocks as coefficients of a polynomial. To find the coefficients of the product polynomial, we evaluate the polynomials at some convenient set of points, multiply sample values, and interpolate. By selecting the principal roots of unity as evaluation points, we can make use of the Fourier transform and the convolution theorem. By letting b be a function of n and using recursion we can multiply two n-bit numbers in $O_B(n \log n \operatorname{loglog} n)$ steps.

To simplify matters, we restrict n to be a power of 2. In the case where n is not a power of 2, add an appropriate number of leading zeros so that n is a power of 2 (this increases only the constant factor). Furthermore, we shall compute the product of two n-bit integers modulo $(2^n + 1)$. To obtain the exact product of two n-bit integers, we must introduce leading 0's and multiply $2n$-bit integers modulo $(2^{2n} + 1)$, thereby again increasing the time by a constant factor.

Let u and v be binary integers between 0 and 2^n which we are to multiply modulo $(2^n + 1)$. Observe that 2^n requires $n + 1$ bits for its binary representation. If either u or v equals 2^n it is represented by a special symbol, -1, and the multiplication is easily handled as a special case. If $u = 2^n$, then uv modulo $(2^n + 1)$ is obtained by computing $(2^n + 1 - v)$ modulo $(2^n + 1)$.

Suppose $n = 2^k$ and let $b = 2^{k/2}$ if k is even, else let $b = 2^{(k-1)/2}$. Let $l = n/b$. Observe that $l \geq b$ and that b divides l. The first step is to divide u and v into b blocks of l bits each. Thus

$$u = u_{b-1}2^{(b-1)l} + \cdots + u_1 2^l + u_0$$

and

$$v = v_{b-1}2^{(b-1)l} + \cdots + v_1 2^l + v_0.$$

The product of u and v is given by

$$uv = y_{2b-2}2^{(2b-2)l} + \cdots + y_1 2^l + y_0, \qquad (7.8)$$

where

$$y_i = \sum_{j=0}^{b-1} u_j v_{i-j}, \qquad 0 \le i < 2b.$$

(For $j < 0$ or $j > b - 1$ take $u_j = v_j = 0$. The term y_{2b-1} is 0 and is included for symmetry only.)

The product uv could be computed by using the Convolution Theorem. Multiplying the Fourier transforms would require $2b$ multiplications. By using a wrapped convolution, we can reduce the number of multiplications to b. This is the reason we compute uv modulo $2^n + 1$. Since $bl = n$, we have $2^{bl} \equiv -1$ modulo $(2^n + 1)$. Thus by (7.8) and Lemma 7.6, taking uv modulo $(2^n + 1)$ gives

$$uv \equiv (w_{b-1}2^{(b-1)l} + \cdots + w_1 2^l + w_0) \text{ modulo } (2^n + 1),$$

where $w_i = y_i - y_{b+i}$, $0 \le i < b$.

Since the product of two l-bit numbers must be less than 2^{2l} and since y_i and y_{b+i} are sums of $i + 1$ and $b - (i + 1)$ such products, respectively, $w_i = y_i - y_{b+i}$ must be in the range $-(b - 1 - i)2^{2l} < w_i < (i + 1)2^{2l}$. Thus there are at most $b2^{2l}$ possible values which w_i may assume. If we can compute the w_i's modulo $b2^{2l}$ we can compute uv modulo $(2^n + 1)$ in $O(b \log(b2^{2l}))$ additional steps by adding the b w_i's together with appropriate shifts.

To compute the w_i's modulo $b2^{2l}$ we compute the w_i's twice, once modulo b and once modulo $2^{2l} + 1$. Let w'_i be w_i modulo b and let w''_i be w_i modulo $(2^{2l} + 1)$. Since b is a power of 2 and $2^{2l} + 1$ is odd, b and $2^{2l} + 1$ are relatively prime. Thus the w_i's can be computed from the w'_i's and w''_i's by the formula

$$w_i = (2^{2l} + 1)((w'_i - w''_i) \text{ modulo } b) + w''_i \text{ [†]}$$

and w_i is between $(b - 1 - i)2^{2l}$ and $(i + 1)2^{2l}$. The work necessary to calculate w_i from w'_i and w''_i is $O(l + \log b)$ for each w_i, for a total of $O(bl + b \log b)$, or $O(n)$.

The w_i's are calculated modulo b by taking $u'_i = u_i$ modulo b and $v'_i = v_i$ modulo b and forming two $(3b \log b)$-bit numbers $\hat{u}$ and $\hat{v}$ as shown in Fig. 7.7. Taking the product $\hat{u}\hat{v}$ by the algorithm of Section 2.6 (p. 62) requires at most $O((3b \log b)^{1.6})$, i.e., less than $O(n)$ steps. We see $\hat{u}\hat{v} = \sum_{i=0}^{2b-1} y'_i 2^{(3 \log b)i}$ where $y'_i = \sum_{j=0}^{2b-1} u'_j v'_{i-j}$. Note that $y'_i < 2^{3 \log b}$, so the y'_i's can be easily recovered from the product $\hat{u}\hat{v}$. The values of the w_i's modulo b can then be found by computing $(y'_i - y'_{b+i})$ modulo b.

[†] If for p_1 and p_2 relatively prime, $w \equiv q_1$ modulo p_1, $w \equiv q_2$ modulo p_2, and $0 \le w < p_1 p_2$, then $w = p_2(p_2^{-1}$ modulo $p_1)(q_1 - q_2$ modulo $p_1) + q_2$. Let $p_1 = b$ and $p_2 = 2^{2l} + 1$. Since b is a power of 2 and $b \le 2^{2l}$, b divides 2^{2l} and thus the multiplicative inverse of $2^{2l} + 1$ modulo b is 1.

$$\hat{u} = u'_{b-1}\,00\ldots0\,u'_{b-2}\,00\ldots0\,u'_{b-3}\cdots00\ldots0\,u'_0$$

$$\hat{v} = v'_{b-1}\,00\ldots0\,v'_{b-2}\,00\ldots0\,v'_{b-3}\cdots00\ldots0\,v'_0$$

Fig. 7.7. Illustration of composite numbers used in computing w_i modulo b. There are 2 log b zeros in each block of zeros.

Having computed the w_i's modulo b, we compute the w_i's modul $2^{2l}+1$ by means of a wrapped convolution. This involves taking the Fourie transform, pairwise multiplication, and an inverse transform. Let $\omega = 2^{4l/b}$ and $m = 2^{2l}+1$. By Theorem 7.5, ω and b have multiplicative inverse modulo m, and ω is a principal bth root of unity. Thus the negative wrappe convolution of $[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]$ and $[v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]$, wher $\psi = 2^{2l/b}$ (ψ is a $2b$th root of unity), is

$$[(y_0 - y_b), \psi(y_1 - y_{b+1}), \ldots, \psi^{b-1}(y_{b-1} - y_{2b-1})] \text{ modulo } 2^{2l}+1,$$

where $y_i = \sum_{j=0}^{b-1} u_j v_{i-j}$ for $0 \le i \le 2b-1$. The w_i's modulo $2^{2l}+1$ can b€ obtained by an appropriate shift. The complete algorithm is summarize below.

Algorithm 7.3. Schönhage–Strassen integer-multiplication algorithm.

Input. Two n-bit integers, u and v, where $n = 2^k$.

Output. The $(n+1)$-bit product of u and v modulo 2^n+1.

Method. If n is small, multiply u and v modulo 2^n+1 by your favorite algorithm. For large n let $b = 2^{k/2}$ if k is even, else let $b = 2^{(k-1)/2}$. Let $l = n/b$. Express $u = \sum_{i=0}^{b-1} u_i 2^{li}$ and $v = \sum_{i=0}^{b-1} v_i 2^{li}$, where u_i and v_i are integers between 0 and $2^l - 1$ (that is, the u_i's are blocks of l bits of u and similarly the v_i's are blocks of l bits of v).

1. Compute the Fourier transform, modulo $2^{2l}+1$, of

   $$[u_0, \psi u_1, \ldots, \psi^{b-1}u_{b-1}]^T \quad\text{and}\quad [v_0, \psi v_1, \ldots, \psi^{b-1}v_{b-1}]^T$$

   with $\psi = 2^{2l/b}$ using ψ^2 as the principal bth root of unity.
2. Compute the pairwise product of the Fourier transforms computed in step 1, modulo $2^{2l}+1$, using Algorithm 7.3 recursively to compute each pairwise product. (The situation in which one of the numbers is 2^{2l} is handled as an easy special case.)
3. Compute the inverse Fourier transform modulo $2^{2l}+1$ of the vector of pairwise products from step 2. The result of this computation will be $[w_0, \psi w_1, \ldots, \psi^{b-1}w_{b-1}]^T$ modulo $2^{2l}+1$, where w_i is the ith term of the negative wrapped convolution of $[u_0, u_1, \ldots, u_{b-1}]^T$ and $[v_0, v_1, \ldots,$

$v_{b-1}]^T$. Compute $w_i'' = w_i$ modulo $2^{2l}+1$ by multiplying $\psi^i w_i$ by ψ^{-i} modulo $2^{2l}+1$.

4. Compute $w_i' = w_i$ mod b as follows.
   a) Let $u_i' = u_i$ modulo b and let $v_i' = v_i$ modulo b for $0 \leq i < b$.
   b) Construct the numbers $\hat{u}$ and $\hat{v}$ by stringing the u_i''s and v_i''s together with 2 log b intervening 0's. That is, $\hat{u} = \sum_{i=0}^{b-1} u_i' 2^{(3 \log b)i}$ and $v = \sum_{i=0}^{b-1} v_i' 2^{(3 \log b)i}$.
   c) Compute the product $\hat{u}\hat{v}$ using the algorithm of Section 2.6 (p. 62).
   d) The product $\hat{u}\hat{v}$ is $\sum_{i=0}^{2b-1} y_i' 2^{(3 \log b)i}$ where $y_i' = \sum_{j=0}^{2b-1} u_j' v_{i-j}'$. The w_i's modulo b can be recovered from this product by evaluating $w_i' = (y_i' - y_{b+i}')$ modulo b, for $0 \leq i < b$.
5. Compute the w_i's exactly using the formula
$$w_i = (2^{2l}+1)((w_i' - w_i'') \text{ modulo } b) + w_i''$$
and w_i is between $(b-1-i)2^{2l}$ and $(i+1)2^{2l}$.
6. Compute $\sum_{j=0}^{b-1} w_i 2^{li}$ modulo (2^n+1). This is the desired result. □

Theorem 7.7. Algorithm 7.3 computes uv modulo (2^n+1).

Proof. By Theorem 7.2, steps 1 through 3 of Algorithm 7.3 correctly evaluate the w_i's modulo $2^{2l}+1$. We leave as an exercise the fact that step 4 computes the w_i's modulo b and that step 5 computes the w_i's modulo $b(2^{2l}+1)$, i.e., the exact value of each w_i. □

Theorem 7.8. The execution time of Algorithm 7.3 is
$$O_B(n \log n \operatorname{loglog} n)$$
steps.

Proof. By the corollary to Theorem 7.6, steps 1 through 3 require time $O_B[bl \log b + bM(2l)]$, where $M(m)$ is the time to multiply two m-bit integers by a recursive application of the algorithm. In step 4, we construct $\hat{u}$ and $\hat{v}$ of length $3b \log b$ and multiply them in $O_B[(3b \log b)^{1.59}]$ steps. For sufficiently large b, $(3b \log b)^{1.59} < b^2$ and hence the time for step 4 can be ignored in view of the $O_B(b^2 \log b)$ term introduced for steps 1 through 3. Steps 5 and 6 are both $O(n)$ and can also be ignored.

Since $n = bl$ and b is at most $\sqrt{n}$, we obtain the recurrence
$$M(n) \leq cn \log n + bM(2l) \tag{7.9}$$
for some constant c and sufficiently large n. Let $M'(n) = M(n)/n$. Then (7.9) becomes
$$M'(n) \leq c \log n + 2M'(2l). \tag{7.10}$$
Since $l \leq 2\sqrt{n}$,
$$M'(n) \leq c \log n + 2M'(4\sqrt{n}), \tag{7.11}$$
which implies $M'(n) \leq c' \log n \operatorname{loglog} n$ for suitable c'. To see this, substi-

tute c' $\log(4\sqrt{n})\text{loglog}\, 4\sqrt{n}$ for $M'(4\sqrt{n})$ in (7.11). Straightforward manipulation yields

$$M'(n) \leq c \log n + 4c' \log(2 + \tfrac{1}{2} \log n) + c' \log n \log(2 + \tfrac{1}{2} \log n).$$

For large n, $2 + \frac{1}{2} \log n \leq \frac{3}{4} \log n$. Thus

$$M'(n) \leq c \log n + 4c' \log \tfrac{3}{4} + 4c' \text{ loglog } n + \\ c' \log \tfrac{3}{4} \log n + c' \log n \text{ loglog } n. \qquad (7.1$$

For large n and sufficiently large c', the first four terms are dominated b the fourth which is negative. Thus $M'(n) \leq c' \log n$ loglog n. From this w conclude that $M(n) \leq c'n \log n$ loglog n. □

EXERCISES

7.1 What are the principal nth roots of unity for $n = 3, 4, 5$ in the ring of comple numbers?

7.2 Show how Algorithm 7.2 can be implemented without the temporary array .

7.3 Compute the discrete Fourier transform of the following sequences with respe to the ring of complex numbers.

a) [0, 1, 2, 3]

b) [1, 2, 0, 2, 0, 0, 0, 1]

7.4 Generalize the fast Fourier transform algorithm to the case where n is not power of 2.

Definition. The triple of integers (ω, n, m) is said to be *admissible* if ω and have multiplicative inverses and ω is a principal nth root of unity in the ring integers modulo m.

7.5 Which of the following are admissible triples?

a) (3, 4, 5)

b) (2, 6, 21)

c) (2, 6, 7)

7.6 Show that if (ω, n, m) is an admissible triple, then $\omega^n \equiv 1$ modulo m and $\omega^p \not\equiv$ if $1 \leq p < n$.

***7.7** Show that if n is a power of 2, $2^n \equiv 1$ modulo m, and $2^p - 1$ is relatively prime to for $1 \leq p < n$, then $(2, n, m)$ is an admissible triple.

****7.8** Show that if m is a prime and ω is arbitrary, then there is some n such tha (ω, n, m) is an admissible triple.

***7.9** Show that if a and b are relatively prime then $ac \equiv 1$ modulo b for some c, an conversely. Prove that c is unique modulo b.

7.10 Find $(10101110011110)_2$ modulo $2^5 + 1$.

7.11 Let t be an integer in base 10. Show that adding the digits of t together, the adding the digits of the result together, and so on, until a single digit remains results in t modulo 9.

7.12 Compute the convolution of the sequences [1, 2, 3, 4] and [4, 3, 2, 1] modulo 17 by the Convolution Theorem using the admissible triple (2, 8, 17).

7.13 Compute the product of the binary numbers $(1011011)_2$ and $(10001111)_2$ using Algorithm 7.3.

7.14 Show that the result of taking the square root of a number $n = 2^{2^k}$, loglog n times is 2.

***7.15** Develop a fast integer-multiplication algorithm using a convolution rather than a wrapped convolution. What is the asymptotic running time of your algorithm?

***7.16** The *cyclic difference* of sequence $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]^T$, denoted $\Delta\mathbf{a}$, is the sequence $[a_0 - a_{n-1}, a_1 - a_0, a_2 - a_1, \ldots, a_{n-1} - a_{n-2}]^T$. Let

$$F(\mathbf{a}) = [a_0', a_1', \ldots, a_{n-1}']^T.$$

Show that $F(\Delta\mathbf{a}) = [0, a_1'(1 - \omega), a_2'(1 - \omega^2), \ldots, a_{n-1}'(1 - \omega^{n-1})]$, where ω is the nth root of unity.

***7.17** Use Exercise 7.16 to show that if $X(n) - X(n - 1) = n$ and $X(0) = 0$, then $X(n) = n(n + 1)/2$.

***7.18** A *circulant* is a matrix in which each row is a circular shift one space to the right of the row above it. For example,

$$\begin{bmatrix} a & b & c \\ c & a & b \\ b & c & a \end{bmatrix}$$

is a 3×3 circulant. Show that computing the discrete Fourier transform of a vector of length n, where n is a prime, is equivalent to multiplying by an $(n - 1) \times (n - 1)$ circulant.

***7.19** Show that the finite Fourier transform over a finite field, for n a prime, can be calculated in $O_A(n \log n)$ steps.

***7.20** Consider representing a polynomial by the values of each of its derivatives at a point. Is the transformation from the coefficients to the values of the derivatives linear?

***7.21** Give a "physical" explanation of the wrapped convolution in terms of polynomial operations.

***7.22** Use the FFT to give an $O(n \log n)$ algorithm for multiplying a Toeplitz matrix times a vector. Compare this solution to that of Exercise 6.26(b).

Research Problem

7.23 Find faster algorithms for integer multiplication or discrete Fourier transforms. Alternatively, show that the Schönhage–Strassen algorithm or the FFT is the best one can do under some restricted model. Take note of Paterson, Fischer, and Meyer [1974], which shows that, under certain restrictions, $O_B((n \log n)/(\text{loglog}\ n))$ time is necessary for bitwise integer multiplication. Similarly, Morgenstern [1973] shows that under certain restrictions, $O_A(n \log n)$ is required for the discrete Fourier transform.

BIBLIOGRAPHIC NOTES

Cooley, Lewis, and Welch [1967] trace the origins of the fast Fourier transform to Runge and König [1924]. The technique has been used by Danielson and Lanczos [1942] and Good [1958, 1960]. The fundamental paper by Cooley and Tukey [1965] made clear the nature of the technique. The development of the FFT as a polynomial division problem, the approach used here, is due to Fiduccia [1972]. Because of its importance in computing, a good deal of attention has been paid to the efficient implementation of the algorithm. For example, see Gentleman and Sande [1966] and many articles from Rabiner and Rader [1972]. The integer-multiplication algorithm is from Schönhage and Strassen [1971]. Exercises 7.18 and 7.19 are from Rader [1968].

INTEGER AND POLYNOMIAL ARITHMETIC

CHAPTER 8

A good reason for treating integer and polynomial arithmetic together is th many of the algorithms for manipulating integers and univariate polynomi are essentially identical. This is true not only for operations like multiplic tion and division but also for more sophisticated operations. For examp finding the residue of an integer modulo a second integer is equivalent evaluating a polynomial at a point. Representing an integer by its residues equivalent to representing a polynomial by its values at several points. Reco structing an integer from residues ("Chinese remaindering") is equivalent interpolating a polynomial.

In this chapter, we shall show that certain integer and polynomial oper tions, such as division and squaring, require the same order of time as mu tiplication. Other operations, such as the residue operations mentione above, or the calculation of greatest common divisors, are shown to require a most a factor of log n more time than multiplication, where n is the length c the binary integer or degree of the polynomial. Our strategy will be to alte nate results for integers with the corresponding results for polynomial usually proving the results for only one and leaving the other as an exercise As in the other chapters, our emphasis is on algorithms that are asymptoticall the most efficient known.

At the end of the chapter we briefly discuss some distinctions between a model for polynomials that assumes most coefficients are nonzero (the dense model) and one that assumes most coefficients are zero (the sparse model) The sparse model is particularly useful when dealing with polynomials involving many variables (a case we do not discuss).

8.1 THE SIMILARITY BETWEEN INTEGERS AND POLYNOMIALS

The most obvious similarity between a nonnegative integer and a polynomial in one variable is that each can be represented as a finite power series $\Sigma_{i=0}^{n-1} a_i x^i$. In the integer case, the a_i's can be chosen from the set $\{0, 1\}$ with $x = 2$. In the polynomial case, the a_i's can be chosen from some coefficient set† with x the indeterminate.

There is a natural "size" measure, which is essentially the length of the power series representing the integer or polynomial. In the binary integer case the size is the number of bits needed to represent the integer; in the polynomial case, the size is the number of coefficients. Thus we make the following definition.

† In what follows, you may regard the coefficient set as the field (see Section 12.1) of real numbers, although the results will apply to any field of coefficients, with computational complexity measured in terms of the number of operations in the coefficient field. Thus we do not consider the size of coefficients in our model.

Definition. If i is a nonnegative integer, then $\mathrm{SIZE}(i) = \lfloor \log i \rfloor + 1$. If $p(x)$ is a polynomial, then $\mathrm{SIZE}(p) = \mathrm{DEG}(p) + 1$, where $\mathrm{DEG}(p)$ is the degree of p, i.e., the highest power of x with a nonzero coefficient.

With integers and polynomials we can perform approximate division. If a and b are two integers, with $b \neq 0$, then there is a unique pair of integers q and r such that

1. $a = bq + r$, and
2. $r < b$

where q and r are the quotient and remainder when a is divided by b.

Similarly, if a and b are polynomials, with b not equal to a constant, then unique polynomials q and r can be found to satisfy

1. $a = bq + r$, and
2. $\mathrm{DEG}(r) < \mathrm{DEG}(b)$.

Another similarity between integers and polynomials is that they each have surprisingly fast multiplication algorithms. In the previous chapter we showed that two nth-degree polynomials with real coefficients can be multiplied in $O_A(n \log n)$ time by use of the FFT. The arithmetic operation measure of complexity is reasonable, since in practice we would represent polynomials by their coefficients to fixed precision and implement the various polynomial operations by arithmetic operations on the coefficients.

If we use the Schönhage-Strassen algorithm of Section 7.5, we can multiply two n-bit integers in $O_B(n \log n \operatorname{loglog} n)$ time. We claim that for integers, bit operations are the only measure of interest. There are essentially two situations in which we would not consider multiplication of integers to be primitive. The first is in designing multiplication hardware. The number of bit operations reflects the number of elements needed in a multiplication circuit. The second application is in designing fixed-point arbitrary precision algorithms for fixed-word-length computers. There, the number of bit operations is related to the number of machine instructions needed to do n-precision multiplication.

Thus the results for polynomial and integer arithmetic will appear quite similar when the two different measures of complexity (arithmetic and bit) are used. Moreover, the two measures are analogous in the sense that bit operations are coefficient operations for the power series representing integers, just as arithmetic operations are coefficient operations for polynomials.

8.2 INTEGER MULTIPLICATION AND DIVISION

We shall show that the time, in bit operations, to do integer multiplication is, to within a constant factor, the same as the time to do integer division and is similarly related to the operations of squaring and taking reciprocals. In this section we shall use four symbols with the following meanings:

Symbol	Meaning
$M(n)$	Time to multiply two integers of size n
$D(n)$	Time to divide an integer of size at most $2n$ by an integer of size n
$S(n)$	Time to square an integer of size n
$R(n)$	Time to compute the reciprocal of an integer of size n

In each case the time is measured in bit operations. We shall also make the reasonable assumption that $M(n)$ satisfies the condition

$$a^2M(n) \geq M(an) \geq aM(n)$$

for $a \geq 1$. We assume the other three functions also share this property.

We first show that the reciprocal of an n-bit integer i can be calculated in essentially the time to multiply two n-bit numbers. Since $1/i$ is not an integer for $i > 1$, what we really mean by the "reciprocal" of i is the n-significant-bit approximation to the fraction $1/i$. Since scaling (shifts of the binary point) is assumed to cost nothing, we may equivalently say that the "reciprocal" of i is the quotient of 2^{2n-1} divided by i. For the remainder of this section we use the term reciprocal with this meaning.

First let us consider finding a sequence of approximations to the number $A = 1/P$, where P is an integer. Let A_i be the ith approximation to $1/P$. Then the exact value of A can be expressed as

$$A = A_i + (1/P)(1 - A_iP). \tag{8.1}$$

If we approximate the value of $1/P$ by A_i in (8.1) we obtain

$$A_{i+1} = A_i + A_i(1 - A_iP) = 2A_i - A_i^2P, \tag{8.2}$$

which can be used as an iteration formula for finding the $(i + 1)$st approximation of A in terms of the ith approximation. Note that if $A_iP = 1 - S$, then

$$A_{i+1}P = 2A_iP - A_i^2P^2 = 2(1 - S) - (1 - S)^2 = 1 - S^2.$$

This shows that the iteration formula (8.2) converges quadratically. If $S \leq \frac{1}{2}$, then the number of correct bits is doubled each iteration.

Since P presumably has many bits, it is unnecessary to use all bits of P for the earlier approximations, because only the leading bits of P affect those bits of A_{i+1} which are correct. Moreover, if the first k bits of A_i to the right of the decimal point are correct, then we can obtain A_{i+1} to $2k$ bits using the formula (8.2). That is, we compute $A_{i+1} = 2A_i - A_i^2P$, with $2A_i$ truncated to k places to the right of the decimal point and A_i^2P computed by approximating P to $2k$ places and then truncating the product to $2k$ places to the right of the decimal point. This last approximation may of course affect the convergence, since we previously assumed that P was exact.

We shall use these ideas in the next algorithm. There we actually compute the quotient $\lfloor 2^{2n-1}/P \rfloor$, where $P = p_1p_2 \cdots p_n$ is an n-bit integer with $p_1 = 1$. The method is essentially that of Eq. (8.2), with scaling introduced to allow us to deal exclusively with integers. Thus no maintenance of the decimal point is necessary.

Algorithm 8.1. Integer reciprocals.

Input. An n-bit integer $P = [p_1p_2 \cdots p_n]$, with $p_1 = 1$. For convenience, we assume n is a power of 2 and $[x]$ is the integer denoted by the bit string x (e.g., $[110] = 6$).

Output. The integer $A = [a_0a_1 \cdots a_n]$ such that $A = \lfloor 2^{2n-1}/P \rfloor$.

Method. We call RECIPROCAL($[p_1p_2 \cdots p_n]$), where RECIPROCAL is the recursive procedure in Fig. 8.1. It computes an approximation to $\lfloor 2^{2k-1}/[p_1p_2 \cdots p_k] \rfloor$ for any k which is a power of 2. Observe that the result is normally a k-bit integer except when P is a power of 2, in which case the result is a $(k + 1)$-bit integer.

Given k bits of the reciprocal of $[p_1p_2 \cdots p_k]$, lines 2–4 compute $2k - 3$ bits of the reciprocal of $[p_1p_2 \cdots p_{2k}]$. Lines 5–7 correct the last three bits. In practice, one would skip lines 5–7 and obtain the desired accuracy by an extra application of Eq. (8.2) at the end. We have chosen to include the loop of lines 5–7 to simplify both the understanding of the algorithm and the proof that the algorithm does indeed work. □

Example 8.1. Let us compute $2^{15}/153$. Here,

$$n = 8 \quad \text{and} \quad 153 = [p_1p_2 \cdots p_8] = [10011001].$$

We call

RECIPROCAL([10011001]),

which in turn calls RECIPROCAL recursively with arguments [1001], [10], and [1]. At line 1, we find RECIPROCAL([1]) = [10] and return to RECIPROCAL([10]) at line 2, where we set $[c_0c_1] \leftarrow [10]$. Then, at line 3, we compute $[d_1 \cdots d_4] \leftarrow [10] * 2^3 - [10]^2 * [10] = [1000]$. Next, we set $[a_0a_1a_2]$ to [100] at line 4. No changes occur in the loop of lines 5–7. We return to RECIPROCAL([1001]) with [100] as an approximation to $2^3/[10]$.

Returning to line 2 with $k = 4$, we have $[c_0c_1c_2] = [100]$. Then $[d_1 \cdots d_8] = [01110000]$ and $[a_0 \cdots a_4] = [01110]$. Again, there are no changes in the loop of lines 5–7. We return to RECIPROCAL([10011001])

```
   procedure RECIPROCAL($[p_1p_2 \cdots p_k]$):
1. if $k = 1$ then return [10]
   else
       begin
2.         $[c_0c_1 \cdots c_{k/2}] \leftarrow$ RECIPROCAL($[p_1p_2 \cdots p_{k/2}]$);
3.         $[d_1d_2 \cdots d_{2k}] \leftarrow [c_0c_1 \cdots c_{k/2}] * 2^{3k/2} -$
               $[c_0c_1 \cdots c_{k/2}]^2 * [p_1p_2 \cdots p_k]$;
           comment Although the right-hand side of line 3 appears to
               produce a $(2k + 1)$-bit number, the leading $[(2k + 1)$st]
               bit is always zero;
4.         $[a_0a_1 \cdots a_k] \leftarrow [d_1d_2 \cdots d_{k+1}]$;
           comment $[a_0a_1 \cdots a_k]$ is a good approximation to
               $2^{2k-1}/[p_1p_2 \cdots p_k]$. The following loop improves the
               approximation by adding to the last three places if nec-
               essary;
5.         for $i \leftarrow 2$ step $-1$ until 0 do
6.             if $([a_0a_1 \cdots a_k] + 2^i) * [p_1p_2 \cdots p_k] \le 2^{2k-1}$ then
7.                 $[a_0a_1 \cdots a_k] \leftarrow [a_0a_1 \cdots a_k] + 2^i$;
8.         return $[a_0a_1 \cdots a_k]$
       end
```

Fig. 8.1. Procedure to compute integer reciprocals.

at line 2 with $[c_0 \cdots c_4] = [01110]$. Then

$$[d_1 \cdots d_{16}] = [0110101011011100].$$

Thus at line 4, $[a_0 \cdots a_8] = [011010101]$. At line 6, we find that $[011010101] * [10011001]$, which is $213 * 153$ in decimal, equals 32589, while $2^{15} = 32768$. Thus in the loop of lines 5–7, 1 is added to 213, yielding answer 214, or [011010110] in binary. □

Theorem 8.1. Algorithm 8.1 finds $[a_0a_1 \cdots a_k]$ such that

$$[a_0a_1 \cdots a_k] * [p_1p_2 \cdots p_k] = 2^{2k-1} - S$$

and $0 \le S < [p_1p_2 \cdots p_k]$.

Proof. The proof is by induction on k. The basis, $k = 1$, is trivial by line 1. For the inductive step, let $C = [c_0c_1 \cdots c_{k/2}]$, $P_1 = [p_1p_2 \cdots p_{k/2}]$, and $P_2 = [p_{k/2+1}p_{k/2+2} \cdots p_k]$. Then $P = [p_1p_2 \cdots p_k] = P_12^{k/2} + P_2$. By the induction hypothesis

$$CP_1 = 2^{k-1} - S,$$

where $0 \le S < P_1$. By line 3, $D = [d_1d_2 \cdots d_{2k}]$ is given by

$$D = C2^{3k/2} - C^2(P_12^{k/2} + P_2). \tag{8.3}$$

Since $p_1 = 1$, $P_1 \geq 2^{k/2-1}$ and thus $C \leq 2^{k/2}$. It follows that $D < 2^{2k}$ and hence the $2k$ bits used to represent D are sufficient.

Consider the product $PD = (P_1 2^{k/2} + P_2)D$, which by (8.3) is:

$$PD = CP_1 2^{2k} + CP_2 2^{3k/2} - (CP_1 2^{k/2} + CP_2)^2. \tag{8.4}$$

By substituting $2^{k-1} - S$ for CP_1 in (8.4) and performing some algebraic simplification, we get:

$$PD = 2^{3k-2} - (S2^{k/2} - CP_2)^2. \tag{8.5}$$

Dividing (8.5) by 2^{k-1}, we have

$$2^{2k-1} = \frac{PD}{2^{k-1}} + T, \tag{8.6}$$

where $T = (S2^{k/2} - CP_2)^2 2^{-(k-1)}$. By the induction hypothesis and the fact that $P_1 < 2^{k/2}$, we have $S < 2^{k/2}$. Since $C \leq 2^{k/2}$ and $P_2 < 2^{k/2}$, we have $0 \leq T < 2^{k+1}$.

At line 4, $A = [a_0 a_1 \cdots a_k] = \lfloor D/2^{k-1} \rfloor$. Now

$$P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor > P\left(\frac{D}{2^{k-1}} - 1\right),$$

so from (8.6) we have

$$2^{2k-1} \geq \frac{PD}{2^{k-1}} \geq P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor > \frac{PD}{2^{k-1}} - P = 2^{2k-1} - T - P > 2^{2k-1} - 2^{k+1} - 2^k.$$

Thus

$$P\left\lfloor \frac{D}{2^{k-1}} \right\rfloor = 2^{2k-1} - S',$$

where $0 \leq S' < 2^{k+1} + 2^k$. Since $P \geq 2^{k-1}$, it follows that by adding at most 6 to $\lfloor D/2^{k-1} \rfloor$ we obtain the number which satisfies the inductive hypothesis for k. Since this job is done by lines 5–7, the induction step follows. □

Theorem 8.2. There is a constant c such that $R(n) \leq cM(n)$.

Proof. It suffices to show that Algorithm 8.1 works in time $O_B(M(n))$. Line 2 requires $R(k/2)$ bit operations. Line 3 consists of a squaring and multiplication, requiring $M(k/2 + 1)$ and $M(k + 2)$ time, respectively, plus a subtraction requiring $O_B(k)$ time. Note that multiplication by powers of 2 does not require any bit operations; the bits of the multiplicand are simply regarded as occupying new positions, i.e., as if they had been shifted. By our assumption on M, $M(k/2 + 1) \leq \frac{1}{2}M(k + 2)$. Furthermore, $M(k + 2) - M(k)$ is $O_B(k)$ (see Section 2.6, for example) and thus line 3 is bounded by $\frac{3}{2}M(k) + c'k$ for some constant c'. Line 4 is clearly $O_B(k)$.

It appears that the loop of lines 5–7 requires three multiplications, but the calculation can be done by one multiplication, $[a_0 a_1 \cdots a_k] * [p_1 p_2 \cdots p_k]$, and some additions and subtractions of at most $2k$-bit integers. Thus line 5–7 are bounded by $M(k) + c''(k)$ for some constant c''. Putting all cost together, we have

$$R(k) \le R\left(\frac{k}{2}\right) + \tfrac{5}{2}M(k) + c_1 k \qquad (8.7$$

for some constant c_1.

We claim that there is a constant c such that $R(k) \le cM(k)$. We can choose c so that $c \ge R(1)/M(1)$ and $c \ge 5 + 2c_1$. We verify the claim by induction on k. The basis, $k = 1$, is immediate. The inductive step follows from (8.7), since

$$R(k) \le cM\left(\frac{k}{2}\right) + \tfrac{5}{2}M(k) + c_1 k. \qquad (8.8)$$

As $M(k/2) \le \frac{1}{2}M(k)$ follows from our assumption about M, and $k \le M(k)$ is obvious, we may rewrite (8.8) as

$$R(k) \le \left(\frac{c}{2} + \frac{5}{2} + c_1\right)M(k). \qquad (8.9)$$

Since $c \ge 5 + 2c_1$, (8.9) implies $R(k) \le cM(k)$. □

It should be evident that Algorithm 8.2 can be used to compute $1/P$ to n significant bits, if P has that number of bits, no matter where the binary point is. For example, if $\frac{1}{2} < P < 1$, and P has n bits, then by scaling in the obvious way, we can produce $1/P$ as 1, followed by $n - 1$ bits in the fractional part.

We next show that $S(n)$, the time needed to square an integer of size n, is of no greater magnitude than $R(n)$, the time to take the reciprocal of an integer of size n. The technique involves the identity

$$P^2 = \frac{1}{\dfrac{1}{P} - \dfrac{1}{P+1}} - P. \qquad (8.10)$$

The next algorithm uses (8.10) with proper scaling.

Algorithm 8.2. Squaring by reciprocals.

Input. An n-bit integer P, in binary representation.

Output. The binary representation of P^2.

Method

1. Use Algorithm 8.1 to compute $A = \lfloor 2^{4n-1}/P \rfloor$ by appending $2n$ 0's to P and applying RECIPROCAL† to compute $\lfloor 2^{6n-1}/P2^{2n} \rfloor$ and then shifting.

† RECIPROCAL was defined in Algorithm 8.1 only for integers whose length was a power of 2. The generalization to integers whose length is not a power of 2 should be obvious—add extra 0's and change scale when necessary.

2. Similarly, compute $B = \lfloor 2^{4n-1}/(P+1) \rfloor$.
3. Let $C = A - B$. Note that $C = 2^{4n-1}/(P^2+P) + T$ where $|T| \le 1$. The T arises from the fact that the truncation in computing A and B may give rise to an error of up to 1. Since $2^{2n-2} < P^2 + P < 2^{2n}$ we have $2^{2n+1} \ge C \ge 2^{2n-1}$.
4. Compute $D = \lfloor 2^{4n-1}/C \rfloor - P$.
5. Let Q be the last four bits of P. Adjust the last four bits of D up or down as little as possible to cause agreement with the last four bits of Q^2. □

Theorem 8.3. Algorithm 8.2 computes P^2.

Proof. Due to the truncation involved in steps 1 and 2, we can be assured only that

$$C = \frac{2^{4n-1}}{P^2+P} + T, \qquad \text{where} \quad |T| \le 1.$$

Since $2^{2n-1} \le C \le 2^{2n+1}$ and since the error in C is in the range -1 to 1, the error in $2^{4n-1}/C$ due to the error in C is at most

$$\left|\frac{2^{4n-1}}{C} - \frac{2^{4n-1}}{C-1}\right| = \left|\frac{2^{4n-1}}{C^2-C}\right|.$$

Since $C^2 - C \ge 2^{4n-3}$, the error is at most 4. The truncation at line 4 can increase the error to 5. Thus $|P^2 - D| \le 5$. Hence computing the last four bits of P^2 at step 5 insures that D is adjusted to be exactly P^2. □

Example 8.2. Let $n = 4$ and $P = [1101]$. Then

$$A = \lfloor 2^{15}/[1101] \rfloor = [100111011000]$$

and

$$B = \lfloor 2^{15}/[1110] \rfloor = [100100100100].$$

Next,

$$C = A - B = [10110100].$$

Then,

$$D = \lfloor 2^{15}/C \rfloor - P = [10110110] - [1101] = [10101001].$$

Thus D is 169, the square of 13, and no correction is necessary at step 5. □

Theorem 8.4. There exists a constant c such that $S(n) \le cR(n)$.

Proof. Algorithm 8.2 uses three reciprocal calculations on strings of length at most $3n$. Further, there are subtractions at steps 3 and 4 requiring $O_B(n)$ time, and a fixed amount of work at step 5, independent of n. Hence

$$S(n) \le 3R(3n) + c_1 n \tag{8.11}$$

for some constant c_1. Thus $S(n) \leq 27R(n) + c_1 n$. Since $R(n) \geq n$, choose $c = 27 + c_1$ to prove the theorem. □

Theorem 8.5. $M(n)$, $R(n)$, $D(n)$, and $S(n)$ are all related by constant factors.

Proof. We have already shown $R(n) \leq c_1 M(n)$ and $S(n) \leq c_2 R(n)$ for constants c_1 and c_2. It is easy to see that $M(n) \leq c_3 S(n)$ by noting that

$$AB = \tfrac{1}{2}[(A+B)^2 - A^2 - B^2].$$

Thus M, R, and S are related by constant factors.

When we discuss division of n-bit numbers, we really mean the division of a number with up to $2n$ bits by one of exactly n bits, producing an answer of at most $n + 1$ bits. It is trivial that $R(n) \leq D(n)$, so $M(n) \leq c_2 c_3 D(n)$. Moreover, using the identity $A/B = A * (1/B)$, we may show, taking care for scaling, that for some constant c

$$D(n) \leq M(2n) + R(2n) + cn. \tag{8.12}$$

Since $R(2n) \leq c_1 M(2n)$, and $M(2n) \leq 4M(n)$ is easy to show, we may rewrite (8.12) as

$$D(n) \leq 4(1 + c_1)M(n) + cn. \tag{8.13}$$

Since $M(n) \geq n$, we have from (8.13) that $D(n) \leq c_4 M(n)$, where $c_4 = 4 + 4c_1 + c$. Thus all functions have been shown to lie between $dM(n)$ and $eM(n)$ for some positive constants d and e. □

Corollary. Division of a $2n$-bit integer by an n-bit integer can be done in $O_B(n \log n \operatorname{loglog} n)$ time.

Proof. By Theorem 7.8 and Theorem 8.5. □

8.3 POLYNOMIAL MULTIPLICATION AND DIVISION

All the techniques of the previous section carry over to univariate polynomial arithmetic. Let $M(n)$, $D(n)$, $R(n)$, and $S(n)$ in this section stand for the time to multiply, divide, take reciprocals of, and square nth-degree polynomials. We assume, as before, that $a^2 M(n) \geq M(an) \geq aM(n)$ for $a \geq 1$ and similarly for the other functions.

By the "reciprocal" of an nth-degree polynomial $p(x)$, we mean $\lfloor x^{2n}/p(x) \rfloor$.† $D(n)$ is the time to find $\lfloor s(x)/p(x) \rfloor$, where $p(x)$ is of degree n and $s(x)$ is of degree at most $2n$. Note that we can "scale" polynomials by multi-

† By analogy with the notation for integers, we use the "floor function" to denote the quotient of polynomials. That is, if $p(x)$ is not a constant, $\lfloor s(x)/p(x) \rfloor$ is the unique $q(x)$ such that $s(x) = p(x)q(x) + r(x)$ and $\mathrm{DEG}(r(x)) < \mathrm{DEG}(p(x))$.

plying and dividing by powers of x, just as we scaled integers by powers of 2 in the previous section.

Since the results of this section are so similar to those for integers, we give only one result in detail: the polynomial "reciprocal" algorithm analogous to Algorithm 8.1 for integers. The polynomial algorithms are somewhat easier than the integer ones, due essentially to the fact that carries from place to place in the power series do not occur as they do for integers. Thus the polynomial algorithms require no adjustment of least significant places as was necessary in, for example, lines 5–7 of Algorithm 8.1.

Algorithm 8.3. Polynomial reciprocals.

Input. A polynomial $p(x)$ of degree $n-1$, where n is a power of 2 [i.e., $p(x)$ has 2^t terms for some integer t].

Output. The "reciprocal" $\lfloor x^{2n-2}/p(x) \rfloor$.

Method. In Fig. 8.2 we define a new procedure

$$\text{RECIPROCAL}\left(\sum_{i=0}^{k-1} a_i x^i\right),$$

where k is a power of 2 and $a_{k-1} \neq 0$. The procedure computes

$$\left\lfloor \frac{x^{2k-2}}{\sum_{i=0}^{k-1} a_i x^i} \right\rfloor.$$

Note that if $k = 1$, then the argument is a constant a_0 whose reciprocal is $1/a_0$, another constant. We assume each operation on coefficients can be done in one step, and no call to RECIPROCAL is necessary to compute $1/a_0$. The algorithm itself is to call RECIPROCAL with argument $p(x)$. □

```
procedure RECIPROCAL $\left(\sum_{i=0}^{k-1} a_i x^i\right)$:
    if $k = 1$ then return $1/a_0$
    else
        begin
            $q(x) \leftarrow$ RECIPROCAL $\left(\sum_{i=k/2}^{k-1} a_i x^{i-k/2}\right)$;
            $r(x) \leftarrow 2q(x)x^{(3/2)k-2} - (q(x))^2 \left(\sum_{i=0}^{k-1} a_i x^i\right)$;
            return $\lfloor r(x)/x^{k-2} \rfloor$
        end
```

Fig. 8.2. Algorithm to compute polynomial reciprocals.

Example 8.3. Let us compute $\lfloor x^{14}/p(x) \rfloor$, where $p(x) = x^7 - x^6 + x^5 + 2x^4 -$ $x^3 - 3x^2 + x + 4$. In line 2 we compute the reciprocal of $x^3 - x^2 + x + 2$, tha is, $q(x) = \lfloor x^6/(x^3 - x^2 + x + 2) \rfloor$. You may verify that $q(x) = x^3 + x^2 - 3$ Since $k = 8$, line 3 computes $r(x) = 2q(x)x^{10} - (q(x))^2 p(x) = x^{13} + x^{12} - 3x^{10} -$ $4x^9 + 3x^8 + 15x^7 + 12x^6 - 42x^5 - 34x^4 + 39x^3 + 51x^2 - 9x - 36$. Then a line 4, the result is $s(x) = x^7 + x^6 - 3x^4 - 4x^3 + 3x^2 + 15x + 12$. We not(that $s(x)p(x)$ is x^{14} plus a polynomial of degree 6. □

Theorem 8.6. Algorithm 8.3 correctly computes the reciprocal of a polynomial.

Proof. We prove by induction on k, for k a power of 2, that if $s(x) =$ RECIPROCAL$(p(x))$, and $p(x)$ is of degree $k - 1$, then $s(x)p(x) = x^{2k-2} + t(x)$, where $t(x)$ is of degree less than $k - 1$. The basis, $k = 1$, is trivial, since $p(x) = a_0$, $s(x) = 1/a_0$, and $t(x)$ need not exist.

For the inductive step, let $p(x) = p_1(x)x^{k/2} + p_2(x)$, where DEG$(p_1) = k/2 - 1$ and DEG$(p_2) \leq k/2 - 1$. Then by the inductive hypothesis, if $s_1(x) =$ RECIPROCAL$(p_1(x))$, then $s_1(x)p_1(x) = x^{k-2} + t_1(x)$, where DEG$(t_1) < k/2 - 1$. At line 3, we compute

$$r(x) = 2s_1(x)x^{(3/2)k-2} - (s_1(x))^2(p_1(x)x^{k/2} + p_2(x)). \tag{8.14}$$

It suffices to show that $r(x)p(x)$ is x^{3k-4} plus terms of degree less than $2k - 3$. Then division by x^{k-2} at line 4 produces the desired result.

By (8.14) and the fact that $p(x) = p_1(x)x^{k/2} + p_2(x)$, we have

$$\begin{aligned} r(x)p(x) = 2s_1(x)p_1(x)x^{2k-2} &+ 2s_1(x)p_2(x)x^{(3/2)k-2} \\ &- (s_1(x)p_1(x)x^{k/2} + s_1(x)p_2(x))^2. \end{aligned} \tag{8.15}$$

In substituting $x^{k-2} + t_1(x)$ for $s_1(x)p_1(x)$, in (8.15) we obtain

$$r(x)p(x) = x^{3k-4} - (t_1(x)x^{k/2} + s_1(x)p_2(x))^2. \tag{8.16}$$

Since DEG$(t_1) < k/2 - 1$, and $s_1(x)$ and $p_2(x)$ are of degree at most $k/2 - 1$, the terms other than x^{3k-4} in (8.16) are of degree at most $2k - 4$. □

The running times of Algorithms 8.3 and 8.1 are clearly analogous when one considers the two measures of complexity (arithmetic and bitwise, respectively) being used. In a similar manner we can show that the other time bounds of Section 8.2 apply to polynomials with arithmetic steps substituted for bit ones. Thus we have the following theorem.

Theorem 8.7. Let $M(n)$, $D(n)$, $R(n)$, and $S(n)$ be the arithmetic complexities of univariate polynomial multiplication, division, reciprocal-taking, and squaring respectively. These functions are all related by constant factors.

Proof. Analogous to Theorem 8.5 and the results leading up to that theorem. □

Corollary. Division of a $2n$th-degree polynomial by an nth-degree polynomial can be done in time $O_A(n \log n)$.

Proof. By Corollary 3 to Theorem 7.4 (p. 269) and Theorem 8.7. □

8.4 MODULAR ARITHMETIC

There are certain applications in which it is convenient to do integer arithmetic in a "modular" notation. That is, instead of representing an integer by a fixed-radix notation, we represent the integer by its residues modulo a set of pairwise relatively prime integers. If $p_0, p_1, \ldots, p_{k-1}$ are pairwise relatively prime integers and $p = \prod_{i=0}^{k-1} p_i$, then we can represent any integer u, $0 \le u < p$, uniquely by the set of residues $u_0, u_1, \ldots, u_{k-1}$ where $u_i = u$ modulo p_i, for $0 \le i < k$. When $p_0, p_1, \ldots, p_{k-1}$ are known, we write $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

It is quite easy to do addition, subtraction, and multiplication, provided the results continue to lie between 0 and $p - 1$ (or alternatively, these calculations can be regarded as being done modulo p). That is, let

$$u \leftrightarrow (u_0, u_1, \ldots, u_{k-1}) \quad \text{and} \quad v \leftrightarrow (v_0, v_1, \ldots, v_{k-1}).$$

Then

$$u + v \leftrightarrow (w_0, w_1, \ldots, w_{k-1}), \quad \text{where} \quad w_i = (u_i + v_i) \text{ modulo } p_i, \tag{8.17}$$

$$u - v \leftrightarrow (x_0, x_1, \ldots, x_{k-1}), \quad \text{where} \quad x_i = (u_i - v_i) \text{ modulo } p_i, \tag{8.18}$$

$$uv \leftrightarrow (y_0, y_1, \ldots, y_{k-1}), \quad \text{where} \quad y_i = u_i v_i \text{ modulo } p_i. \tag{8.19}$$

Example 8.4. Let $p_0 = 5$, $p_1 = 3$, and $p_2 = 2$. Then $4 \leftrightarrow (4, 1, 0)$, since $4 = 4$ modulo 5, $1 = 4$ modulo 3, and $0 = 4$ modulo 2. Similarly, $7 \leftrightarrow (2, 1, 1)$ and $28 \leftrightarrow (3, 1, 0)$. We observe that by (8.19) above, $4 \times 7 \leftrightarrow (3, 1, 0)$, which is the representation of 28. That is, the first component of 4×7 is 4×2 modulo 5, which is 3; the second component is 1×1 modulo 3, which is 1; and the last component is 0×1 modulo 2, which is 0. Also, $4 + 7 \leftrightarrow (1, 2, 1)$, which is the representation of 11, and $7 - 4 \leftrightarrow (3, 0, 1)$, which is the representation of 3. □

However, it is not clear how to do division economically using modular arithmetic. Note that u/v is not necessarily an integer and even if it were, we could not necessarily find its modular representation by computing (u_i/v_i) modulo p_i for each i. In fact, if p_i is not a prime, there may be several integers w between 0 and $p_i - 1$ which could be (u_i/v_i) modulo p_i in the sense that $wv_i \equiv u_i$ modulo p_i. For example, if $p_i = 6$, $v_i = 3$, and $u_i = 3$, then w could be 1, 3, or 5, since $1 \times 3 \equiv 3 \times 3 \equiv 5 \times 3 \equiv 3$ modulo 6. Thus (u_i/v_i) modulo p_i may not "make sense."

The advantage of modular representation is chiefly that arithmetic can be implemented with less hardware than is required for conventional arithmetic, since calculations are done for each modulus independently of the others. No carries are needed as for the usual (radix) number representations. Unfortunately, the problems of doing division and of detecting overflows (telling whether the result is outside the range 0 to $p - 1$) efficiently appear insurmountable, and because of this such systems are rarely implemented in general purpose computer hardware.

Nevertheless, the ideas involved do find use, mostly in the polynomial domain. Here, we are likely to find ourselves in a situation where no polynomial division is required. Also, we shall see in the next section that evaluation of polynomials and computation of residues of polynomials (modulo other polynomials) are closely related. We first prove that modular arithmetic for integers "works" as intended.

The first part of the proof is that Eqs. (8.17), (8.18), and (8.19) hold. These relationships are straightforward, and we leave them as exercises. The second part of the proof is to show that the correspondence $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is one-to-one (an isomorphism). Although this result is not hard, we give it as a lemma.

Lemma 8.1. Let $p_0, p_1, \ldots, p_{k-1}$ be a set of integers which are pairwise relatively prime. Let

$$p = \prod_{i=0}^{k-1} p_i$$

and let $u_i = u$ modulo p_i. Then $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is a one-to-one correspondence between integer u, $0 \le u < p$, and

$$(u_0, u_1, \ldots, u_{k-1}), \quad 0 \le u_i < p_i,$$

for $0 \le i < k$.

Proof. Clearly, for each u there exists a corresponding k-tuple. Since there are exactly p values of u in the interval and exactly p k-tuples, it suffices to show that to each k-tuple there corresponds at most one integer u. Assume u and v, where $0 \le u < v < p$, both correspond to the k-tuple $(u_0, u_1, \ldots, u_{k-1})$. Then $v - u$ must be a multiple of each p_i. Since the p_i's are relatively prime, $v - u$ must be a multiple of p. Since $u \ne v$, and $v - u$ is a multiple of p, then u and v must differ by at least p and hence cannot both be in the interval 0 to $p - 1$. □

In order to use modular arithmetic, algorithms are needed to convert from radix notation to modular notation and back. One method to convert an integer u from radix notation to modular notation is to divide u by each of the p_i, $0 \le i < k$.

Assume that each p_i requires b bits in binary notation. Then

$$p = \prod_{i=0}^{k-1} p_i$$

requires roughly bk bits, and division of u by each of the p_i's, where $0 \leq u < p$, could involve k divisions of a kb-bit integer by integers of b bits. By breaking each division into k divisions of $2b$-bit integers by b-bit integers, we can convert to modular notation in $O_B(k^2D(b))$ time, where $D(n)$ is the time to do integer division [at most $O_B(n \log n \operatorname{loglog} n)$ by the corollary to Theorem 8.5].

However, we can do the job in considerably less time by a method reminiscent of the technique used to perform polynomial division in Section 7.2. Instead of dividing an integer u by each of k moduli $p_0, p_1, \ldots, p_{k-1}$, we first compute the products $p_0p_1, p_2p_3, \ldots, p_{k-2}p_{k-1}$, then the products $p_0p_1p_2p_3, p_4p_5p_6p_7, \ldots$ and so on. Next we compute the residues by a divide-and-conquer approach. By division we obtain the residues u_1 and u_2 of u modulo $p_0 \cdots p_{k/2-1}$ and u modulo $p_{k/2} \cdots p_{k-1}$. The problem of computing u modulo p_i, $0 \leq i < k$, is now reduced to two problems of half size. That is, u modulo $p_i = u_1$ modulo p_i for $0 \leq i < k/2$, and u modulo $p_i = u_2$ modulo p_i for $k/2 \leq i < k$.

Algorithm 8.4. Computation of residues.

Input. Moduli $p_0, p_1, \ldots, p_{k-1}$, and integer u, where $0 \leq u < p = \prod_{i=0}^{k-1} p_i$.

Output. u_i, $0 \leq i < k$, where $u_i = u$ modulo p_i.

Method. Assume k is a power of 2, say $k = 2^t$. (If necessary, add extra moduli, all of which are 1, to the input so k becomes a power of 2.) We begin by computing certain products of the moduli similar to the q_{lm}'s computed in Section 7.2.

For $0 \leq j < t$, i a multiple of 2^j, and $0 \leq i < k$, let

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m.$$

Thus $q_{i0} = p_i$ and $q_{ij} = q_{i,j-1} \times q_{i+2^{j-1},j-1}$.

We first compute the q_{ij}'s, then find the remainder u_{ij} when u is divided by each of the q_{ij}'s. The desired answers are the u_{i0}'s. The details are in the program of Fig. 8.3. □

Theorem 8.8. Algorithm 8.4 correctly computes the u_i's.

Proof. The proof parallels that of Theorem 7.3, where a polynomial was evaluated at the nth roots of unity. It is easy to show by induction on j that line 4 computes the q_{ij}'s correctly. Then, by backwards induction on the j of line 6, we show that $u_{ij} = u$ modulo q_{ij}. Lines 8 and 9 make this easy. Letting $j = 0$, we have $u_i = u$ modulo p_i. Details are left as an exercise. □

Theorem 8.9. Algorithm 8.4 requires $O_B(M(bk) \log k)$ time if at most b bits are needed to represent each of the p_i's.

Proof. It is easy to see that the loops of lines 3–4 and 7–9 are the

```
        begin
1.          for $i \leftarrow 0$ until $k-1$ do $q_{i0} \leftarrow p_i$;
2.          for $j \leftarrow 1$ until $t-1$ do
3.              for $i \leftarrow 0$ step $2^j$ until $k-1$ do
4.                  $q_{ij} \leftarrow q_{i,j-1} * q_{i+2^{j-1},j-1}$;
5.              $u_{0t} \leftarrow u$;
6.              for $j \leftarrow t$ step $-1$ until 1 do
7.                  for $i \leftarrow 0$ step $2^j$ until $k-1$ do
                        begin
8.                          $u_{i,j-1} \leftarrow$ REMAINDER$(u_{ij}/q_{i,j-1})$;
9.                          $u_{i+2^{j-1},j-1} \leftarrow$ REMAINDER$(u_{ij}/q_{i+2^{j-1},j-1})$
                        end;
10.             for $i \leftarrow 0$ until $k-1$ do $u_i \leftarrow u_{i0}$
        end
```

Fig. 8.3. Computation of residues.

most costly. Each requires $O_B(2^{t-j}M(2^{j-1}b))$ time.† Since we assume $M(an) \geq aM(n)$ for $a \geq 1$, we see the cost of these loops is bounded by $O_B(M(2^t b)) = O_B(M(kb))$. Since each of these loops is executed $t = \log k$ times at most, we have our result. □

Corollary. If b bits are required to represent each of the moduli $p_0, p_1, \ldots, p_{k-1}$ then the residues may be computed in at most $O_B(bk \log k \log bk \operatorname{loglog} bk)$ time.

8.5 MODULAR POLYNOMIAL ARITHMETIC AND POLYNOMIAL EVALUATION

Results analogous to those for integers hold for polynomials. Let $p_0, \ldots, p_{k-1}$ be polynomials and $p = \Pi_{i=0}^{k-1} p_i$. Then each polynomial u can be represented by the sequence $u_0, u_1, \ldots, u_{k-1}$ of remainders obtained by dividing u by each p_i. That is, u_i is the unique polynomial with DEG(u_i) < DEG(p_i) such that $u = p_i q_i + u_i$ for some polynomial q_i. We write $u_i = u$ modulo p_i in this situation, in complete analogy with integer modular arithmetic.

In analogy with Lemma 8.1, we may show that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is a one-to-one correspondence if the p_i's are pairwise relatively prime and u is restricted to have degree less than that of p, i.e., SIZE(u) < SIZE(p). More importantly, Algorithm 8.4 for computing residues works if the p_i's are polynomials instead of integers. Instead of b (the number of bits in the p_i's), we

† Recall that $D(n)$ and $M(n)$ are essentially the same.

must consider the maximum degree of the polynomials p_i. Of course, complexity is in terms of arithmetic steps rather than bitwise ones. With these changes, we have the following analog of Theorem 8.9.

Theorem 8.10. By a conversion of Algorithm 8.4 to the polynomial domain, it is possible to find the residues with respect to polynomials $p_0, p_1, \ldots, p_{k-1}$ of polynomial u in time $O_A(M(dk) \log k)$, where d is an upper bound on the degree of the p_i's and the degree of u is less than that of $\Pi_{i=0}^{k-1} p_i$.

Proof. Analogous to Theorem 8.9 and left for an exercise. □

Corollary 1. Finding residues of a polynomial u with respect to polynomials $p_0, p_1, \ldots, p_{k-1}$ requires time at most $O_A(dk \log k \log dk)$, where d is an upper bound on the degree of the p_i's and the degree of u is less than that of $\Pi_{i=0}^{k-1} p_i$.

Example 8.5. Consider the four polynomial moduli

$$\begin{aligned} p_0 &= x - 3, \\ p_1 &= x^2 + x + 1, \\ p_2 &= x^2 - 4, \\ p_3 &= 2x + 2, \end{aligned}$$

and suppose $u = x^5 + x^4 + x^3 + x^2 + x + 1$. First compute the products

$$\begin{aligned} p_0 p_1 &= x^3 - 2x^2 - 2x - 3, \\ p_2 p_3 &= 2x^3 + 2x^2 - 8x - 8. \end{aligned}$$

Then, compute

$$\begin{aligned} u' &= u \text{ modulo } p_0 p_1 = 28x^2 + 28x + 28, \\ u'' &= u \text{ modulo } p_2 p_3 = 21x + 21. \end{aligned}$$

That is, $u = p_0 p_1 (x^2 + 3x + 9) + 28x^2 + 28x + 28$, and $u = p_2 p_3 (\frac{1}{2}x^2 + \frac{5}{2}) + 21x + 21$.

Next, compute

$$\begin{aligned} u \text{ modulo } p_0 &= u' \text{ modulo } p_0 = 364, \\ u \text{ modulo } p_1 &= u' \text{ modulo } p_1 = 0, \\ u \text{ modulo } p_2 &= u'' \text{ modulo } p_2 = 21x + 21, \\ u \text{ modulo } p_3 &= u'' \text{ modulo } p_3 = 0. \; \Box \end{aligned}$$

Note that the FFT algorithm of Section 7.2 is really an implementation of this algorithm, where the polynomials $p_0, p_1, \ldots, p_{k-1}$ are $x - \omega^0, x - \omega^1, \ldots, x - \omega^{n-1}$. The FFT algorithm took advantage of the fact that $p_i = x - \omega^i$. Because of the ordering of the p_i's, each product had the form of a power of x minus a power of ω and hence division was especially easy.

As we observed in Section 7.2, if p_i is the first-degree polynomial $x - a$ then u modulo p_i is $u(a)$. Therefore, the case in which all the p_i's are o; degree 1 is especially important. We have the following corollary tc Theorem 8.10.

Corollary 2. An nth-degree polynomial can be evaluated at n points in time $O_A(n \log^2 n)$.

Proof. To evaluate $u(x)$ at the n points $a_0, a_1, \ldots, a_{n-1}$ we compute $u(x)$ modulo$(x - a_i)$ for $0 \leq i < n$. This evaluation requires $O_A(n \log^2 n)$ time by Corollary 1, since d there is 1. □

8.6 CHINESE REMAINDERING

We now consider the problem of converting an integer from modular notation to radix notation.† Suppose we are given relatively prime moduli $p_0, p_1, \ldots, p_{k-1}$ and residues $u_0, u_1, \ldots, u_{k-1}$, where $k = 2^t$, and we wish to find the integer u such that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$. We may do so by the integer analog of the *Lagrangian interpolation formula* for polynomials.

Lemma 8.2. Let c_i be the product of all the p_j's except p_i (that is, $c_i = p/p_i$ where $p = \Pi_{j=0}^{k-1} p_j$). Let d_i be c_i^{-1} modulo p_i (that is, $d_i c_i \equiv 1$ modulo p_i, and $0 \leq d_i < p_i$). Then

$$u \equiv \sum_{i=0}^{k-1} c_i d_i u_i \text{ modulo } p. \tag{8.20}$$

Proof. Since the p_j's are relatively prime to one another, we know d_i exists and is unique (Exercise 7.9). Also, c_i is divisible by p_j for $j \neq i$, so $c_i d_i u_i \equiv 0$ modulo p_j if $j \neq i$. Thus

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv c_j d_j u_j \text{ modulo } p_j.$$

Since $c_j d_j \equiv 1$ modulo p_j, we have

$$\sum_{i=0}^{k-1} c_i d_i u_i \equiv u_j \text{ modulo } p_j.$$

Since p_j divides p, these relationships hold even if all arithmetic is done modulo p. Thus (8.20) holds. □

Our problem is to compute (8.20) efficiently. To begin, it is hardly clear how to compute the d_i's from the p_i's except by trial and error. We shall later see that this task is not hard, given the Euclidean algorithm of Section 8.8 and

† This process is known as *Chinese remaindering*, since an algorithm for the process was known to the Chinese over 2000 years ago.

the fast implementation in Section 8.10. However, in this section we shall study only the "preconditioned" form of the Chinese remainder problem. If a portion of the input is fixed for a number of problems, then all constants depending on the fixed portion of the input can be precomputed and supplied as part of the input. If any such precomputation of constants is done, the algorithm is said to be *preconditioned.*

For the preconditioned Chinese remainder algorithm, the input will be not only the moduli and the p_i's, but also the inverses (the d_i's). This situation is not unrealistic. If we are frequently using the Chinese remainder algorithm to convert numbers represented by a fixed set of moduli, it is reasonable to precompute all functions of these moduli that are used by the algorithm. In the corollary to Theorem 8.21 we shall see that as far as order of magnitude is concerned, it makes only a modest difference whether the algorithm is preconditioned or not.

If we look at (8.20), we notice that the terms $c_i d_i u_i$ have many factors in common as i varies. For example, $c_i d_i u_i$ has $p_0 p_1 \cdots p_{k/2-1}$ as a factor whenever $i \geq k/2$, and it has $p_{k/2} p_{k/2+1} \cdots p_{k-1}$ as a factor if $i < k/2$. Thus we could write (8.20) as

$$u = \left(\sum_{i=0}^{k/2-1} c'_i d_i u_i\right) \times \prod_{i=k/2+1}^{k-1} p_i + \left(\sum_{i=k/2+1}^{k-1} c''_i d_i u_i\right) \times \prod_{i=0}^{k/2-1} p_i$$

where c'_i is the product $p_0 p_1 \cdots p_{(k/2)-1}$ with p_i missing, and c''_i is the product $p_{k/2} p_{k/2+1} \cdots p_{k-1}$ with p_i missing. This observation should suggest a divide-and-conquer approach similar to that used for computing residues. We compute the products

$$q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$$

(as in Algorithm 8.4) and then the integers

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

If $j = 0$, then $s_{i0} = d_i u_i$. If $j > 0$, we compute s_{ij} by the formula

$$s_{ij} = s_{i,j-1} q_{i+2^{j-1},j-1} + s_{i+2^{j-1},j-1} q_{i,j-1}.$$

Ultimately we produce $s_{0t} = u$, which evaluates (8.20).

Algorithm 8.5. Preconditioned fast Chinese remainder algorithm.

Input

1. Relatively prime integer moduli $p_0, p_1, \ldots, p_{k-1}$, where $k = 2^t$ for some t.
2. The set of "inverses" $d_0, d_1, \ldots, d_{k-1}$ such that $d_i = (p/p_i)^{-1}$ modulo p_i, where $p = \prod_{i=0}^{k-1} p_i$.
3. A sequence of residues $(u_0, u_1, \ldots, u_{k-1})$.

```
   begin
1.     for $i \leftarrow 0$ until $k - 1$ do $s_{i0} \leftarrow d_i * u_i$;
2.     for $j \leftarrow 1$ until $t$ do
3.         for $i \leftarrow 0$ step $2^j$ until $k - 1$ do
4.             $s_{ij} \leftarrow s_{i,j-1} * q_{i+2^{j-1},j-1} + s_{i+2^{j-1},j-1} * q_{i,j-1}$;
5.     write $s_{0t}$ modulo $q_{0t}$
   end
```

Fig. 8.4. Program to compute integer from modular representation.

Output. The unique integer u, $0 \leq u < p$, such that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

Method. First compute $q_{ij} = \Pi_{m=1}^{i+2^j-1} p_m$ as in Algorithm 8.4.† Then execute the program in Fig. 8.4, where s_{ij} is intended to be

$$\sum_{m=}^{i+2^j-1} \frac{q_{ij} d_m u_m}{p_m}. \quad \square$$

Example 8.6. Let p_0, p_1, p_2, p_3 be 2, 3, 5, 7, and let (u_0, u_1, u_2, u_3) be (1, 2, 4, 3). Then $q_{i0} = p_i$ for $0 \leq i < 4$, $q_{01} = 6$, $q_{21} = 35$, and $q_{02} = p = 210$. We note that

$$\begin{array}{lll}
d_0 = (3 * 5 * 7)^{-1} \text{ modulo } 2 = 1, & \text{since} & 1 * 105 \equiv 1 \text{ modulo } 2, \\
d_1 = (2 * 5 * 7)^{-1} \text{ modulo } 3 = 1, & \text{since} & 1 * 70 \equiv 1 \text{ modulo } 3, \\
d_2 = (2 * 3 * 7)^{-1} \text{ modulo } 5 = 3, & \text{since} & 3 * 42 \equiv 1 \text{ modulo } 5, \\
d_3 = (2 * 3 * 5)^{-1} \text{ modulo } 7 = 4, & \text{since} & 4 * 30 \equiv 1 \text{ modulo } 7.
\end{array}$$

Thus the effect of line 1 is to compute

$$\begin{array}{ll}
s_{00} = 1 * 1 = 1, & s_{10} = 1 * 2 = 2, \\
s_{20} = 3 * 4 = 12, & s_{30} = 4 * 3 = 12.
\end{array}$$

We then execute the loop of lines 3–4 with $j = 1$. Here, i takes on the values 0 and 2; so we compute

$$\begin{aligned}
s_{01} &= s_{00} * q_{10} + s_{10} * q_{00} = 1 * 3 + 2 * 2 = 7, \\
s_{21} &= s_{20} * q_{30} + s_{30} * q_{20} = 12 * 7 + 12 * 5 = 144.
\end{aligned}$$

Next, we execute the loop of lines 3–4 with $j = 2$, and i takes only the value 0. We compute

$$s_{02} = s_{01} * q_{21} + s_{21} * q_{01} = 7 * 35 + 144 * 6 = 1109.$$

† Note that the q_{ij}'s are functions only of the p_i's. We should rightly include them as inputs rather than calculating them, since we allow preconditioning. However, it is easily shown that the order-of-magnitude running time is not affected by whether or not the q_{ij}'s are precomputed.

The result at line 5 is thus 1109 modulo 210, which is 59. You may check that the residues of 59 modulo 2, 3, 5, and 7 are 1, 2, 4, and 3, respectively. Figure 8.5 graphically portrays the computations. □

Theorem 8.11. Algorithm 8.5 correctly computes the integer u such that $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$.

Proof. An elementary induction on j proves that s_{ij} is given its intended value, that is

$$s_{ij} = \sum_{m=i}^{i+2^j-1} q_{ij} d_m u_m / p_m.$$

The correctness of the algorithm then follows immediately from Lemma 8.2, that is, from the correctness of Eq. (8.20). □

Theorem 8.12. Suppose we are given k relatively prime integer moduli $p_0, p_1, \ldots, p_{k-1}$ and residues $(u_0, u_1, \ldots, u_{k-1})$. If each of the p_i's requires at most b bits, there is a preconditioned algorithm that computes u such that $0 \le u < p = \prod_{i=0}^{k-1} p_i$ and $u \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ in time $O_B(M(bk) \log k)$, where $M(n)$ is the time to multiply two n-bit numbers.

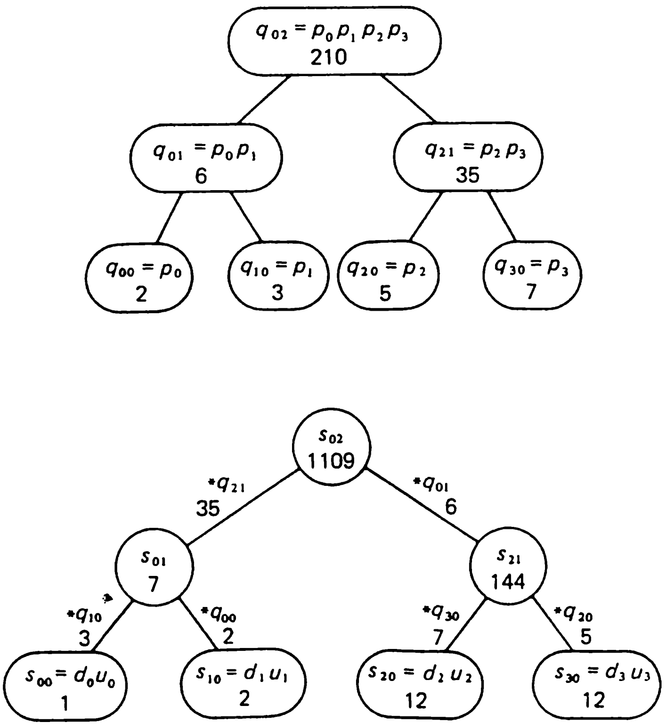


Fig. 8.5 Computations of Example 8.6.

Proof. The computation of the q_{ij}'s requires $O_B(M(bk) \log k)$ time.† For the analysis of the body of the algorithm, note that s_{ij} requires at most $b2^j + b + j$ bits, since it is the sum of 2^j terms, each of which is the product of $2^j + 1$ integers of b or fewer bits. Hence each term requires no more than $b(2^j + 1)$ bits, and the sum of 2^j such terms requires at most $b(2^j + 1) + \log(2^j) = b2^j + b + j$ bits. Thus line 4 takes time $O_B(M(b2^j))$. The loop of lines 3–4 is executed $k/2^j$ times for fixed j, so the total cost of the loop is

$$O_B\left(\frac{k}{2^j} M(b2^j)\right),$$

which, by our usual assumption about the growth of $M(n)$, is bounded above by $O_B(M(bk))$. Since the loop of lines 2–4 is iterated $\log k$ times, the total cost is $O_B(M(bk) \log k)$ time. Line 5 is easily shown to cost less than this. □

Corollary. The preconditioned Chinese remaindering algorithm with k moduli of b bits each requires at most $O_B(bk \log k \log bk \operatorname{loglog} bk)$ steps.

8.7 CHINESE REMAINDERING AND INTERPOLATION OF POLYNOMIALS

It should be clear that all the results of the previous section hold for polynomial moduli $p_0, p_1, \ldots, p_{k-1}$ as well as for integers. Thus we have the following theorem and corollary.

Theorem 8.13. Suppose $p_0(x), p_1(x), \ldots, p_{k-1}(x)$ are polynomials of degree at most d, and $M(n)$ is the number of arithmetic steps needed to multiply two nth-degree polynomials. Then given polynomials $u_0(x), u_1(x), \ldots, u_{k-1}(x)$, where the degree of $u_i(x)$ is less than that of $p_i(x)$, for $0 \le i < k$, there is a preconditioned algorithm to compute the unique polynomial $u(x)$ of degree less than that of $p(x) = \Pi_{i=0}^{k-1} p_i(x)$ such that $u(x) \leftrightarrow (u_0(x), u_1(x), \ldots, u_{k-1}(x))$ in $O_A(M(dk) \log k)$ time.

Proof. Similar to Algorithm 8.5 and Theorem 8.12. □

Corollary. There is a preconditioned algorithm for Chinese remaindering of polynomials requiring $O_A(dk \log k \log dk)$ time.

An important special case occurs when all the moduli have degree 1. If $p_i = x - a_i$ for $0 \le i < k$, then the residues (the u_i's) are constants, i.e., polynomials of degree 0. If $u(x) \equiv u_i$ modulo $(x - a_i)$, then $u(x) = q(x)(x - a_i) + u_i$, so $u(a_i) = u_i$. Thus the unique polynomial $u(x)$ of degree less than k such that $u(x) \leftrightarrow (u_0, u_1, \ldots, u_{k-1})$ is the unique polynomial of degree less than k such

† Since $D(n)$ and $M(n)$ are essentially the same functions, we use $M(n)$ in preference throughout.

that $u(a_i) = u_i$ for each i, $0 \le i < k$. Put another way, $u(x)$ is the polynomial interpolated through the points (a_i, u_i) for $0 \le i < k$.

Since interpolation is an important polynomial operation, it is pleasant to observe that interpolation through k points can be done in $O_{\rm A}(k \log^2 k)$ time even if preconditioning is not allowed. This is so because, as we shall see from the next lemma, the coefficients d_i from Eq. (8.20) can be evaluated easily in this special case.[†]

Lemma 8.3. Let $p_i(x) = x - a_i$, for $0 \le i < k$, where the a_i's are distinct (i.e., the $p_i(x)$'s are relatively prime). Let $p(x) = \prod_{j=0}^{k-1} p_j(x)$, let $c_i(x) = p(x)/p_i(x)$ and let $d_i(x)$ be the constant polynomial such that $d_i(x)c_i(x) \equiv 1$ modulo $p_i(x)$. Then $d_i(x) = 1/b$, where $b = \left.\dfrac{d}{dx}\, p(x)\right|_{x=a_i}$.

Proof. We can write $p(x) = c_i(x)p_i(x)$, so

$$\frac{d}{dx}\, p(x) = p_i(x)\, \frac{d}{dx}\, c_i(x) + c_i(x)\, \frac{d}{dx}\, p_i(x). \tag{8.21}$$

Now, $dp_i(x)/dx = 1$, and $p_i(a_i) = 0$. Thus

$$\left.\frac{dp(x)}{dx}\right|_{x=a_i} = c_i(a_i). \tag{8.22}$$

Note that $d_i(x)$ has the property that $d_i(x)c_i(x) \equiv 1$ modulo $(x - a_i)$, so $d_i(x)c_i(x) = q_i(x)(x - a_i) + 1$ for some $q_i(x)$. Thus $d_i(a_i) = 1/c_i(a_i)$. The lemma is now immediate from (8.22) since $d_i(x)$ is a constant. □

Theorem 8.14. We may interpolate a polynomial through k points in $O_{\rm A}(k \log^2 k)$ time without preconditioning.

Proof. By Lemma 8.3, the computation of the d_i's is equivalent to evaluating the derivative of a $(k-1)$st-degree polynomial at k points. The polynomial $p(x) = \prod_{i=0}^{k-1} p_i(x)$ can be obtained in $O_{\rm A}(k \log^2 k)$ time by first computing $p_0p_1, p_2p_3, \ldots$, then $p_0p_1p_2p_3, p_4p_5p_6p_7, \ldots$, and so on. The derivative of $p(x)$ can be taken in $O_{\rm A}(k)$ steps. The evaluation of the derivative requires $O_{\rm A}(k \log^2 k)$ time by Corollary 2 to Theorem 8.10. The theorem then follows from the corollary to Theorem 8.13 with $d = 1$. □

Example 8.7. Let us interpolate a polynomial through the points (1, 2), (2, 7), (3, 4), and (4, 8). That is, $a_i = i + 1$ for $0 \le i < 4$, $u_0 = 2$, $u_1 = 7$, $u_2 = 4$, and $u_3 = 8$. Then $p_i(x) = x - i -.1$, and $p(x) = \prod_{i=0}^{3} p_i(x)$ is $x^4 - 10x^3 + 35x^2 - 50x + 24$. Next, $dp(x)/dx = 4x^3 - 30x^2 + 70x - 50$, and its values at 1, 2, 3, 4 are -6, $+2$, -2, $+6$, respectively. Thus d_0, d_1, d_2, and d_3 are $-\frac{1}{6}$,

[†] As was alluded to in Section 8.6, the task is really not hard in the general case. However, in the general case, we need the machinery of the next section.

$+\frac{1}{2}$, $-\frac{1}{2}$, and $+\frac{1}{6}$, respectively. Using the fast Chinese remainder algorithm redone for polynomials (Algorithm 8.5) we compute:

$$\begin{aligned} s_{01} &= d_0u_0p_1 + d_1u_1p_0 = (-\tfrac{1}{6})(2)(x-2) + (\tfrac{1}{2})(7)(x-1) = \tfrac{19}{6}x - \tfrac{17}{6}, \\ s_{21} &= d_2u_2p_3 + d_3u_3p_2 = (-\tfrac{1}{2})(4)(x-4) + (\tfrac{1}{6})(8)(x-3) = -\tfrac{2}{3}x + 4. \end{aligned}$$

Then

$$\begin{aligned} s_{02} = u(x) &= s_{01}q_{21} + s_{21}q_{01} \\ &= (\tfrac{19}{6}x - \tfrac{17}{6})(x^2 - 7x + 12) + (-\tfrac{2}{3}x + 4)(x^2 - 3x + 2), \\ u(x) &= \tfrac{5}{2}x^3 - 19x^2 + \tfrac{89}{2}x - 26. \ \square \end{aligned}$$

As we mentioned in Chapter 7, we can do polynomial arithmetic, such as addition, subtraction, and multiplication, by evaluating polynomials at n points, performing the arithmetic on the values at the points, and then interpolating a polynomial through the resulting values. If the answer is a polynomial of degree $n - 1$ or less, this technique will yield the correct answer.

The FFT is a method of doing just this, where the points selected are $\omega^0, \omega^1, \ldots, \omega^{n-1}$. In this case, the evaluation and interpolation algorithms were made especially easy because of properties of the powers of ω and the particular order of these powers that we chose. However, it is worth noting that we could use any collection of points to substitute for the powers of ω. Then we would have a "transform" that required $O_A(n \log^2 n)$, rather than $O_A(n \log n)$ time, to compute and invert.

8.8 GREATEST COMMON DIVISORS AND EUCLID'S ALGORITHM

Definition: Let a_0 and a_1 be positive integers. A positive integer g is called a *greatest common divisor* of a_0 and a_1, often denoted GCD(a_0, a_1), if

1. g divides both a_0 and a_1, and
2. every divisor of both a_0 and a_1 divides g.

It is easy to show that if a_0 and a_1 are positive integers, then g is unique. For example, GCD(57, 33) is 3.

Euclid's algorithm for computing GCD(a_0, a_1) is to compute the *remainder sequence* $a_0, a_1, \ldots, a_k$, where a_i, for $i \geq 2$, is the nonzero remainder resulting from the division of a_{i-2} by a_{i-1}, and where a_k divides a_{k-1} exactly (i.e., $a_{k+1} = 0$). Then GCD(a_0, a_1) $= a_k$.

Example 8.8. In the example above, $a_0 = 57$, $a_1 = 33$, $a_2 = 24$, $a_3 = 9$, $a_4 = 6$ and $a_5 = 3$. Thus $k = 5$ and GCD(57, 33) = 3. □

Theorem 8.15. Euclid's algorithm correctly computes GCD(a_0, a_1).

Proof. The algorithm computes $a_{i+1} = a_{i-1} - q_ia_i$ for $1 \leq i < k$, where $q_i = \lfloor a_{i-1}/a_i \rfloor$. Since $a_{i+1} < a_i$, the algorithm will clearly terminate. Moreover

```
begin
    $x_0 \leftarrow 1$; $y_0 \leftarrow 0$; $x_1 \leftarrow 0$; $y_1 \leftarrow 1$; $i \leftarrow 1$;
    while $a_i$ does not divide $a_{i-1}$ do
        begin
            $q \leftarrow \lfloor a_{i-1}/a_i \rfloor$;
            $a_{i+1} \leftarrow a_{i-1} - q * a_i$;,
            $x_{i+1} \leftarrow x_{i-1} - q * x_i$;
            $y_{i+1} \leftarrow y_{i-1} - q * y_i$;
            $i \leftarrow i + 1$
        end
    write $a_i$; write $x_i$; write $y_i$
end
```

Fig. 8.6. Extended Euclidean algorithm.

any divisor of both a_{i-1} and a_i is a divisor of a_{i+1}, and any divisor of a_i and a_{i+1} is also a divisor of a_{i-1}. Hence $\text{GCD}(a_0, a_1) = \text{GCD}(a_1, a_2) = \cdots = \text{GCD}(a_{k-1}, a_k)$. Since $\text{GCD}(a_{k-1}, a_k)$ is clearly a_k, we have our result. □

The Euclidean algorithm can be extended to find not only the greatest common divisor of a_0 and a_1, but also to find integers x and y such that $a_0x + a_1y = \text{GCD}(a_0, a_1)$. The algorithm is as follows.

Algorithm 8.6. Extended Euclidean algorithm.

Input. Positive integers a_0 and a_1.

Output. $\text{GCD}(a_0, a_1)$ and integers x and y such that $a_0x + a_1y = \text{GCD}(a_0, a_1)$.

Method. We execute the program in Fig. 8.6. □

Example 8.9. If $a_0 = 57$ and $a_1 = 33$, we obtain the following values for the a_i's, x_i's, and y_i's.

i	a_i	x_i	y_i
0	57	1	0
1	33	0	1
2	24	1	−1
3	9	−1	2
4	6	3	−5
5	3	−4	7

Note that $57 \times (-4) + 33 \times 7 = 3$. □

It should be clear that Algorithm 8.6 correctly computes $\text{GCD}(a_0, a_1)$, since the a_i's clearly form the remainder sequence. An important property of the x_i's and y_i's computed by Algorithm 8.6 is the subject of the next lemma.

Lemma 8.4. In Algorithm 8.6, for $i \geq 0$

$$a_0x_i + a_1y_i = a_i. \tag{8.23}$$

Proof. Equation (8.23) holds for $i = 0$ and $i = 1$ by line 1 of Algorithm 8.6. Assume (8.23) holds for $i - 1$ and i. Then $x_{i+1} = x_{i-1} - qx_i$ by line 5 and $y_{i+1} = y_{i-1} - qy_i$ by line 6. Thus

$$a_0x_{i+1} + a_1y_{i+1} = a_0x_{i-1} + a_1y_{i-1} - q(a_0x_i + a_1y_i). \tag{8.24}$$

By the inductive hypothesis and (8.24), we have

$$a_0x_{i+1} + a_1y_{i+1} = a_{i-1} - qa_i.$$

Since $a_{i-1} - qa_i = a_{i+1}$ by line 4, we have our result. □

Note that Lemma 8.4 does not even depend on the way q is computed at line 3, although line 3 is essential to guarantee that Algorithm 8.6 does compute $\text{GCD}(a_0, a_1)$. We may put these observations together and prove the following theorem.

Theorem 8.16. Algorithm 8.6 computes $\text{GCD}(a_0, a_1)$ and numbers x and y such that $a_0x + a_1y = \text{GCD}(a_0, a_1)$.

Proof. Elementary exercise from Lemma 8.4. □

We now introduce some notation that will be useful in developments of the next section.

Definition. Let a_0 and a_1 be integers with remainder sequence $a_0, a_1, \ldots, a_k$. Let $q_i = \lfloor a_{i-1}/a_i \rfloor$, for $1 \leq i \leq k$. We define 2×2 matrices $R_{ij}^{(a_0,a_1)}$, or R_{ij} where a_0 and a_1 are understood for $0 \leq i \leq j \leq k$, by:

1. $R_{ii} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, for $i \geq 0$.

2. If $j > i$, then $R_{ij} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -q_{j-1} \end{bmatrix} * \cdots * \begin{bmatrix} 0 & 1 \\ 1 & -q_{i+1} \end{bmatrix}$.

Example 8.10. Let $a_0 = 57$ and $a_1 = 33$, with remainder sequence 57, 33, 24, 9, 6, 3 and quotients q_i, for $1 \leq i \leq 4$, given by 1, 1, 2, 1. Then

$$R_{04}^{(57,33)} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 3 & -5 \\ -4 & 7 \end{bmatrix}. \; \square$$

Two interesting properties of these matrices are given in the next lemma.

Lemma 8.5

a) $\begin{bmatrix} a_j \\ a_{j+1} \end{bmatrix} = R_{ij} \begin{bmatrix} a_i \\ a_{i+1} \end{bmatrix}$ for $i < j < k$,

b) $R_{0j} = \begin{bmatrix} x_j & y_j \\ x_{j+1} & y_{j+1} \end{bmatrix}$, for $0 \leq j < k$, where the a_i's, x_i's, and y_i's are as defined in the extended Euclidean algorithm.

Proof. Elementary inductive exercises. □

We should observe that all the developments of this section go through with univariate polynomials instead of integers, with the following modification. While it is easy to show that the greatest common divisor of two integers is uniquely defined, for polynomials over a field the greatest common divisor is unique only up to multiplication by a constant. That is, if $g(x)$ divides polynomials $a_0(x)$ and $a_1(x)$, and any other divisor of these two polynomials also divides $g(x)$, then $cg(x)$ also has this property for any constant $c \neq 0$. We shall be satisfied with any polynomial that divides $a_0(x)$ and $a_1(x)$ and that is divisible by any divisor of these. To insure uniqueness we could (but don't) insist that the greatest common divisor be *monic*, i.e., that its highest-degree term have coefficient 1.

8.9 AN ASYMPTOTICALLY FAST ALGORITHM FOR POLYNOMIAL GCD'S

For a discussion of greatest common divisor algorithms we reverse our pattern and discuss polynomials first, since there are several extra details which must be handled when we adapt the algorithm to integers. Let $a_0(x)$ and $a_1(x)$ be the two polynomials whose greatest common divisor we wish to compute. We assume $\text{DEG}(a_1(x)) < \text{DEG}(a_0(x))$. This condition can be easily enforced as follows. If $\text{DEG}(a_0) = \text{DEG}(a_1)$, replace the polynomials a_0 and a_1 by a_1 and a_0 modulo a_1, i.e., the second and third terms of the remainder sequence, and proceed from there.

We shall break the problem into two parts. The first is to design an algorithm that obtains the last term in the remainder sequence whose degree is more than half that of a_0. Formally, let $l(i)$ be the unique integer such that $\text{DEG}(a_{l(i)}) > i$ and $\text{DEG}(a_{l(i)+1}) \leq i$. Note that if a_0 is of degree n, then $l(i) \leq n - i - 1$ on the assumption $\text{DEG}(a_1) < \text{DEG}(a_0)$, since $\text{DEG}(a_i) \leq \text{DEG}(a_{i-1}) - 1$ for all $i \geq 1$.

We now introduce a recursive procedure HGCD (half GCD) which takes polynomials a_0 and a_1, with $n = \text{DEG}(a_0) > \text{DEG}(a_1)$, and produces the matrix R_{0j} (see Section 8.8), where $j = l(n/2)$, that is, a_j is the last term of the remainder sequence whose degree exceeds half that of a_0.

The principle behind the HGCD algorithm is that quotients of polynomials of degrees d_1 and d_2, with $d_1 > d_2$, depend only on the leading $2(d_1 - d_2) + 1$ terms of the dividend and the leading $d_1 - d_2 + 1$ terms of the divisor. HGCD is defined in Fig. 8.7.

Example 8.11. Let

$$p_1(x) = x^5 + x^4 + x^3 + x^2 + x + 1$$

```
    procedure HGCD($a_0$, $a_1$):
1.  if DEG($a_1$) $\le$ DEG($a_0$)/2 then return $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
    else
        begin
2.          let $a_0 = b_0 x^m + c_0$, where $m = \lfloor \text{DEG}(a_0)/2 \rfloor$ and
                DEG($c_0$) < $m$;
3.          let $a_1 = b_1 x^m + c_1$, where DEG($c_1$) < $m$;
            comment $b_0$ and $b_1$ are the leading terms of $a_0$ and $a_1$.
                We have DEG($b_0$) = $\lceil \text{DEG}(a_0)/2 \rceil$ and DEG($b_0$) −
                DEG($b_1$) = DEG($a_0$) − DEG($a_1$);
4.          $R \leftarrow$ HGCD($b_0$, $b_1$);
5.          $\begin{bmatrix} d \\ e \end{bmatrix} \leftarrow R \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$;
6.          $f \leftarrow d$ modulo $e$;
            comment $e$ and $f$ are successive terms in the remainder
                sequence, of degree at most $\lceil 3m/2 \rceil$, that is, 3/4
                the degree of $a_0$;
7.          let $e = g_0 x^{\lfloor m/2 \rfloor} + h_0$, where DEG($h_0$) < $\lfloor m/2 \rfloor$;
8.          let $f = g_1 x^{\lfloor m/2 \rfloor} + h_1$, where DEG($h_1$) < $\lfloor m/2 \rfloor$;
            comment $g_0$ and $g_1$ are each of degree $m + 1$, at most;
9.          $S \leftarrow$ HGCD($g_0$, $g_1$);
10.         $q \leftarrow \lfloor d/e \rfloor$;
11.         return $S * \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} * R$
        end
```

Fig. 8.7. The procedure HGCD.

and

$$p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7.$$

Suppose we attempt to compute HGCD(p_1, p_2). If $a_0 = p_1$ and $a_1 = p_2$, then at lines 2 and 3 we have $m = 2$, and

$$\begin{aligned} b_0 &= x^3 + x^2 + x + 1, & c_0 &= x + 1, \\ b_1 &= x^2 - 2x + 3, & c_1 &= -x - 7. \end{aligned}$$

Then we call HGCD(b_0, b_1) at line 4. We may check that R is given the value

$$\begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix}$$

at that step. Next, at lines 5 and 6 we compute

$$\begin{aligned} d &= x^4 - 2x^3 + 3x^2 - x - 7, \\ e &= 4x^3 - 7x^2 + 11x + 22, \\ f &= -\tfrac{3}{16}x^2 - \tfrac{93}{16}x - \tfrac{45}{8}. \end{aligned}$$

We find $\lfloor m/2 \rfloor = 1$, so at lines 7 and 8 we obtain

$$\begin{aligned} g_0 &= 4x^2 - 7x + 11, \qquad & h_0 &= 22, \\ g_1 &= -\tfrac{3}{16}x - \tfrac{93}{16}, & h_1 &= -\tfrac{45}{8}. \end{aligned}$$

Thus at line 9, we find

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

At line 10, $q(x)$, the quotient $\lfloor d(x)/e(x) \rfloor$, is found to be† $\frac{1}{4}x - \frac{1}{16}$. So at line 11 we have the result

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -(\frac{1}{4}x - \frac{1}{16}) \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -(x+3) \end{bmatrix} = \begin{bmatrix} 1 & -(x+3) \\ -(\frac{1}{4}x - \frac{1}{16}) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Note that

$$T \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix},$$

which is correct since in the remainder sequence for p_1 and p_2, e is the last polynomial whose degree exceeds half that of p_1. □

Let us consider the matrix R computed at line 4 of HGCD. Presumably, R is

$$\prod_{j=1}^{l(\lceil m/2 \rceil)} \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix},$$

where $q_j(x)$ is the jth quotient of the remainder sequence for b_0 and b_1. That is, $R = R_{0,l(\lceil m/2 \rceil)}^{(b_0,b_1)}$. Yet on line 5, we used R as if it were the matrix $R_{0,l(\lceil 3m/2 \rceil)}^{(a_0,a_1)}$ to obtain d and e, where d is the last term in the remainder sequence of degree greater than $3m/2$. We must show that both these interpretations of R are correct. That is, $R_{0,l(\lceil 3m/2 \rceil)}^{(a_0,a_1)} = R_{0,l(\lceil m/2 \rceil)}^{(b_0,b_1)}$. Similarly, we must show that S, computed on line 9, may play the role assigned to it. That is,

$$S = R_{0,l(\lceil m/2 \rceil)}^{(g_0,g_1)} = R_{0,l(m)}^{(e,f)}.$$

These results are implied by the next lemmas.

Lemma 8.6. Let

$$f(x) = f_1(x)x^k + f_2(x), \tag{8.25}$$

† This computation could be done right after line 5, of course.

where $\mathrm{DEG}(f_2) < k$, and let

$$g(x) = g_1(x)x^k + g_2(x), \tag{8.26}$$

where $\mathrm{DEG}(g_2) < k$. Let q and q_1 be the quotients $\lfloor f(x)/g(x) \rfloor$ and $\lfloor f_1(x)/g_1(x) \rfloor$, and let r and r_1 be the remainders $f(x) - q(x)g(x)$ and $f_1(x) - q_1(x)g_1(x)$, respectively. If $\mathrm{DEG}(f) > \mathrm{DEG}(g)$ and $k \leq 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$ [i.e., $\mathrm{DEG}(g_1) \geq \frac{1}{2}\mathrm{DEG}(f_1)$], then

a) $q(x) = q_1(x)$ and

b) $r(x)$ and $r_1(x)x^k$ agree in all terms of degree $k + \mathrm{DEG}(f) - \mathrm{DEG}(g)$ or higher.

Proof. Consider dividing $f(x)$ by $g(x)$ using the ordinary division algorithm which divides the first term of $f(x)$ by the first term of $g(x)$ to get the first term of the quotient. The first term of the quotient is multiplied by $g(x)$ and subtracted from $f(x)$ and so on. The first $\mathrm{DEG}(g) - k$ terms produced do not depend on $g_2(x)$. But the quotient has only terms of degree $\mathrm{DEG}(f) - \mathrm{DEG}(g)$. Thus if $\mathrm{DEG}(f) - \mathrm{DEG}(g) \leq \mathrm{DEG}(g) - k$, that is, $k \leq 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$, the quotient does not depend on $g_2(x)$. If $\mathrm{DEG}(f) - \mathrm{DEG}(g) \leq \mathrm{DEG}(f) - k$, then the quotient does not depend on $f_2(x)$. But $\mathrm{DEG}(f) - \mathrm{DEG}(g) \leq \mathrm{DEG}(f) - k$ follows from $k \leq 2\mathrm{DEG}(g) - \mathrm{DEG}(f)$ and $\mathrm{DEG}(f) > \mathrm{DEG}(g)$. Thus part (a) follows. For part (b), similar reasoning shows that the remainder terms of degree $\mathrm{DEG}(f) - (\mathrm{DEG}(g) - k)$ or greater do not depend on $g_2(x)$. Similarly, terms of the remainder of degree k or greater do not depend on $f_2(x)$. But $\mathrm{DEG}(f) - \mathrm{DEG}(g) + k > k$. Thus $r(x)$ and $r_1(x)x^k$ agree in all terms of degree $\mathrm{DEG}(f) - \mathrm{DEG}(g) + k$ or higher. □

Lemma 8.7. Let $f(x) = f_1(x)x^k + f_2(x)$ and $g(x) = g_1(x)x^k + g_2(x)$, where $\mathrm{DEG}(f_2) < k$ and $\mathrm{DEG}(g_2) < k$. Let $\mathrm{DEG}(f) = n$, and $\mathrm{DEG}(g) < \mathrm{DEG}(f)$. Then

$$R^{(f,g)}_{0,l(\lceil (n+k)/2 \rceil)} = R^{(f_1,g_1)}_{0,l(\lceil n-k)2 \rceil)}.$$

That is, the quotients of the remainder sequences for (f, g) and (f_1, g_1) agree at least until the latter reaches a remainder whose degree is no more than half that of f_1.

Proof. Lemma 8.6 assures that the quotients agree, and that in the corresponding remainders of the two remainder sequences a sufficient number of the high-order terms agree. □

Theorem 8.17. Let $a_0(x)$ and $a_1(x)$ be polynomials, with $\mathrm{DEG}(a_0) = n$ and $\mathrm{DEG}(a_1) < n$. Then $\mathrm{HGCD}(a_0, a_1) = R_{0,l(n/2)}$.

Proof. The result is a straightforward induction on n, using Lemma 8.7 to insure that R at line 4 is $R^{(a_0,a_1)}_{0,l(\lceil 3m/2 \rceil)}$ and that S at line 9 is $R^{(a_0,a_1)}_{l(\lceil 3m/2 \rceil)+1,l(m)}$. □

Theorem 8.18. HGCD requires $O_A(M(n) \log n)$ time if its arguments are of degree at most n, where $M(n)$ is the time required to multiply two polynomials of degree n.

Proof. We show the result for n a power of 4. Since the time for HGCD is clearly a nondecreasing function, the theorem then follows for all n. If $\text{DEG}(a_0)$ is a power of 4, then

$$\text{DEG}(b_0) = \tfrac{1}{2}\text{DEG}(a_0) \qquad \text{and} \qquad \text{DEG}(g_0) \le \tfrac{1}{2}\text{DEG}(a_0).$$

Thus $T(n)$, the time for HGCD with input of degree n, is bounded by

$$T(n) \le 2T\left(\frac{n}{2}\right) + cM(n) \tag{8.27}$$

for some constant c. That is, the body of HGCD involves two calls to itself with half-sized arguments and a constant number of other operations which are either $O_A(n)$ or $O_A(M(n))$ in time complexity. The solution to (8.27) should be familiar; it is bounded from above by $c_1 M(n) \log n$ for some constant c_1. □

Now we proceed to the complete algorithm for greatest common divisors. It uses HGCD to calculate $R_{0,n/2}$, then $R_{0,3n/4}$, then $R_{0,7n/8}$, and so on, where n is the degree of the input.

Algorithm 8.7. GCD algorithm.

Input. Polynomials $p_1(x)$ and $p_2(x)$, where $\text{DEG}(p_2) < \text{DEG}(p_1)$.

Output. $\text{GCD}(p_1, p_2)$, the greatest common divisor of p_1 and p_2.

Method. We call the procedure $\text{GCD}(p_1, p_2)$, where GCD is the recursive procedure of Fig. 8.8. □

Example 8.12. Let us continue Example 8.11 (p. 303). There, $p_1(x) = x^5 + x^4 + x^3 + x^2 + 1$ and $p_2(x) = x^4 - 2x^3 + 3x^2 - x - 7$. We already found

$$\text{HGCD}(p_1, p_2) = \begin{bmatrix} 1 & -(x+3) \\ -(\frac{1}{4}x - \frac{1}{16}) & \frac{1}{4}x^2 + \frac{11}{16}x + \frac{13}{16} \end{bmatrix}.$$

Thus we compute $b_0 = 4x^3 - 7x^2 + 11x + 22$ and $b_1 = -\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}$ at line 3. We find that b_1 does not divide b_0. At line 5, we find

$$b_0 \text{ modulo } b_1 = 3952x + 3952.$$

Since the latter divides $-\frac{3}{16}x^2 - \frac{93}{16}x - \frac{45}{8}$, the call to GCD at line 6 terminates at line 1 and produces $3952x + 3952$ as an answer. Of course, $x + 1$ is also a greatest common divisor of p_1 and p_2. □

```
    procedure GCD($a_0$, $a_1$):
1.  if $a_1$ divides $a_0$ then return $a_1$
    else
        begin
2.          $R \leftarrow$ HGCD($a_0$, $a_1$);
3.          $\begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \leftarrow R * \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$;
4.          if $b_1$ divides $b_0$ then return $b_1$
            else
              begin
5.              $c \leftarrow b_0$ modulo $b_1$;
6.              return GCD($b_1$, $c$)
              end
        end
```

Fig. 8.8. Procedure GCD.

The correctness of Algorithm 8.7 is trivial if we can show that it terminates. Thus the correctness of the algorithm is implied by its timing analysis, the result of the next theorem.

Theorem 8.19. If $\text{DEG}(p_1) = n$, then Algorithm 8.7 requires $O_{\text{A}}(M(n) \log n)$ time, where $M(n)$ is the time needed to multiply two polynomials of degree n.

Proof. The inequality

$$T(n) \leq T\left(\frac{n}{2}\right) + c_1 M(n) + c_2 M(n) \log n, \tag{8.28}$$

where c_1 and c_2 are constants, describes the running time of Algorithm 8.7. That is, the degree of b_1 is less than half that of a_0, so the first term of (8.28) accounts for the recursive call on line 6. The term $c_1 M(n)$ accounts for the divisions and multiplications on lines 1, 3, 4, and 5, and the last term accounts for the call to HGCD on line 2. The solution to (8.28) is easily seen to be bounded from above by $kM(n) \log n$ for a constant k. □

Corollary. The GCD of two polynomials of degree at most n can be computed in $O_{\text{A}}(n \log^2 n)$ time.

8.10 INTEGER GCD'S

We shall now briefly discuss the modifications to the procedures HGCD and GCD to make them work for integers. To understand where problems arise, let us look at Lemma 8.6, which showed that when taking quotients of polynomials of degrees n and $n - d$, we have no need for terms of degree less than $n - 2d$ in either.

In analogy with Lemma 8.6, we may consider two integers f and g, with $f > g$, and write $f = f_1 2^k + f_2$ and $g = g_1 2^k + g_2$, where $f_2 < 2^k$ and $g_2 < 2^k$. In place of the condition $\mathrm{DEG}(g_1(x)) \geq \frac{1}{2}\mathrm{DEG}(f_1(x))$, we may assume $f_1 \leq (g_1)^2$. Then, we may let $f = qg + r$ and $f_1 = q_1 g_1 + r_1$. Combining these formulas, we obtain

$$g_1 2^k (q - q_1) = f_2 + r_1 2^k - q g_2 - r. \tag{8.29}$$

Since $r_1 < g_1$, $f_2 < 2^k$, and all integers are nonnegative, we may easily show from (8.29) that $q - q_1 \leq 0$. Let $q = q_1 - m$, for some $m \geq 0$. Then from (8.29) we may conclude

$$m g_1 2^k \leq q g_2 + r = (q_1 - m) g_2 + r.$$

Hence

$$mg = m g_1 2^k + m g_2 \leq q_1 g_2 + r. \tag{8.30}$$

Since $f_1 \leq (g_1)^2$, we have $q_1 \leq g_1$. Also, $g_2 < 2^k$ and $r < g$ are known, so (8.30) implies:

$$mg < g_1 2^k + g \leq 2g. \tag{8.31}$$

Now $m < 2$ follows immediately from (8.31). We conclude that either $q = q_1$ or $q = q_1 - 1$.

In the former case, there is no problem. If $q = q_1 - 1$, on the other hand, we cannot expect HGCD to work properly. Fortunately, we can show that $q_1 \neq q$ only if the quotient is the last one in the remainder sequence whose matrix is produced by HGCD. That is, if we substitute $q - q_1 = -1$ into (8.29), we have

$$r = f_2 + r_1 2^k - q g_2 + g_1 2^k. \tag{8.32}$$

Since $r < g = g_1 2^k + g_2$, we conclude from (8.32) that $r_1 2^k + f_2 < g_2(1 + q)$, or surely $r_1 < 1 + q$, that is, $r_1 < q_1$.

Thus r_1, which is the term in the remainder sequence following f_1 and g_1, must be less than f_1/g_1. Since $g_1 \geq \sqrt{f_1}$, we have $r_1 < \sqrt{f_1}$, meaning that HGCD would return a matrix involving quotients up to f_1/g_1 but no further. If this matrix is used in line 5 of HGCD (p. 304), it is possible that f computed on line 6 will not be less than $a_0^{3/4}$. However, since there was an error of only one in the last quotient, it is possible to show that extending the remainder sequence a limited amount (independent of the size of a_0) after line 6 is sufficient to bring the sequence below $a_0^{3/4}$. A similar "fixup" is needed for GCD after line 5 of that procedure.

Another source of problems concerns Lemma 8.6. With polynomials, we were able to show that the remainder sequence formed by considering leading terms of the polynomials agreed in certain high-order terms with the sequence formed from the complete polynomials. The analogous results hold approximately for integers, provided we have $q = q_1$. However, we can only limit the difference between r and $r_1 2^k$ to within $2^k(q + 1)$; we cannot be sure that any particular bits of r and $r_1 2^k$ will agree. Nevertheless, this source of "rounding error" will cause only a limited number of additional terms of the remainder sequence to be needed after line 6 of HGCD and line 5 of GCD.

Since the additional cost of extending the remainder sequence a bounded amount is $O_B(M(n))$ if $M(n)$ is the time to multiply n-bit numbers, the timing analyses of HGCD and GCD are essentially unaffected. We therefore have the following theorem.

Theorem 8.20. If $M(n)$ is the time needed to multiply two n-bit numbers, then there is an algorithm to find $\mathrm{GCD}(a_0, a_1)$ for integers a_0 and a_1 in $O_B(M(n) \log n)$ time.

Proof. Exercise based on the above suggested modification to procedures HGCD and GCD. □

Corollary. We can find integer GCD's in $O_B(n \log^2 n \operatorname{loglog} n)$ time. □

8.11 CHINESE REMAINDERING REVISITED

As promised, we shall now see how the GCD algorithm can be used to develop an asymptotically fast algorithm for the integer case of Chinese remaindering without preconditioning. Recall that the preconditioned Algorithm 8.5 requires $O_B(M(bk) \log k)$ time for reconstructing u from k moduli of b bits each. The problem is to compute $d_i = (p/p_i)^{-1}$ modulo p_i, where $p_0, p_1, \ldots, p_{k-1}$ are the moduli and p their product.

Using the obvious divide-and-conquer algorithm, we can compute p itself by first computing products of pairs of p_i's, then products of four p_i's, etc., in $O_B(M(bk) \log k)$ time. The techniques of Algorithm 8.5 enable us to compute $e_i = (p/p_i)$ modulo p_i in $O_B(M(bk) \log k)$ steps, for $0 \leq i < k$, without preconditioning. It remains to determine the time necessary to calculate $d_i = e_i^{-1}$ modulo p_i.

Since p/p_i is the product of the moduli other than p_i, it must be relatively prime to p_i. If we express p/p_i as $qp_i + e_i$ for some integer q, it follows that e_i and p_i are relatively prime, that is, $\mathrm{GCD}(e_i, p_i) = 1$. Thus, given x and y such that $e_i x + p_i y = 1$, we have $e_i x \equiv 1$ modulo p_i. It follows that $x \equiv e_i^{-1} = d_i$ modulo p_i. But the extended Euclidean algorithm computes such an x and y.

While procedure GCD was designed to produce only $\mathrm{GCD}(p_1, p_2)$, we designed HGCD to produce the matrix $R_{0,l(n/2)}$. Thus a simple modification to the GCD algorithm could allow it to produce $R_{0,l(0)}$. It is then possible to obtain x, since it will be the upper left element of this matrix. It is now possible to state the timing result for Chinese remaindering without preconditioning.

Theorem 8.21. Given k moduli of b bits each, in the integer case Chinese remaindering is $O_B(M(bk) \log k) + O_B(kM(b) \log b)$.

Proof. By the above analysis, the first term accounts for computing the c_i's and executing Algorithm 8.5. The second term accounts for computing the d_i's, since the computation of x and y may be done modulo p_i, permitting b-bit arithmetic throughout. □

Corollary. Chinese remaindering without preconditioning requires time at most $O_B[bk \log^2 bk \operatorname{loglog} bk]$.†

8.12 SPARSE POLYNOMIALS

We have been using a representation of univariate polynomials which assumes that the polynomial $\Sigma_{i=0}^{n-1} a_i x^i$ is *dense*, that is, almost all of the coefficients are nonzero. For many applications, it is useful to assume that the polynomial is *sparse*, that is, the number of nonzero coefficients is much less than the largest degree. In this situation, the logical representation of a polynomial is the list of pairs (a_i, j_i) consisting of a nonzero coefficient and its corresponding power of x.

While we cannot delve into all the techniques known for doing arithmetic on sparse polynomials, we shall mention two interesting aspects of the theory. First, it is unreasonable to use Fourier transforms to do multiplication; thus we shall give one reasonable algorithm to multiply sparse polynomials. Second, we shall exhibit a surprising difference between the way arithmetic should be done on dense and on sparse polynomials by considering the computation of $[p(x)]^4$.

The most reasonable known strategy for handling sparse polynomials when multiplications are being done is to represent a polynomial $\Sigma_{i=1}^{n} a_i x^{j_i}$ as the list of pairs $(a_1, j_1), (a_2, j_2), \ldots, (a_n, j_n)$, where we assume the j's are distinct and in decreasing order, i.e., $j_i > j_{i+1}$ for $1 \le i < n$. To multiply two polynomials represented in this way, we compute products of pairs and sort the resulting terms by their exponents (second components of the pairs), combining any terms with identical exponents. The penalty for not doing so is that our representations may have increasingly many terms with the same exponent. Thus, as more and more arithmetic is done, the cost begins to significantly exceed what it could have been if we had combined terms at each step.

† It is interesting to note that for $M(n) = n \log n \operatorname{loglog} n$, this figure is the best that can be obtained from Theorem 8.21, no matter how b and k are related.

If we multiply sorted sparse polynomials, we can take advantage of the fact that they are sorted and the fact that one may have many more terms than the other to make the sorting of the product as simple as possible. We present an informal algorithm to do multiplication of sparse polynomials in this way.

Algorithm 8.8. Multiplication of sorted sparse polynomials.

Input. Polynomials

$$f(x) = \sum_{i=1}^{m} a_i x^{j_i} \qquad \text{and} \qquad g(x) = \sum_{i=1}^{n} b_i x^{k_i},$$

represented by lists of pairs

$$(a_1, j_1), (a_2, j_2), \ldots, (a_m, j_m) \qquad \text{and} \qquad (b_1, k_1), (b_2, k_2), \ldots, (b_n, k_n),$$

where the j's and k's are monotonically decreasing.

Output

$$\sum_{i=1}^{p} c_i x^{l_i} = f(x)g(x),$$

represented by a list of pairs where the l_i's are monotonically decreasing.

Method. Without loss of generality, assume $m \geq n$.

1. Construct the sequences S_i, for $1 \leq i \leq n$, whose rth term, $1 \leq r \leq m$, is $(a_r b_i, j_r + k_i)$. That is, S_i represents the product of $f(x)$ with the ith term of $g(x)$.
2. Merge S_{2i-1} with S_{2i} for $1 \leq i \leq n/2$, combining terms. Then merge the resulting sequences in pairs, combining terms, and repeat until one sorted sequence remains. □

Theorem 8.22. Algorithm 8.8 requires $O(mn \log n)$ time,† where $m \geq n$ is assumed.

Proof. Step 1 is $O(mn)$, surely. Step 2 must be repeated $\lceil \log n \rceil$ times, and the total work at each pass is clearly $O(mn)$. □

Now let us see how Algorithm 8.8 and its time complexity affects the way sparse polynomial arithmetic should be done.

Example 8.13. Consider the computation of $p^4(x)$, where $p(x)$ is a polynomial with n terms in both the dense and sparse cases. Given that p is dense, it is easy to show that the best way to compute $p^4(x)$ is by two squarings. That is.

† Note we are using complexity on a RAM rather than arithmetic complexity here, since branches are inherent in the program for Algorithm 8.8, even if m and n are fixed.

assume $M(n)$, the time to multiply dense polynomials, is $cn \log n$. Then we can compute $p^2(x)$ in $cn \log n$ steps and square the result in $2cn \log 2n$ steps, for a total cost of $3cn \log n + 2cn$ steps. In comparison, if we compute $p^4 = p \times (p \times (p \times p))$, the time required is easily seen to be $6cn \log n$, on the assumption that the multiplication $p \times p^2$ requires $2M(n)$ steps and the multiplication $p \times p^3$ requires $3M(n)$ steps. Thus, for dense polynomials, we make the expected observation that p^4 should be calculated by repeated squaring.

Now suppose instead that $p(x)$ is a sparse polynomial with n terms. If we compute $(p^2)^2$ using Algorithm 8.8, the time required is $cn^2 \log n$ for the first squaring and, assuming that few terms can be combined, $cn^4 \log n^2$ for the second, a total of $c(2n^4 + n^2) \log n$. On the other hand, computation of $p \times (p \times (p \times p))$ requires $cn^2 \log n + cn^3 \log n + cn^4 \log n = c(n^4 + n^3 + n^2) \log n$. This figure is less than the time to square twice. Thus repeated squaring of sparse polynomials is not always a good way to compute p^4. The effect becomes more pronounced if we consider computation of p^{2^t} for large integers t. □

EXERCISES

8.1 Use Algorithm 8.1 to find the "reciprocal" of 429.

8.2 Use Algorithm 8.2 to find 429^2.

8.3 Use Algorithm 8.3 to find the "reciprocal" of

$$x^7 - x^6 + x^5 - 3x^4 + x^3 - x^2 + 2x + 1.$$

***8.4** Give an algorithm analogous to Algorithm 8.2 to compute squares of polynomials.

8.5 Use your algorithm from Exercise 8.4 to compute $(x^3 - x^2 + x - 2)^2$.

8.6 Use Algorithm 8.4 to find the representation for one million when the moduli are 2, 3, 5, 7, 11, 13, 17, 19.

8.7 Write a complete algorithm to find residues of a polynomial modulo a collection of polynomial moduli.

8.8 Find the residues of $x^7 + 3x^6 + x^5 + 3x^4 + x^2 + 1$ modulo $x + 3$, $x^3 - 3x + 1$, $x^2 + x - 2$, and $x^2 - 1$.

8.9 The polynomials in Exercise 8.8 were carefully selected to make hand computation feasible. Select at random four polynomials of degrees 1, 2, 3, and 4 and find the residues of $x^9 - 4$ with respect to the four polynomials. What happens?

8.10 Let 5, 6, 7, 11 be four moduli. Find that u less than their product such that $u \leftrightarrow (1, 2, 3, 4)$.

***8.11** Generalize Lemma 8.3 to apply to arbitrary polynomial moduli whose roots are known or can be found (e.g., polynomials of degree less than 5). What is the complexity of polynomial Chinese remaindering by use of your algorithm, as a function of the number and degree of the moduli?

8.12 Find a polynomial whose values at 0, 1, 2, 3 are, respectively, 1, 1, 2, 2.

8.13 Find the greatest common divisor of

$$x^6 + 3x^5 + 3x^4 + x^3 - x^2 - x - 1,$$
$$x^6 + 2x^5 + x^4 + 2x^3 + 2x^2 + x + 1.$$

8.14 The polynomials in Exercise 8.13 were carefully selected to make hand computation feasible. Select two arbitrary polynomials of degrees 7 and 8 respectively and attempt to find their GCD. What happens? What do you suspect the GCD would turn out to be?

***8.15** Give a complete algorithm to find integer GCD's that runs in $O_B(n \log^2 n \operatorname{loglog} n)$ time on n-bit integers.

8.16 Use your algorithm from Exercise 8.15 to find GCD(377,233).

***8.17** Suppose $p(x)$ is a sparse nth-degree polynomial. Determine, as a function of n, the best method of evaluating $p^8(x)$ if multiplication is by Algorithm 8.8.

****8.18** The following method of computing $f(x)/g(x)$ for polynomials f and g, on the assumption that g divides f, is proposed. Let f and g be of $(n - 1)$st degree or less.

(1) Compute F and G, the discrete Fourier transforms of f and g, respectively.

(2) Divide the terms of F by corresponding terms in G to yield sequence H.

(3) Take the inverse transform of H. The result is f/g. Will this algorithm work?

****8.19** Let $M(n)$ be the time to multiply n-bit numbers and $Q(n)$ the time to find $\lfloor\sqrt{i}\rfloor$ for an n-bit integer i. Assume $M(an) \geq aM(n)$ for $a \geq 1$ and similarly for $Q(n)$. Show that $M(n)$ and $Q(n)$ are within a constant factor of one another.

***8.20** Generalize Exercise 8.19 to (a) polynomials, (b) rth roots for fixed r.

****8.21** Give an $O_A(n \log n)$ algorithm to evaluate an $(n - 1)$st-degree polynomial and all its derivatives at one point.

****8.22** A dense polynomial in r variables† can be represented as

$$\sum_{i_1=0}^{n-1} \sum_{i_2=0}^{n-1} \cdots \sum_{i_r=0}^{n-1} a_{i_1 i_2 \cdots i_r} x_1^{i_1} x_2^{i_2} \ . \ . \ . \ x_r^{i_r}.$$

Show that by evaluating and interpolating these polynomials at points $x_1 = \omega^{j_1}$, $x_2 = \omega^{j_2}, \ldots, x_r = \omega^{j_r}$, for $0 \leq j_k < 2n$, where ω is the principal $2n$th root of unity, we can multiply such polynomials in $O_A(n^r \log n)$.

****8.23** Show that under reasonable assumptions about the smoothness of $M(n)$ and $D(n)$, the times needed to multiply and divide dense polynomials in r variables, $M(n)$ and $D(n)$, are within a constant factor of one another.

****8.24** Find an $O_B(n \log^2 n \operatorname{loglog} n)$ algorithm to convert (a) n-bit binary numbers to decimal; (b) n-place decimal number to binary.

† As r gets large, the dense assumption becomes progressively less useful.

****8.25** The *least common multiple* (*LCM*) of integers (polynomials) x and y is that z (some z in the polynomial case) divisible by x and y which divides any other integer (polynomial) also divisible by x and y. Show that the time to find the LCM of two n-bit numbers is at least as great as the time to multiply n-bit numbers.

8.26 Does the method of Section 2.6 for integer multiplication yield an $O_A(n^{1.59})$ time polynomial multiplication algorithm?

****8.27** Show that we may evaluate an $(n-1)$st-degree polynomial at the points $a^0, a^1, \ldots, a^{n-1}$ in $O_A(n \log n)$ steps. [*Hint:* Show that $c_j = \sum_{i=0}^{n-1} b_i a^{ij}$ can be written as $c_j = \sum_{i=0}^{n-1} f(i) g(j-i)$ for some functions f and g.]

****8.28** Show that we may evaluate an $(n-1)$st-degree polynomial at the n points $ba^{2j} + ca^j + d$, for $0 \le j < n$, in $O_A(n \log n)$ steps.

8.29 Give recursive versions of Algorithms 8.4 and 8.5.

Research Problems

8.30 In this chapter, we have shown a number of polynomial and integer problems to be

i) essentially as complex as multiplication, or
ii) at most a log factor more complex than multiplication.

Some of these problems appear in the exercises of this chapter. Exercise 9.9 gives another problem in group (ii)—the problem of "and-or" multiplication.

A reasonable research problem is to add to the set of problems in either class (i) or (ii). Another area is to show that some of the problems in group (ii) must necessarily be of the same complexity. For example, one might conjecture that this is true for GCD's and LCM's.

8.31 Another problem which appears deceptively simple is to determine whether Algorithm 8.8 is the best that can be done to multiply sorted sparse polynomials and sort the result. Johnson [1974] has considered some aspects of this problem.

8.32 The value of n for which many of the algorithms described here become practical is enormous. However, we could carefully combine them with the $O(n^{1.59})$ methods mentioned in Section 2.6 and Exercise 8.26 for some small n's and the obvious $O(n^2)$ method for the smallest values of n. Obtain in this way upper bounds on the values of n for which the problems of this chapter have better algorithms than the obvious ones. Some work along these lines has been done by Kung [1973].

BIBLIOGRAPHIC NOTES

Theorem 8.2, the fact that finding reciprocals is no harder than multiplication, is from Cook [1966]. Curiously, the fact that the algorithm has a polynomial analog was not realized for several years. Moenck and Borodin [1972] gave an $O_B(n \log^2 n \log\log n)$ algorithm for division, and shortly thereafter the $O_B(n \log n \log\log n)$ division algorithm was observed independently by several people, including Sieveking [1972].

The development of algorithms for modular arithmetic, and for interpolation and evaluation of polynomials follows the lines of Moenck and Borodin [1972]. An $O_B(n \log^2 n \operatorname{loglog} n)$ preconditioned algorithm for Chinese remaindering was found by Heindel and Horowitz [1971]. Borodin and Munro [1971] gave an $O(n^{1.91})$ multipoint polynomial evaluation algorithm, and Horowitz [1972] extended this to interpolation. Kung [1973] developed an $O_B(n \log^2 n)$ polynomial evaluation algorithm without using a fast $(n \log n)$ division algorithm. The unity of Chinese remaindering, interpolation, the evaluation of polynomials, and the computation of integer residues is expressed in Lipson [1971].

The $O_B(n \log^2 n \operatorname{loglog} n)$ integer GCD algorithm is by Schönhage [1971]. It was adapted to polynomials and general Euclidean domains by Moenck [1973].

A survey of classical techniques for GCD's can be found in Knuth [1969]. A sampling of the material on the complexity of sparse polynomial arithmetic can be found in Brown [1971], Gentleman [1972] and Gentleman and Johnson [1973]. Additional results on computer implementation of polynomial arithmetic can be found in Hall [1971], Brown [1973], and Collins [1973]. Algorithm 8.8 with a heap data structure has been implemented by S. Johnson in ALTRAN [Brown, 1973].

Exercises 8.19 and 8.20 are due to R. Karp and J. Ullman. Exercise 8.21 on the evaluation of a polynomial and its derivatives was observed by Vari [1974] and Aho, Steiglitz, and Ullman [1974], independently. Exercise 8.28 is from the latter. Exercise 8.27 is due to Bluestein [1970] and Rabiner, Schafer, and Rader [1969]. An expanded treatment of polynomial and integer arithmetic is found in Borodin and Munro [1975].

PATTERN-MATCHING ALGORITHMS

CHAPTER 9

Pattern matching is an integral part of many text-editing, data retrieval, and symbol manipulation problems. In a typical string-matching problem we are given a text string x and a set of pattern strings $\{y_1, y_2, \ldots\}$. The problem is to locate either an occurrence or all occurrences of pattern strings in x. The set of patterns is often a regular set specified by means of a regular expression. In this chapter we present several techniques for solving pattern-matching problems of this nature.

We begin by reviewing regular expressions and finite automata. Next we give an algorithm for finding in a string x an occurrence of any substring y which is in a set denoted by a given regular expression. The algorithm runs in time on the order of the product of the length of x and the length of the regular expression. We then present a linear algorithm for determining whether a given string y is a substring of another given string x. There then follows a powerful theoretical result which states that any pattern recognition problem which can be solved by a two-way deterministic pushdown automaton can be solved in linear time on a RAM. The result is remarkable since the pushdown automaton may take quadratic or even exponential time to solve the problem. Finally, we introduce the concept of a position tree and apply it to some other pattern-matching problems such as finding the longest repeated substrings in a given string.

9.1 FINITE AUTOMATA AND REGULAR EXPRESSIONS

Many pattern recognition problems and their solutions can be expressed in terms of regular expressions and finite automata. Thus we begin with a review of this material.

Definition. An *alphabet* is a finite set of symbols. A *string* over an alphabet I is a finite-length sequence of symbols from I. The *empty string*, denoted by ϵ, is the string with no symbols. If x and y are strings, then the *concatenation* of x and y is the string xy. If xyz is a string, then x is a *prefix*, y a *substring*, and z a *suffix* of xyz. The *length* of a string x, denoted $|x|$, is the total number of symbols in x. For example, the string aab is of length 3; ϵ is of length 0.

A *language* over an alphabet I is a set of strings over I. Let L_1 and L_2 be two languages. The language L_1L_2, called the *concatenation* of L_1 and L_2, is $\{xy|x \in L_1 \text{ and } y \in L_2\}$.

Let L be a language. Then define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The *Kleene closure* of L, denoted L^*, is the language $L^* = \bigcup_{i=0}^{\infty} L^i$ and the *positive closure* of L, denoted L^+, is the language $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Regular expressions and the languages they denote (*regular sets*) are a useful concept in many areas of computer science. In this chapter, we shall find that they are useful descriptors of patterns.

Definition. Let I be an alphabet. The *regular expressions over I* and the languages that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3. For each a in I, a is a regular expression and denotes the set $\{a\}$.
4. If p and q are regular expressions denoting the regular sets P and Q, respectively, then $(p + q)$, (pq), and (p^*) are regular expressions that denote the sets $P \cup Q$, PQ, and P^*, respectively.

In writing a regular expression we can omit many parentheses if we assume that * has higher precedence than concatenation or +, and that concatenation has higher precedence than +. For example, $((0(1^*)) + 0)$ may be written $01^* + 0$. We also abbreviate the expression pp^* to p^+.

Example 9.1

1. 01 is a regular expression that denotes $\{01\}$.
2. $(0 + 1)^*$ denotes $\{0, 1\}^*$.
3. $1(0 + 1)^*1 + 1$ denotes the set of all strings beginning and ending with a 1. □

A language is said to be *regular* if and only if it can be denoted by a regular expression. Two regular expressions α and β are said to be *equivalent*, denoted by $\alpha = \beta$, if they denote the same set. For example, $(0 + 1)^* = (0^*1^*)^*$.

The concept of a deterministic finite automaton was introduced in Chapter 4 (p. 143). It may be viewed as a device consisting of a "control" which is always in one of a finite set of states and an input tape which is scanned from left to right by a tape head. The deterministic finite automaton makes "moves" determined by the current state of the control and input symbol under the input head. Each move consists of entering a new state and shifting the input head one square to the right. An important fact about finite automata is that a language can be represented by a regular expression if and only if it can be accepted by a finite automaton.

A major generalization is the nondeterministic finite automaton. For each state and input symbol, a nondeterministic finite automaton has zero or more choices of next move. It may also have the choice of changing state with no shift of the input head.

Definition. A *nondeterministic finite automaton* (NDFA) M is a 5-tuple (S, I, δ, s_0, F), where

1. S is the finite set of states of the control.
2. I is the alphabet from which input symbols are chosen.
3. δ is the *state transition function* which maps $S \times (I \cup \{\epsilon\})$ to the set of subsets of S.
4. s_0 in S is the *initial state* of the finite control.
5. $F \subseteq S$ is the set of *final* (or *accepting*) *states*.

An *instantaneous description* (ID) of an NDFA M is a pair (s, w), where $s \in S$ represents the state of the finite control and $w \in I^*$ represents the unused portion of the input string (the symbol under the input tape head followed by those to the right). An *initial* ID of M is an ID in which the first component is the initial state and the second component is the string to be recognized, i.e., (s_0, w) for some $w \in I^*$. An *accepting* ID is one of the form (s, ϵ), where $s \in F$.

We represent moves of an NDFA by a binary relation $\vdash$ on ID's. If $\delta(s, a)$ contains s', then we write $(s, aw) \vdash (s', w)$ for all w in I^*. Note that a may be either ϵ or a symbol of I. If $a = \epsilon$, then the state transition may be made independently of the input symbol scanned. If a is not ϵ, then a must appear on the next input square and the input head is shifted one square to the right.

We use $\overset{*}{\vdash}$ to denote the reflexive and transitive closure of $\vdash$. We say w is *accepted* by M if $(s_0, w) \overset{*}{\vdash} (s, \epsilon)$ for some s in F. That is, an input string w is accepted by M if there is some sequence of zero or more moves by which M can go from the initial ID (s_0, w) to an accepting ID (s, ϵ). The set of strings accepted by M, denoted $L(M)$, is the *language accepted* by M.

Example 9.2. Consider a nondeterministic finite automaton M that accepts all strings of a's and b's that end in aba. That is, $L(M) = (a + b)^*aba$. Let $M = (\{s_1, s_2, s_3, s_4\}, \{a, b\}, \delta, s_1, \{s_4\})$, where δ is defined as in Fig. 9.1. (The ϵ-transitions here are not necessary.)

Suppose the input to M is $ababa$. M can trace out the sequences of ID's shown in Fig. 9.2. Since $(s_1, ababa) \overset{*}{\vdash} (s_4, \epsilon)$ and since s_4 is a final state, the string $ababa$ is accepted by M. □

Associated with an NDFA is a directed graph which naturally represents its state transition function.

Definition. Let $M = (S, I, \delta, s_0, F)$ be an NDFA. The *transition diagram* associated with M is a directed graph $G = (S, E)$ with labeled edges. The set E of edges and the labels are defined as follows. If $\delta(s, a)$ contains s' for some a in $I \cup \{\epsilon\}$, then the edge (s, s') is in E. The label of (s, s') is the set of b in $I \cup \{\epsilon\}$ such that $\delta(s, b)$ contains s'.

	Input		
State	a	b	ϵ
s_1	$\{s_1, s_2\}$	$\{s_1\}$	$\emptyset$
s_2	$\emptyset$	$\{s_3\}$	$\emptyset$
s_3	$\{s_4\}$	$\emptyset$	$\{s_1\}$
s_4	$\emptyset$	$\emptyset$	$\{s_2\}$

Fig. 9.1. State transition function δ.

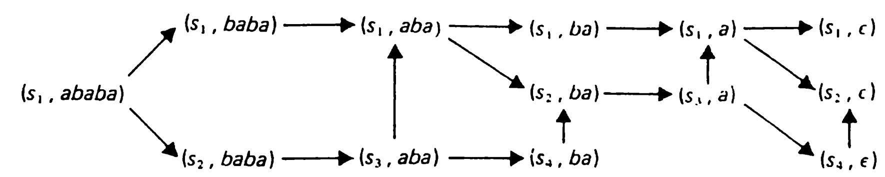


Fig. 9.2 Sequences of ID's for input *ababa*.

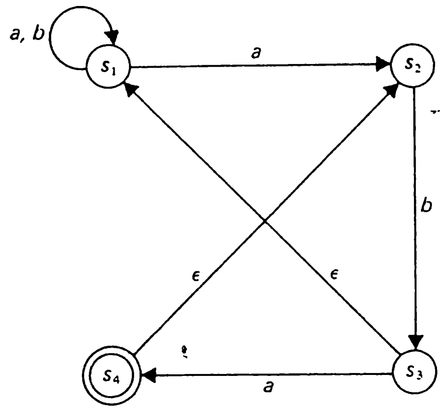


Fig. 9.3 Transition diagram for Example 9.2.

Example 9.3. The transition diagram for the NDFA M of Example 9.2 is shown in Fig. 9.3. Final states are doubly circled. □

We can relate transition diagrams of NDFA's and path problems in graphs by means of a certain closed semiring. Let I be an alphabet and let $S_I = (\mathscr{P}(I^*), \cup, \cdot, \emptyset, \{\epsilon\})$. From Section 5.6 we know that S_I is a closed semiring in which $\mathscr{P}(I^*)$ is the set of all languages over I, $\emptyset$ is the identity under union, and $\{\epsilon\}$ is the identity under $\cdot$ (concatenation).

Theorem 9.1. Each language accepted by a nondeterministic finite automaton is a regular set.

Proof. Let $M = (S, I, \delta, s_0, F)$ be an NDFA and let $G = (S, E)$ be the corresponding transition diagram. Using Algorithm 5.5, we can compute for each pair of vertices s and s' in the transition diagram, the language $L_{ss'}$, which is the set of all strings labeling paths from s to s'. We see that the label of each edge of a transition diagram is a regular expression. Moreover, if the sets C_{ij}^{k-1} computed in Algorithm 5.5 have regular expressions, then by line 5 of the algorithm, the sets C_{ij}^{k} will also have regular expressions. Thus each language $L_{ss'}$ can be denoted by a regular expression, and hence is a regular set. $L(M) = \cup_{s \in F} L_{s_0 s}$ is a regular set, since, by definition, the union of regular sets is regular. □

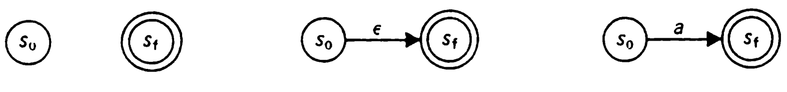
The converse of Theorem 9.1 is also true. That is, given a regular expression, there is an NDFA accepting the language denoted. What will turn out to be most important from the computational complexity point of view is that we can find such an NDFA with no more states than twice the length of the regular expression and such that no state has more than two successors.

> **Theorem 9.2.** Let α be a regular expression. Then there exists an NDFA $M = (S, I, \delta, s_0, \{s_f\})$ accepting the language denoted by α having the following properties:
>
> 1. $\|S\| \leq 2|\alpha|$, where $|\alpha|$ denotes the length of α.†
> 2. For each a in $I \cup \{\epsilon\}$, $\delta(s_f, a)$ is empty.
> 3. For each s in S, the sum of $\|\delta(s, a)\|$ over all a in $I \cup \{\epsilon\}$ is at most 2.

Proof. The proof is by induction on the length of α. For the basis, $|\alpha| = 1$, α must be of one of the three forms (a) $\emptyset$, (b) ϵ, or (c) a for some $a \in I$. The three two-state automata shown in Fig. 9.4 accept the denoted languages and satisfy the conditions of the theorem.

For the inductive step, α must be of one of the four forms (a) $\beta + \gamma$, (b) $\beta\gamma$, (c) β^*, or (d) (β) for regular expressions β and γ. In case (d), α and β denote the same language, so the induction is immediate. For the other cases, let M' and M'' be NDFA's accepting the languages of β and γ, respectively and having disjoint sets of states. Let their initial states be s_0' and s_0'' and their final states be s_f' and s_f''. Then the transition diagrams in cases (a), (b), and (c) are as shown in Fig. 9.5.

Let the lengths of α, β, and γ be $|\alpha|$, $|\beta|$, and $|\gamma|$, respectively. Let n, n' and n'' be the numbers of states in M, M', and M''. In case (a), we have $|\alpha| = |\beta| + |\gamma| + 1$ and $n = n' + n'' + 2$. By the inductive hypothesis, $n' \leq 2|\beta|$ and $n'' \leq 2|\gamma|$, so $n \leq 2|\alpha|$. Also, the only edges added in case (a) are two out of s_0, which satisfies the constraint of at most two edges leaving any vertex, and one each out of s_f' and s_f''. Since by hypothesis, s_f' and s_f'' each had



expressions of length 1: (a) foi $\emptyset$, (b) for $\{\epsilon\}$, (c) for $\{a\}$.

† The length of a regular expression α is the number of symbols in the string α. For example, $|\epsilon^*| = 2$. We could strengthen this statement by ignoring parentheses when counting the length of α [e.g., under these circumstances the regular expression $(a^*b^*)^*$ would be of "length" 5 rather than 7].

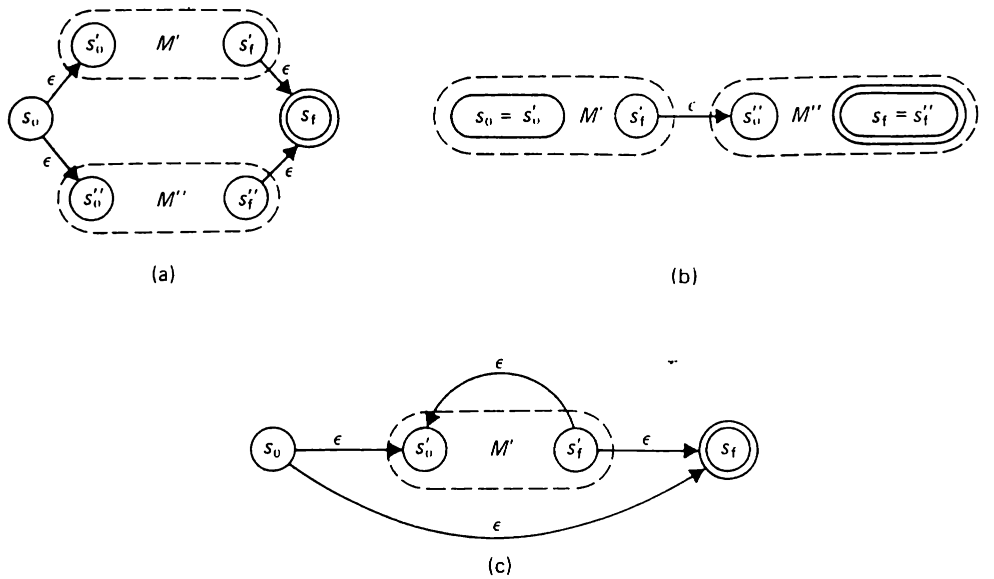


Fig. 9.5 Accepting the languages denoted by increasingly longer regular expressions: (a) for $\beta + \gamma$; (b) for $\beta\gamma$; (c) for β^*.

zero edges leaving before, the constraint on edges leaving is satisfied for s'_f and s''_f. Finally, s_f clearly has no edges leaving. Case (b) is similarly checked.†

For case (c), we have $|\alpha| = |\beta| + 1$ and $n = n' + 2$. Since $n' \leq 2|\beta|$, it follows that $n \leq 2|\alpha|$. The constraint that no more than two edges leave any vertex is easy to check. □

Example 9.4. Let us construct an NDFA for the regular expression $ab^* + c$, which is of length 5. The NDFA's for a, b, and c are as in Fig. 9.4(c). Using the construction shown in Fig. 9.5(c), we construct the automaton for b^* as shown in Fig. 9.6(a). Then, using the construction of Fig. 9.5(b), we construct for ab^* the automaton shown in Fig. 9.6(b). Finally, we use the construction of Fig. 9.5(a) to construct the NDFA for $ab^* + c$. This automaton, which has 10 states, is shown in Fig. 9.6(c). □

One additional result needs mentioning. Given any NDFA, we can find an equivalent "deterministic" machine. The deterministic finite automaton may, however, have as many as 2^n states, given an n-state NDFA, so conversion to a deterministic finite automaton is not always an efficient way to simulate a nondeterministic finite automaton. Nevertheless, the deterministic finite automaton finds use in pattern recognition, and we recall the definition here.

† In fact, the edge from s'_f to s''_0 is not necessary. We could identify s'_f and s''_0 instead. Similarly, in Fig. 9.5(a), s'_f and s''_f could be identified and made final.

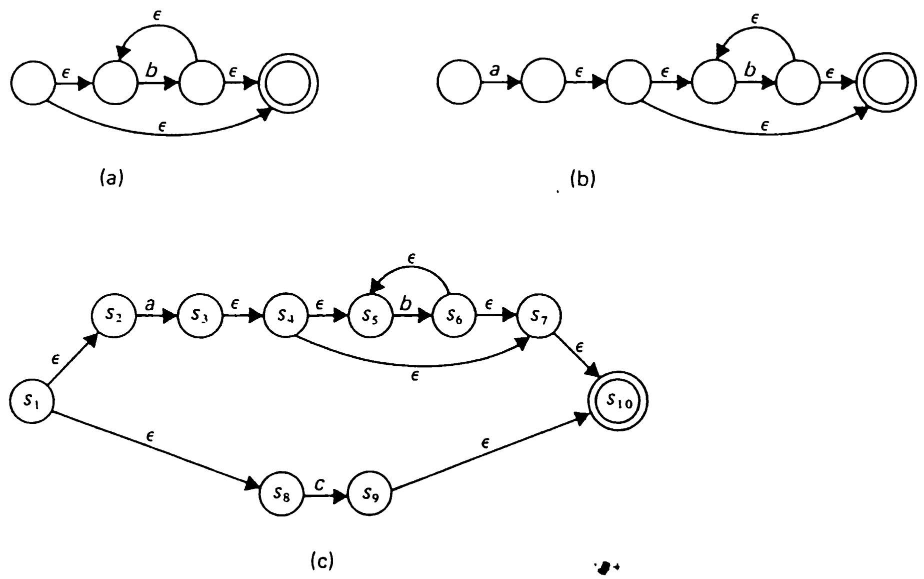


Fig. 9.6 Construction of an NDFA for $ab^* + c$: (a) for b^*; (b) for ab^*; (c) for $ab^* + c$.

Definition. A *deterministic finite automaton* (DFA) is a nondeterministic finite automaton (S, I, δ, s_0, F) such that

1. $\delta(s, \epsilon) = \emptyset$ for all $s \in S$, and
2. $\|\delta(s, a)\| \leq 1$ for all $s \in S$ and $a \in I$.

Theorem 9.3. If L is a regular set, then L is accepted by a DFA.

Proof. We know by Theorem 9.2 that L is accepted by an NDFA $M = (S, I, \delta, s_0, \{s_f\})$. We convert M to a DFA, as follows. First, we find those pairs of states (s, t) such that $(s, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon)$. To do so, we first construct a directed graph $G = (S, E)$, with edge (s, t) if and only if $\delta(s, \epsilon)$ contains t. Then we compute $G' = (S, E')$, the reflexive and transitive closure of G. We claim that $(s, \epsilon) \overset{*}{\underset{M}{\vdash}} (t, \epsilon)$ if and only if (s, t) is in E'.

We now construct an NDFA $M' = (S', I, \delta', s_0, F)$, such that $L(M') = L(M)$ and M' has no ϵ-transitions, as follows.

i) $S' = \{s_0\} \cup \{t | \delta(s, a)$ contains t for some s in S and a in $I\}$.

ii) For each s in S' and a in I,

$$\delta'(s, a) = \{u | (s, t) \text{ is in } E' \text{ and } \delta(t, a) \text{ contains } u\}.$$

iii) $F' = \{s | (s, f)$ is in E' and f is in $F\}$.

We leave it as an exercise to show that $L(M) = L(M')$. Surely M' has no transitions on ϵ.

Next, we construct from M' a DFA M'' whose set of states is the power set of S'. That is, $M'' = (\mathscr{P}(S'), I, \delta'', \{s_0\}, F'')$, where

i) for each subset S of S' and a in I,

$$\delta''(S, a) = \{t \mid \delta'(s, a) \text{ contains } t \text{ for some } s \text{ in } S\},$$

and

ii) $F'' = \{S \mid S \cap F \neq \emptyset\}$.

We leave it as an exercise to show by induction on $|w|$ that $(\{s_0\}, w) \overset{*}{\underset{M''}{\vdash}} (S, \epsilon)$ if and only if $S = \{t \mid (s_0, w) \overset{*}{\underset{M'}{\vdash}} (t, \epsilon)\}$. Hence $L(M) = L(M') = L(M'')$. □

Example 9.5. Consider the NDFA M of Fig. 9.7. The initial state s_1 can reach s_3 and the final state s_4 along paths labeled ϵ. Thus, in computing G', the reflexive and transitive closure of the directed graph G mentioned in the proof of Theorem 9.3, we must add edge (s_1, s_4). The entire graph G' is shown in Fig. 9.8. From M and G' we construct the NDFA M' shown in Fig. 9.9. Since there is an edge entering s_4 from every vertex of G', we make all states in M' final. Since the only edge entering s_3 in M is labeled ϵ, s_3 does not appear in M'.

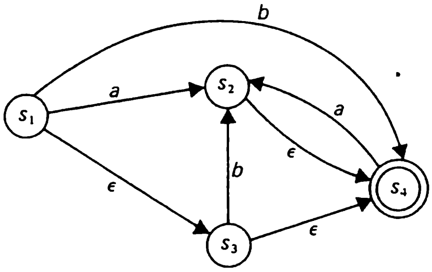


Fig. 9.7 The NDFA M of Example 9.5.

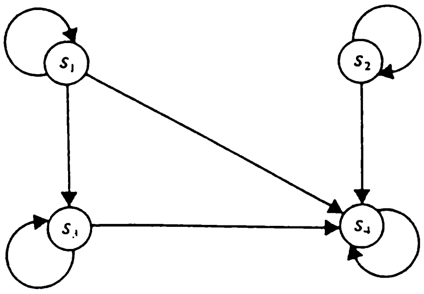


Fig. 9.8 The graph G'.

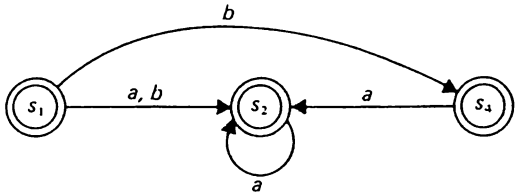


Fig. 9.9 The NDFA M'.

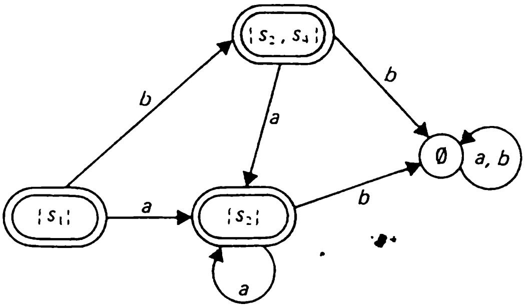


Fig. 9.10 The DFA M''.

In constructing the DFA M'' from M', we construct eight states. However, only four can be reached from the initial state and thus the other four can be deleted. The resulting DFA M'' is shown in Fig. 9.10. □

9.2 RECOGNITION OF REGULAR EXPRESSION PATTERNS

Consider a pattern recognition problem in which we are given a *text string* $x = a_1a_2 \cdots a_n$ and a regular expression α, called the *pattern*. We wish to find the least j, and given that j, some i such that the substring $a_ia_{i+1} \cdots a_j$ of x is in the language denoted by α.

Questions of this nature are very common in text-editing applications. Many text-editing programs allow a user to specify replacements in a string of text. For example, a user may specify that he wants to replace a word y by some other word in a piece of text x. To process such a command the text-editing program must be capable of locating an occurrence of the word y as a substring of x. Some sophisticated text editors allow the user to specify a regular set as a set of possible strings to be replaced. For example, a user might say: "Replace [I^*] in x by the empty string," meaning that a pair of brackets and their intervening symbols are to be erased from x.

The problem above can be reformulated by replacing the given regular expression α by the expression $\beta = I^*\alpha$, where I is the alphabet of the text string. We can find the first occurrence of α in $x = a_1a_2 \cdots a_n$ by finding the shortest prefix of x which is in the language denoted by β. This problem can be solved by first constructing an NDFA M to recognize the set denoted by β and then applying Algorithm 9.1 (below) to determine the sequence of sets of states S_i in which the NDFA can be after reading $a_1a_2 \cdots a_i$ for $i = 1, 2, \ldots, n$. As soon as S_j contains a final state, we know that $a_1a_2 \cdots a_j$ has a suffix $a_ia_{i+1} \cdots a_j$ such that $a_ia_{i+1} \cdots a_j$ is in the language denoted by α for some $1 \leq i \leq j$. Techniques for finding i, the left end of the pattern, are discussed in Exercises 9.6–9.8.

One way to simulate the behavior of the NDFA M on the text string x is to convert the NDFA into a deterministic finite automaton as in Theorem 9.3. However, this method might be quite costly, since we could go from a regular expression β to an NDFA with $2|\beta|$ states and then to a DFA with nearly $2^{2|\beta|}$ states. Just constructing the DFA can be prohibitive.

Another way to simulate the behavior of the NDFA M on string x is to first eliminate ϵ-transitions from M and thereby construct an ϵ-free NDFA M' as we did in Theorem 9.3. Then, we can simulate the NDFA M' on the input string $x = a_1a_2 \cdots a_n$ by computing for each i, $1 \leq i \leq n$, the set of states S_i in which M' could be after reading $a_1a_2 \cdots a_i$. Each S_i is actually the state in which the DFA M'' of Theorem 9.3 would be after reading $a_1a_2 \cdots a_i$.

With this technique, we do not need to construct M''; rather we compute only those states of M'' that arise in processing the string x. To explain how the set S_i can be calculated, we must show how to construct S_i from S_{i-1}. It is easy to see that

$$S_i = \bigcup_{s \in S_{i-1}} \delta'(s, a_i),$$

where δ' is the next state transition function of M'. Then S_i is the union of up to $2|\beta|$ sets each of which contains at most $2|\beta|$ members. Since we must eliminate duplicate members when taking unions (otherwise the representation of sets can become cumbersome) the obvious simulation of M' takes $O(|\beta|^2)$ steps per input symbol, or $O(n|\beta|^2)$ steps for the entire simulation.

Surprisingly, in many practical cases it turns out to be far more efficient not to eliminate the ϵ-transitions, but rather to work directly from the NDFA M constructed by Theorem 9.2 from the regular expression β. The crucial property introduced in Theorem 9.2 is that each state of M has at most two edges leaving in its transition diagram. This property enables us to show that Algorithm 9.1 (below) requires $O(n|\beta|)$ steps to simulate the NDFA constructed from β on input string $x = a_1a_2 \cdots a_n$.

Algorithm 9.1. Simulation of a nondeterministic finite automaton.

Input. An NDFA $M = (S, I, \delta, s_0, F)$ and a string $x = a_1a_2 \cdots a_n$ in I^*.

Output. The sequence of states $S_0, S_1, S_2, \ldots, S_n$ such that

$$S_i = \{s \mid (s_0, a_1a_2 \cdots a_i) \overset{*}{\vdash} (s, \epsilon)\},\ 0 \leq i \leq n.$$

```
 1.  for $i \leftarrow 0$ until $n$ do
         begin
 2.          if $i = 0$ then $S_i \leftarrow \{s_0\}$
 3.          else $S_i \leftarrow \bigcup_{s \in S_{i-1}} \delta(s, a_i)$;
             comment $S_i$ has not yet reached its final value. It now corre-
               sponds to the set $T_i$ mentioned above;
 4.          mark each $t$ in $S_i$ "considered";
 5.          mark each $t$ in $S - S_i$ "unconsidered";
 6.          QUEUE $\leftarrow S_i$;
 7.          while QUEUE not empty do
                 begin
 8.                  find and delete $t$, the first element of QUEUE;
 9.                  for each $u$ in $\delta(t, \epsilon)$ do
10.                      if $u$ is "unconsidered" then
                             begin
11.                              mark $u$ "considered";
12.                              add $u$ to QUEUE and to $S_i$
                             end
                 end
         end
```

Fig. 9.11. Simulation of a nondeterministic finite automaton.

Method. To compute S_i from S_{i-1}, first find the set of states $T_i = \{t | \delta(s, a_i)$ contains t for some s in $S_{i-1}\}$. Then compute the "closure" of T_i by adding to T_i all those states u such that $\delta(t, \epsilon)$ contains u for some t previously added to T_i. The closure of T_i, which is S_i, is computed by constructing a queue of states t in T_i for which $\delta(t, \epsilon)$ has not yet been examined. The algorithm is shown in Fig. 9.11. □

Example 9.6. Let M be the NDFA of Fig. 9.6(c) (p. 324). Suppose the input is $x = ab$. Then $S_0 = \{s_1\}$ at line 2. At lines 9–12, s_1 causes s_2 and s_8 to be added to both QUEUE and S_0. Consideration of s_2 and s_8 adds nothing. Thus $S_0 = \{s_1, s_2, s_8\}$. Then at line 3, $S_1 = \{s_3\}$. Considering s_3 adds s_4 to S_1, and considering s_4 adds s_5 and s_7. Consideration of s_5 adds nothing, but consideration of s_7 adds s_{10}. Thus $S_1 = \{s_3, s_4, s_5, s_7, s_{10}\}$. Then S_2 is set to $\{s_6\}$ at line 3 Consideration of s_6 adds s_5 and s_7 to S_2. Consideration of s_7 adds s_{10}. Thus $S_2 = \{s_5, s_6, s_7, s_{10}\}$. □

Theorem 9.4. Algorithm 9.1 correctly computes the sequence of states $S_0, S_1, \ldots, S_n$, where $S_i = \{s | (s_0, a_1a_2 \cdots a_i) \overset{*}{\vdash} (s, \epsilon)\}$.

Proof. The correctness of Algorithm 9.1 is a simple inductive exercise, and we leave it to the reader. □

Theorem 9.5. Suppose M is such that its transition diagram has no more than e edges leaving any vertex, and suppose M has m states. Then Algorithm 9.1 takes $O(emn)$ steps on an input string of length n.

Proof. Consider the calculation of S_i for one particular value of i. Lines 8–12 of Fig. 9.11 require $O(e)$ steps. Since no state finds its way onto QUEUE twice for given i, the loop of lines 7–12 requires time $O(em)$. It is thus easy to show that the body of the main loop, lines 2–12, requires $O(em)$ time. Thus the entire algorithm requires $O(emn)$ steps. □

We have the following important corollary, which relates the recognition of regular sets to Algorithm 9.1.

Corollary. If β is any regular expression and $x = a_1a_2 \cdots a_n$ is a string of length n, then there is an NDFA M accepting the language denoted by β, such that Algorithm 9.1 requires $O(n|\beta|)$ steps to determine the sequence of states $S_0, S_1, \ldots, S_n$, where $S_i = \{s | (s_0, a_1a_2 \cdots a_i) \overset{*}{\vdash} (s, \epsilon)\}$ for $0 \leq i \leq n$.

Proof. By Theorem 9.2, we may construct M to have at most $2|\beta|$ states and at most two edges out of any state. Thus e of Theorem 9.5 is at most 2. □

Various pattern recognition algorithms may be constructed from Algorithm 9.1. For example, suppose we are given a regular expression α and a text string $x = a_1a_2 \cdots a_n$, and we wish to find the least k such that there exists a $j < k$ for which $a_ja_{j+1} \cdots a_k$ is in the set denoted by α. Using Theorem 9.2 we can construct from α an NDFA M to accept the language $I^*\alpha$. To find the least k such that $a_1a_2 \cdots a_k$ is in $L(M)$, we can insert a test at the end of the block of lines 2–12 in Fig. 9.11 to see whether S_i contains a state of F. We may, by Theorem 9.2, take F to be a singleton, so this test is not time-consuming; it is $O(m)$, where m is the number of states in M. If S_i contains a state of F, then we break out of the main loop, having found $a_1a_2 \cdots a_i$ to be the shortest prefix of x in $L(M)$.

Algorithm 9.1 can be further modified to produce for each such k the greatest $j < k$ (or the least j) such that $a_ja_{j+1} \cdots a_k$ is in the set denoted by α. This is done by associating an integer with each state in the sets S_i. The integer associated with state s in S_k indicates the greatest j (or the least j) such that $(s_0, a_ja_{j+1} \cdots a_k) \overset{*}{\vdash} (s, \epsilon)$. The details of updating these integers in Algorithm 9.1 are left as an exercise.

9.3 RECOGNITION OF SUBSTRINGS

An important special case of the general problem described in the last section occurs when the regular expression α is of the form $y_1 + y_2 + \cdots + y_k$ where each y_i is a string over some alphabet I. The corollary to Theorem 9.5

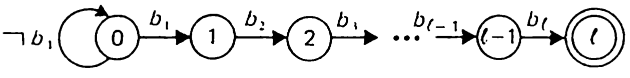


Fig. 9.12 A skeletal machine.

implies that we could find the first occurrence of a pattern y_i in text string $x = a_1a_2 \cdots a_n$ in $O(ln)$ steps, where l is the sum of the lengths of the y_i's. However, an $O(l+n)$ solution is possible. We shall first consider the case where there is only one pattern string $y = b_1b_2 \cdots b_l$, where each b_i is a symbol in I.

We shall construct from the pattern y a deterministic pattern-matching machine M_y that recognizes the shortest instance of a string in I^*y. To construct M_y we first construct a *skeletal* DFA with $l+1$ states labeled $0, 1, \ldots, l$ and a transition from state $i-1$ to state i on input symbol b_i as shown in Fig. 9.12. State 0 also has a transition to itself on all $b \neq b_1$. We can think of state i as a pointer to the ith position in the pattern string y.

The pattern-matching machine M_y operates like a deterministic finite automaton except that it can make several state transitions while scanning the same input symbol. M_y has the same set of states as the skeletal machine. Thus state j of M_y corresponds to the prefix $b_1b_2 \cdots b_j$ of the pattern string y.

M_y starts off in state 0 with its input pointer at a_1, the first symbol of the text string $x = a_1a_2 \cdots a_n$. If $a_1 = b_1$, then M_y enters state 1 and advances its input pointer to position 2 of the text string. If $a_1 \neq b_1$, then M_y remains in state 0 and advances its input pointer to position 2.

Suppose after having read $a_1a_2 \cdots a_k$, M_y is in state j. This implies that the last j symbols of $a_1a_2 \cdots a_k$ are $b_1b_2 \cdots b_j$ and that the last m symbols of $a_1a_2 \cdots a_k$ are not a prefix of $b_1b_2 \cdots b_l$ for $m > j$. If a_{k+1}, the next input symbol, agrees with b_{j+1}, M_y enters state $j+1$ and advances its input pointer to a_{k+2}. If $a_{k+1} \neq b_{j+1}$, M_y enters the highest-numbered state i such that $b_1b_2 \cdots b_i$ is a suffix of $a_1a_2 \cdots a_{k+1}$.

To help determine state i, the machine M_y has associated with it an integer-valued function f, called the *failure function*, such that $f(j)$ is the largest s less than j for which $b_1b_2 \cdots b_s$ is a suffix of $b_1b_2 \cdots b_j$. That is, $f(j)$ is the largest $s < j$ such that $b_1b_2 \cdots b_s = b_{j-s+1}b_{j-s+2} \cdots b_j$. If there is no such $s \geq 1$, then $f(j) = 0$.

Example 9.7. Suppose $y = aabbaab$. The values of f are as follows.

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

For example, $f(6) = 2$ since *aa* is the longest proper prefix of *aabbaa* that is a suffix of *aabbaa*. □

We shall give an algorithm to compute the failure function shortly. First, to see how the failure function is used by M_y, let us define the function $f^{(m)}(j)$ as follows:

i) $f^{(1)}(j) = f(j)$, and
ii) $f^{(m)}(j) = f(f^{m-1}(j))$, for $m > 1$.

That is, $f^{(m)}(j)$ is just f applied m times to j. [In Example 9.7, $f^{(2)}(6) = 1$.]

Suppose once again that M_y is in state j, having read $a_1a_2 \cdots a_k$, and that $a_{k+1} \neq b_{j+1}$. At this point M_y applies the failure function repeatedly to j until it finds the smallest value of m for which either

1. $f^{(m)}(j) = u$ and $a_{k+1} = b_{u+1}$, or
2. $f^{(m)}(j) = 0$ and $a_{k+1} \neq b_1$.

That is, M_y backs up through states $f^{(1)}(j), f^{(2)}(j), \ldots$ until either case 1 or 2 holds for $f^{(m)}(j)$ but not for $f^{(m-1)}(j)$. If case 1 holds, M_y enters state $u + 1$. If case 2 holds, M_y enters state 0. In either case, the input pointer is advanced to position a_{k+2}.

In case 1 it is easy to verify that if $b_1b_2 \cdots b_j$ was the longest prefix of y that is a suffix of $a_1a_2 \cdots a_k$, then $b_1b_2 \cdots b_{f^{(m)}(j)+1}$ is the longest prefix of y that is a suffix of $a_1a_2 \cdots a_{k+1}$. In case 2, no prefix of y is a suffix of $a_1a_2 \cdots a_{k+1}$.

M_y then processes input symbol a_{k+2}. M_y continues operating in this fashion either until it enters the final state l, in which case we know that the last l input symbols scanned constitute an instance of the pattern $y = b_1b_2 \cdots b_l$, or until M_y has processed the last input symbol of x without entering state l, in which case we know that y is not a substring of x.

Example 9.8. Suppose $y = aabbaab$. A pattern-matching machine M_y is shown in Fig. 9.13. The dashed arrow points to the value of the failure function at each state. On input $x = abaabaabbaab$, M_y would undergo the following sequence of state transitions.

Input:		*a*		*b*		*a*		*a*		*b*		*a*		*a*		*b*		*b*		*a*		*a*		*b*	
State:	0		1		0		1		2		3		1		2		3		4		5		6		7
			0								0														

For example, initially M_y is in state 0. On reading the first symbol of x, M_y enters state 1. Since there is no transition from state 1 on the second input

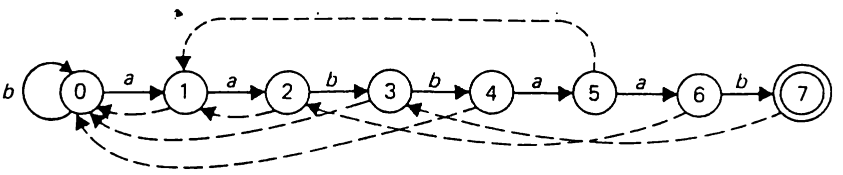


Fig. 9.13 A pattern-matching machine.

symbol b, M_y then enters state 0, the value of the failure function for state 1, without moving the input pointer. Since the first symbol of y is not b, case 2 prevails and M_y remains in state 0 and advances its input pointer to position 3.

After reading the 12th input symbol, M_y enters the final state 7. Thus, at position 12 of x, M_y has found an occurrence of the pattern y. □

The function f can be computed iteratively in much the same manner as that in which M_y operates. By definition, $f(1) = 0$. Suppose that we have computed $f(1), f(2), \ldots, f(j)$. Let $f(j) = i$. To compute $f(j+1)$, we examine b_{j+1} and b_{i+1}. If $b_{j+1} = b_{i+1}$, then $f(j+1) = f(j) + 1$, since

$$b_1 b_2 \cdots b_i b_{i+1} = b_{j-i+1} b_{j-i+2} \cdots b_j b_{j+1}.$$

If $b_{j+1} \neq b_{i+1}$, we then find the smallest m for which either

1. $f^{(m)}(j) = u$ and $b_{j+1} = b_{u+1}$, or
2. $f^{(m)}(j) = 0$ and $b_{j+1} \neq b_1$.

In case 1, we set $f(j+1) = u + 1$. In case 2, we set $f(j+1) = 0$. The following algorithm gives the details.

Algorithm 9.2. Computing the failure function.

Input. Pattern $y = b_1 b_2 \cdots b_l$, $l \geq 1$.

Output. Failure function f for y.

Method. Execute the program in Fig. 9.14. □

Example 9.9. Consider the behavior of Algorithm 9.2 on input $y = aabbaab$. The initialization gives $f(1) = 0$. Since $b_2 = b_1$, $f(2) = 1$. However, $b_3 \neq b_2$ and $b_3 \neq b_1$ so $f(3) = 0$. Continuing in this fashion, we get the values for f given in Example 9.7 □

```
begin
    $f(1) \leftarrow 0$;
    for $j \leftarrow 2$ until $l$ do
        begin
            $i \leftarrow f(j-1)$;
            while $b_j \neq b_{i+1}$ and $i > 0$ do $i \leftarrow f(i)$;
            if $b_j \neq b_{i+1}$ and $i = 0$ then $f(j) \leftarrow 0$
            else $f(j) \leftarrow i + 1$
        end
end
```

Fig. 9.14. Computation of the failure function.

We shall now prove that Algorithm 9.2 correctly computes f in $O(|y|)$ time. We shall prove the correctness of Algorithm 9.2 first.

Theorem 9.6. Algorithm 9.2 computes f.

Proof. We shall prove by induction on j that $f(j)$ is the largest integer i less than j such that $b_1 b_2 \cdots b_i = b_{j-i+1} b_{j-i+2} \cdots b_j$. If no such i exists, $f(j) = 0$.

By definition, $f(1) = 0$. Suppose the inductive hypothesis is true for all $f(k)$, $k < j$. In computing $f(j)$, Algorithm 9.2 compares b_j with $b_{f(j-1)+1}$ at line 4.

CASE 1. Suppose $b_j = b_{f(j-1)+1}$. Since $f(j-1)$ is the largest i such that $b_1 \cdots b_i = b_{j-i} \cdots b_{j-1}$, it follows that $f(j) = i + 1$ is correct. Thus by lines 5 and 6, $f(j)$ is computed correctly.

CASE 2. Suppose $b_j \neq b_{f(j-1)+1}$. Then we must find the largest value of i such that $b_1 \cdots b_i = b_{j-i} \cdots b_{j-1}$ and $b_{i+1} = b_j$ if such an i exists. If no such i exists then clearly $f(j) = 0$, and $f(j)$ is correctly computed by line 5. Let $i_1, i_2, \ldots$ be the largest, second largest, etc., values of i such that

$$b_1 b_2 \cdots b_i = b_{j-i} \cdots b_{j-1}.$$

By a simple induction we see that $i_1 = f(j-1)$, $i_2 = f(i_1) = f^{(2)}(j-1), \cdots,$ $i_k = f(i_{k-1}) = f^{(k)}(j-1)$, since i_{k-1} is the $(k-1)$st largest value of i such that $b_1 \cdots b_i = b_{j-i} \cdots b_{j-1}$ and i_k is the largest value of $i < i_{k-1}$ such that $b_1 \cdots b_i = b_{i_{k-1}-i+1} \cdots b_{i_{k-1}} = b_{j-i} \cdots b_{j-1}$. Line 4 considers $i_1, i_2, \ldots$ in turn until a value of i is found such that $b_1 \cdots b_i = b_{j-i} \cdots b_{j-1}$ and $b_{i+1} = b_j$, if such a value of i exists. On termination of the execution of the **while** statement, $i = i_m$ if such an i_m exists and thus $f(j)$ is correctly computed by line 5.

Thus $f(j)$ is correctly computed for all j. □

Theorem 9.7. Algorithm 9.2 computes f in $O(l)$ steps.

Proof. Lines 3 and 5 are of fixed cost. The cost of the **while** statement is proportional to the number of times i is decremented by the statement $i \leftarrow f(i)$ following the **do** on line 4. The only way i is incremented is by assigning $f(j) = i + 1$ in line 6 then incrementing j by 1 at line 2 and setting i to $f(j-1)$ at line 3. Since $i = 0$ initially, and line 6 is executed at most $l - 1$ times, we conclude that the **while** statement of line 4 cannot be executed more than l times. Thus the total cost of executing line 4 is $O(l)$. The remainder of the algorithm is clearly $O(l)$, and thus the whole algorithm takes $O(l)$ time. □

By an argument identical to that in Theorem 9.6 we can prove that the pattern-matching machine M_y will be in state i after reading $a_1 a_2 \cdots a_k$ if and only if $b_1 b_2 \cdots b_i$ is the longest prefix of y that is a suffix of $a_1 a_2 \cdots a_k$.

Thus M_y correctly finds the leftmost occurrence of y in the text string $x = a_1a_2 \cdots a_n$.

By the argument of Theorem 9.7 we can prove that M_y will undergo at most $2|x|$ state transitions in processing the input string x. Thus we can determine whether y is a substring of x by tracing out the state transitions of M_y on input x.† To do this, all we need is the failure function f for y. By Theorem 9.7 the function f can be constructed in $O(|y|)$ time. Thus we can determine whether y is a substring of x in $O(|x| + |y|)$ time that is independent of the alphabet size. If the alphabet of the pattern string is small and if the text string is considerably longer than the pattern, then we might consider simulating a DFA to accept the language I^*y. The DFA makes exactly one state transition per input symbol.

Algorithm 9.3. Construction of a DFA for I^*y.

Input. A pattern string $y = b_1b_2 \cdots b_l$ over alphabet I. For convenience, we take b_{l+1} to be a new symbol not equal to any symbol in I.

Output. A DFA M such that $L(M) = I^*y$.

Method

1. Use Algorithm 9.2 to construct the failure function f for y.
2. Let $M = (S, I, \delta, 0, \{l\})$, where $S = \{0, 1, \ldots, l\}$ and δ is constructed as follows.

```
begin
    for $j = 1$ until $l$ do $\delta(j-1, b_j) \leftarrow j$;
    for each $b$ in $I$, $b \neq b_1$ do $\delta(0, b) \leftarrow 0$;
    for $j = 1$ until $l$ do
        for each $b$ in $I$, $b \neq b_{j+1}$ do $\delta(j, b) \leftarrow \delta(f(j), b)$
end □
```

Theorem 9.8. Algorithm 9.3 constructs a DFA M such that

$$(0, a_1a_2 \cdots a_k) \overset{*}{\vdash} (j, \epsilon)$$

if and only if $b_1b_2 \cdots b_j$ is a suffix of $a_1a_2 \cdots a_k$, but for no $i > j$ is $b_1b_2 \cdots b_i$ a suffix of $a_1a_2 \cdots a_k$.

Proof. The proof is an induction on k, making use of the arguments found in the proof of Theorem 9.6. We leave the proof to the reader. □

Example 9.10. The DFA M for $y = aabbaab$ resulting from using Algorithm 9.3 is shown in Fig. 9.15.

† Recall that the state of M_y is really a pointer to a position in the pattern y. Thus the state transitions of M_y can be directly implemented by moving a pointer within y.

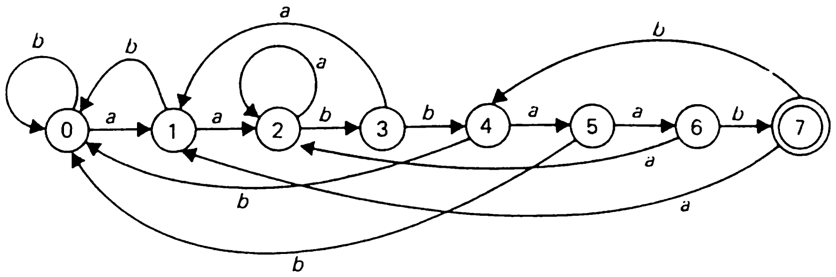


Fig. 9.15 A deterministic finite automaton accepting $(a + b)^* aabbaab$.

On input $x = abaabaabbaab$, M would undergo the following state transitions.

Input: a b a a b a a b b a a b
State: 0 1 0 1 2 3 1 2 3 4 5 6 7

The only difference between M and M_y is that M has precomputed the next state in case of a mismatch. Thus M makes exactly one state transition on each input symbol. □

We summarize the main results of this section in the following theorem.

Theorem 9.9. In $O(|x| + |y|)$ time we can determine whether y is a substring of x.

Let us now mention the case in which we are given several pattern strings $y_1, y_2, \ldots, y_k$. Our problem is to determine whether some y_i is a substring of a given string $x = a_1 \cdots a_n$. The methods of this section can also be applied to this problem. We first construct a skeletal machine for $y_1, y_2, \ldots, y_k$. This machine will be a tree. We can then compute the failure function on the tree in time proportional to $l = |y_1| + |y_2| + \cdots + |y_k|$. The pattern-matching machine can then be constructed in the same manner as before. Thus in $O(l + n)$ steps we can determine whether some y_i is a substring of x. The details are left for an exercise.

9.4 TWO-WAY DETERMINISTIC PUSHDOWN AUTOMATA

Once one suspects that there is an $O(|x| + |y|)$ algorithm to determine whether y is a substring of x, such an algorithm is not hard to find. But what would cause someone to suspect the existence of such an algorithm in the first place? One possible approach arises from the study of two-way deterministic pushdown automata (2DPDA, for short).

A 2DPDA is a special type of Turing machine that accepts a language. We can reformulate many pattern recognition problems in terms of language recognition problems. For example, let L be the language $\{xcy \mid x$ and y are in I^*, $c \notin I$, and y is a substring of $x\}$. Then determining whether y is a substring of x is equivalent to determining whether the string xcy is a sentence of the language L. In this section we shall show that there is a 2DPDA that can recognize L. Although the 2DPDA recognizing this language may take time $O(n^2)$, there is a powerful simulation technique whereby a RAM can in $O(n)$ time simulate the behavior of a given 2DPDA on any input of length n. In this section we shall study this simulation technique in detail.

A 2DPDA can be viewed as a two-tape Turing machine in which one of the tapes is used as a pushdown store as shown in Fig. 9.16. A 2DPDA has a read-only input tape with a left endmarker ¢ in the leftmost square and a right endmarker \$ in the rightmost square. The input string to be recognized is located between the two endmarkers, one symbol per square. Input symbols are drawn from an input alphabet I which is presumed not to contain ¢ or \$. The input tape head can read one symbol at a time, and in one move the input tape head can shift one square to the left, remain stationary, or shift one square to the right. We assume the tape head cannot fall off either end of the input tape; the presence of the endmarkers ¢ and \$ allows us to write a next-move function which never moves the input head left from ¢ nor right from \$.

The pushdown list is a stack holding symbols from a pushdown list alphabet T. The bottom cell of the pushdown list holds the special symbol Z_0, which marks the bottom of the stack. We assume Z_0 is not in T.

The finite control is always in one of a finite set of states S. The operation of the machine is dictated by a next-move function δ, where for s in S, a in

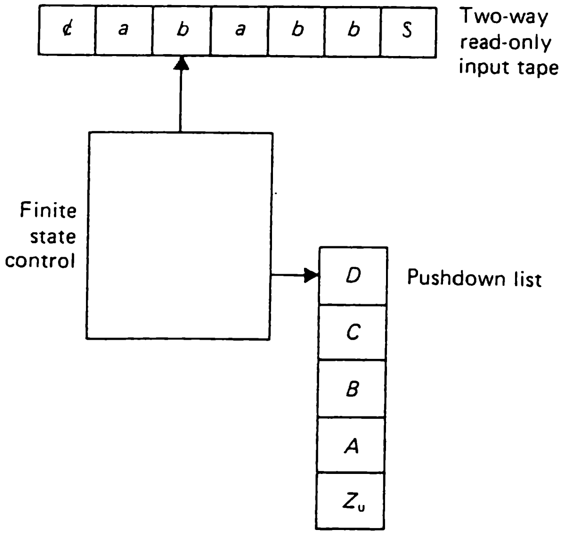


Fig. 9.16 A two-way deterministic pushdown automaton.

$I \cup \{¢, \$\}$, and A in $T \cup \{Z_0\}$, $\delta(s, a, A)$ indicates the action the machine will take when the finite control is in state s, the input tape head is scanning the symbol a, and the pushdown list has the symbol A on top. There are three possible actions.

$$\delta(s, a, A) = \begin{cases} (s', d, \textbf{push } B) & \text{provided} \quad B \neq Z_0; \\ (s', d); & \\ (s', d, \textbf{pop}) & \text{if} \quad A \neq Z_0. \end{cases}$$

In these three actions, the machine enters state s' and moves its input head in the direction d (where $d = -1, +1$, or 0, meaning move one square to the left, move one square to the right, or remain stationary). **push** B means add the symbol B to the top of the pushdown list. **pop** means remove the topmost symbol from the pushdown list.

Definition. We can formally define a 2DPDA as a 7-tuple

$$P = (S, I, T, \delta, s_0, Z_0, s_f),$$

where

1. S is the set of *states of the finite control.*
2. I is the *input alphabet* (excluding ¢ and \$).
3. T is the *pushdown list alphabet* (excluding Z_0).
4. δ is a mapping on $(S - \{s_f\}) \times (I \cup \{¢, \$\}) \times (T \cup \{Z_0\})$. The value of $\delta(s, a, A)$ is, if defined, of one of the forms $(s', d;$ **push** $B)$, (s', d), or $(s', d,$ **pop**$)$ where $s' \in S$, $B \in T$, and $d \in \{-1, 0, +1\}$. We assume a 2DPDA makes no moves from the final state s_f, and certain other states may have no moves defined. Also, $\delta(s, ¢, A)$ does not have second component $d = -1$ and $\delta(s, \$, A)$ does not have second component $d = +1$ for any s and A. Finally $\delta(s, a, Z_0)$ does not have a third component **pop.**
5. s_0 in S is the *initial state* of the finite control.
6. Z_0 is the *bottom marker* for the pushdown list.
7. s_f is the one designated *final state.*

An *instantaneous description* (ID) of a 2DPDA P on input $w = a_1 a_2 \cdots a_n$ is a triple (s, i, α), where

1. s is a state in S.
2. i is an integer, $0 \leq i \leq n + 1$, indicating the position of the input head. We assume $a_0 = ¢$ and $a_{n+1} = \$$.
3. α is a string representing the contents of the pushdown list with the leftmost symbol of α on top.

A *move* by a 2DPDA with input $a_1 a_2 \cdots a_n$ can be defined by a binary relation $\vdash$ on ID's. We write the following.

1. $(s, i, A\alpha) \vdash (s', i+d, BA\alpha)$ if $\delta(s, a_i, A) = (s', d, \textbf{push } B)$.
2. $(s, i, A\alpha) \vdash (s', i+d, A\alpha)$ if $\delta(s, a_i, A) = (s', d)$.
3. $(s, i, A\alpha) \vdash (s', i+d, \alpha)$ if $\delta(s, a_i, A) = (s', d, \textbf{pop})$.

Note that symbols can be added to, read from, or removed from only the top of the pushdown list. We also have the restriction that a 2DPDA will have only one bottom marker on its pushdown list at any time. We use $\overset{*}{\vdash}$ to denote sequences of zero or more moves, where $\overset{*}{\vdash}$ is the reflexive transitive closure of $\vdash$.

The *initial ID* of a 2DPDA P with input $w = a_1a_2 \cdots a_n$ (excluding endmarkers) is $(s_0, 1, Z_0)$, indicating P is in its initial state s_0, the input head is scanning the leftmost symbol of $a_1a_2 \cdots a_na_{n+1}$,† and the pushdown list contains only the bottom marker Z_0.

An *accepting ID* for $w = a_1 \cdots a_n$ is one of the form (s_f, i, Z_0) for some $0 \leq i \leq n + 1$, indicating P has entered the final state s_f and the pushdown list contains only the bottom marker.

A 2DPDA P is said to *accept* an input string w if under input w,

$$(s_0, 1, Z_0) \overset{*}{\vdash} (s_f, i, Z_0)$$

for some $0 \leq i \leq |w| + 1$. The *language accepted* by P, denoted $L(P)$, is the set of strings accepted by P.

Let us consider some examples of 2DPDA's and the languages they accept. We shall describe the 2DPDA's informally in terms of their behavior rather than as formal 7-tuples.

Example 9.11. Consider the language $L = \{xcy | x$ and y are in $\{a, b\}^*$ and y is a substring of $x\}$. A 2DPDA P can recognize L as follows. Suppose P is given an input string w of the form xcy, where $x = a_1a_2 \cdots a_n$ and $a_i \in \{a, b\}$, $1 \leq i \leq n$.

1. P moves its input head to the right until it encounters c.
2. P then moves its input head to the left, copying $a_na_{n-1} \cdots a_1$ onto the pushdown list until the input head reaches the left endmarker. At this point P has $xZ_0 = a_1a_2 \cdots a_nZ_0$ on its pushdown list (with the leftmost symbol of xZ_0 on top).
3. P then moves its input head to the first symbol to the right of c (that is, to the beginning of y), without disturbing the pushdown list, and prepares to match y against the pushdown list.
4. **while** the top symbol of the pushdown list matches the symbol scanned by the input head **do**

```
    begin
        pop the top symbol off the pushdown list;
        move the input head one position to the right
    end
```

† If $n = 0$, that is, $w = \epsilon$, then P is scanning $a_{n+1} = \$$, the right endmarker.

5. At this point there are two possibilities:
   a) The input head has reached $\$$, the right endmarker, in which case P accepts the input.
   b) The symbol on top of the pushdown list disagrees with the current input symbol. Since the string y and pushdown list have matched so far, P may restore its pushdown list by moving its input head to the left, copying the input onto the pushdown list until the input symbol c is reached. At this point P has $a_i a_{i+1} \cdots a_n Z_0$ on its pushdown list for some $1 \leq i \leq n$.[†] If $i = n$, P halts in a nonfinal state. Otherwise P pops a_i off the pushdown list and moves its input head to the right, to the first symbol of $y\$$. Then P goes back to step 4, attempting to find y as a prefix of the string $a_{i+1} \cdots a_n$.

We see that P discovers whether y is a substring of $x = a_1 a_2 \cdots a_n$ in the natural way. P attempts to match y with a prefix of $a_i a_{i+1} \cdots a_n$ for $i = 1, 2, \ldots, n$, in turn. If a match is found, P accepts. Otherwise P rejects. Note that this procedure may require $O(|x| \cdot |y|)$ moves by P. □

Example 9.12. Consider the language $L = \{w | w \in \{a, b, c\}^*, w = xy$ such that $|x| \geq 2$ and $x^R = x\}$ (i.e., some prefix of w of length 2 or greater is a palindrome). Here superscript R stands for string reversal; for example $(abb)^R = bba$. The requirement that $|x| \geq 2$ is imposed since every string in $\{a, b, c\}^+$ begins with one of the trivial palindromes a, b, or c. Consider a 2DPDA P that behaves as follows, given an input string $a_1 a_2 \cdots a_n$.

1. P moves its input head to the right endmarker, copying the input onto the pushdown list. The pushdown list now contains $a_n \cdots a_2 a_1 Z_0$. If $n < 2$, P rejects the input immediately.
2. P then moves its input head to the left endmarker without disturbing the pushdown list. P positions its input head over the symbol immediately to the right of the left endmarker.
3.
```
while the top symbol of the pushdown list matches the symbol scanned
  by the input head do
   begin
       pop the top symbol off the pushdown list;
       move the input head one position to the right
   end
```
4. At this point there are two possibilities:
   a) The pushdown list contains only Z_0, in which case P halts and accepts the input.
   b) The symbol on top of the pushdown list disagrees with the current input symbol. In this case P moves its input head to the left, copying the input back onto the pushdown list until the left endmarker is reached.

[†] The first time we reach here, $i = 1$, but subsequently i will increase by 1 each time through.

If the original input string was $a_1a_2 \cdots a_n$, then at this point P has $a_i \cdots a_2a_1Z_0$ on its pushdown list for some i. If $i = 2$, P halts and rejects. Otherwise, P pops a_i off the pushdown list and moves its input head to the right one square, to the symbol immediately to the right of the left endmarker. P then goes back to step 3.

Thus P determines whether an input string $a_1 \cdots a_n$ begins with a palindrome by attempting to match $a_i \cdots a_1$ with a prefix of $a_1 \cdots a_n$ for each i, $2 \leq i \leq n$. This procedure may require P to make $O(n^2)$ moves. □

We shall now show that we can determine in $O(|w|)$ time whether a 2DPDA P accepts an input string w regardless of how many moves P actually makes. Throughout the following discussion we assume that we are talking about a fixed 2DPDA $P = (S, I, T, \delta, s_0, Z_0, s_f)$ and a fixed input string w of length n.

Definition. A *surface configuration* (*configuration*, for short) of P is a triple (s, i, A) such that $s \in S$, i is an input head position, $0 \leq i \leq n + 1$, and A is a single pushdown list symbol in $T \cup \{Z_0\}$.

Note that a configuration is an ID, although not every ID is a configuration. In a sense, each surface configuration represents many ID's, those whose "surface"—the top of the pushdown list—matches that of the configuration. We say configuration $C = (s,i,A)$ *derives* configuration $C' = (s',i',A)$, written $C \Longrightarrow C'$, if there exists a sequence of moves by P such that, for $m \geq 0$,

$$\begin{aligned} C &\vdash (s_1, i_1, \alpha_1) \\ &\vdash (s_2, i_2, \alpha_2) \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &\vdash (s_m, i_m, \alpha_m) \\ &\vdash C', \end{aligned}$$

where $|\alpha_i| \geq 2$ for each i.† In this sequence of moves, the intermediate ID's (s_j, i_j, α_j) have at least two symbols on the pushdown list. We shall deal with configurations rather than ID's because there are only $O(n)$ configurations, while there can be an exponential number of distinct ID's used.

Definition. A configuration (s, i, A) is said to be *terminal* if $\delta(s, a_i, A)$ is undefined or of the form $(s', d, \mathbf{pop})$. That is, a terminal configuration causes P to halt or to pop a symbol off the pushdown list. The *terminator* of a configuration C is the unique terminal configuration C' (if it exists) such that $C \overset{*}{\Longrightarrow} C'$, where $\overset{*}{\Longrightarrow}$ is the reflexive and transitive closure of $\Longrightarrow$.

† If $m = 0$, then $C \vdash C'$.

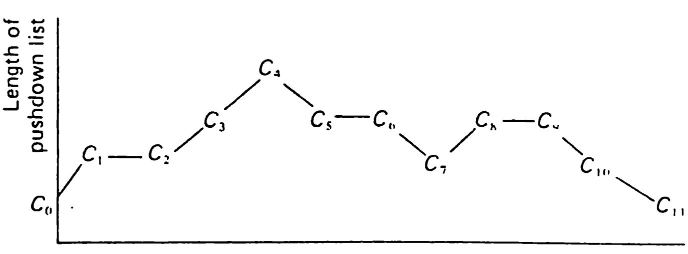


Fig. 9.17 Sequence of configurations.

Example 9.13. If we plot the length of the pushdown list as a function of the number of moves made by P, we can obtain a curve that has the shape shown in Fig. 9.17. On this graph we have labeled each point by the configuration (not ID) of P after each move.

From Fig. 9.17 we see that C_0 and C_{11}, for example, have Z_0 on top of the stack. Since there is no intervening configuration with Z_0 on top of the stack, we conclude that $C_0 \Rightarrow C_{11}$. If C_{11} is a final configuration, then C_{11} is the terminator both for itself and for C_0 as well. From this graph, we may also infer that $C_1 \Rightarrow C_2$, $C_2 \Rightarrow C_7$, $C_1 \overset{*}{\Rightarrow} C_{10}$, and that C_{10} is a terminal configuration, since C_{10} causes P to pop a symbol from the pushdown list. □

The following two simple lemmas are the key to the simulation algorithm.

Lemma 9.1. P accepts w if and only if some configuration of the form (s_f, i, Z_0) for some $0 \leq i \leq |w| + 1$ is the terminator of the initial configuration $(s_0, 1, Z_0)$.

Proof. The result follows directly from the definition of how a 2DPDA accepts an input string. □

Definition. We say that a configuration C is a *predecessor* of configuration D if $C \overset{*}{\Rightarrow} D$. We say that C is an *immediate predecessor* of D if $C \Rightarrow D$. We say that a pair (C, D) of configurations is *below* a pair (C', D') of (not necessarily distinct) configurations if

(1) $C = (s, i, A) \qquad D = (t, j, A)$.

(2) $C' = (s', i', B) \qquad D' = (t', j', B)$.

(3) $(s, i, A) \vdash (s', i', BA)$,

and

(4) $(t', j', BA) \vdash (t, j, A)$,

that is, if P can go from C to C' by a push move and from D' to D by a pop move.

Lemma 9.2. If (C, D) is below (C', D') and $C' \overset{*}{\Rightarrow} D'$, then $C \Rightarrow D$.

Proof. Easy exercise. □

Example 9.14. In Fig. 9.17, (C_3, C_5) is below (C_4, C_4), and (C_7, C_{10}) is below (C_8, C_9). However, we cannot be sure whether (C_2, C_{10}) is below (C_3, C_9), since C_3 and C_9 may not have the same pushdown symbol on top of the pushdown list. □

The simulation algorithm works by finding the terminator for each configuration of the 2DPDA P with input w. Once we have found the terminator for the initial configuration $(q_0, 1, Z_0)$ we are done.

We shall use an array called TERM to store the terminator of each configuration. We assume that the configurations have been linearly ordered (by means of some lexicographic conventions). Then we may treat configuration name C as if it were an integer and let TERM$[C]$ be the terminator of C.

We shall also use PRED, an array of lists. PRED is indexed by configurations, and PRED$[D]$ will be a list of configurations C such that $C \Longrightarrow D$.

In addition to the arrays TERM and PRED, we shall also use two additional lists, NEW and TEMP. The list NEW will contain pairs of not yet considered configurations (C, D) such that TERM$[C] = D$. The list TEMP is used in a procedure UPDATE (C, D) to hold predecessors of configuration C.

We proceed as follows. We first set TERM$[C] = C$ if C is a terminal configuration. (Each terminal configuration is its own terminator.) Add (C, C) to NEW. We then consider pairs (C, D) of configurations such that $C \vdash D$ in one move of P. (Note that in such a move the pushdown list is not disturbed.)

If the terminator of D is already known, we set TERM$[E] =$ TERM$[D]$ for all E that are known to be predecessors of C at this time, including C itself. (The proper predecessors are found on PRED$[C]$.) We also add the pair $(E,$ TERM$[D])$ to the list NEW.

If the terminator of D is not yet known, then C is placed on PRED$[D]$, the list of immediate predecessors of D.

At this point, for each configuration C we shall have determined the unique configuration D such that $C \overset{\cdot}{\Longrightarrow} D$ without manipulating the stack, and either D is the terminator of C or $D \vdash (s, i, \alpha)$ for $|\alpha| = 2$. We now consider all pairs of configurations that have been added to the list NEW. In general NEW contains unconsidered pairs of configurations (A, B) such that $A \overset{\cdot}{\Longrightarrow} B$ and B is terminal. Suppose NEW contains the pair (A, B). We remove (A, B) from NEW and consider each pair (C, D) of configurations below (A, B). If the terminator of D has already been computed, then we set TERM$[C] =$ TERM$[D]$ and add the pair $(C,$ TERM$[D])$ to the list NEW. For each E that is on PRED$[C]$, we set TERM$[E] =$ TERM$[D]$ and add $(E,$ TERM$[D])$ to the list NEW. However, if the terminator of D has not yet been computed, then we place C on the list PRED$[D]$. We continue this procedure until NEW becomes empty. At this point we shall have determined the terminator (if it exists) for each configuration C.

Once we have exhausted NEW we look at TERM$[C_0]$, where C_0 is the initial configuration. If TERM$[C_0]$ is an accepting configuration, we know the 2DPDA P accepts w. Otherwise P rejects w.

The details are given more precisely in the following algorithm.

Algorithm 9.4. Simulation of a 2DPDA.

Input. A 2DPDA $P = (S, I, T, \delta, s_0, Z_0, s_f)$ and an input string $w \in I^*$, $|w| = n$.

Output. The answer "yes" if $w \in L(P)$, "no" otherwise.

Method

1. Initialize the arrays and lists as follows. For each configuration C, set TERM$[C] = \emptyset$ and PRED$[C] = \emptyset$. Set NEW $= \emptyset$ and TEMP $= \emptyset$.
2. For each terminal configuration C, set TERM$[C] = C$ and add the pair (C, C) to NEW.
3. For each configuration C, determine whether $C \vdash D$ for some configuration D in one move. If so, call UPDATE(C, D). The procedure UPDATE is given in Fig. 9.18.
4. While NEW is not empty, remove a pair (C', D') of configurations from NEW. For each pair (C,D) such that (C,D) is below (C',D'), call UPDATE(C,D).
5. If TERM$[C_0]$, where C_0 is the initial configuration, is a final configuration, then answer "yes." Otherwise, answer "no." □

Example 9.15. Consider some of the computations that occur when Algorithm 9.4 is applied to the 2DPDA suggested by Fig. 9.17 (p. 341).

```
procedure UPDATE($C$, $D$):
begin
    comment Whenever UPDATE($C$, $D$) is called, we have $C \Rightarrow D$;
    if TERM[$D$] = $\emptyset$ then add $C$ to PRED[$D$]
    else
        begin
            TEMP ← {$C$};
            while TEMP $\neq \emptyset$ do
                begin
                    select and remove a configuration $B$ from TEMP;
                    TERM[$B$] ← TERM[$D$];
                    add ($B$, TERM[$D$]) to NEW;
                    for each $A$ on PRED[$B$] do add $A$ to TEMP
                end
        end
end
```

Fig. 9.18. The procedure UPDATE.

In step 2 we determine that C_4, C_6, C_9, C_{10}, and C_{11} are terminal configurations, and hence their own terminators. We add the pairs (C_4, C_4), (C_6, C_6), (C_9, C_9), (C_{10}, C_{10}), and (C_{11}, C_{11}) to NEW.

In step 3 we call UPDATE(C_1, C_2). Since TERM$[C_2] = \emptyset$ at this time, UPDATE merely places C_1 on the list PRED$[C_2]$. In step 3 we also call UPDATE(C_5, C_6). Since TERM$[C_6] = C_6$, UPDATE sets TERM$[C_5] = C_6$ and adds (C_5, C_6) to NEW. Also in step 3 we call UPDATE(C_8, C_9), which sets TERM$[C_8] = C_9$ and adds (C_8, C_9) to NEW. Thus after step 3 NEW contains the following seven pairs.

$$(C_4, C_4) \quad (C_6, C_6) \quad (C_9, C_9) \quad (C_{10}, C_{10}) \quad (C_{11}, C_{11}) \quad (C_5, C_6) \quad (C_8, C_9)$$

In step 4 we remove (C_4, C_4) from NEW and call UPDATE(C_3, C_5), since (C_3, C_5) is below (C_4, C_4). Since TERM$[C_5] = C_6$ at this time, UPDATE sets TERM$[C_3] = C_6$ and adds (C_3, C_6) to NEW. Then, in step 4, we remove (C_6, C_6) from NEW, and since (let us suppose) there is no pair below (C_6, C_6) we do not call UPDATE.† Similarly, for pairs (C_9, C_9), (C_{10}, C_{10}), (C_{11}, C_{11}), and (C_5, C_6), we do not call UPDATE.

When (C_8, C_9) is removed from NEW, we call UPDATE(C_7, C_{10}), making TERM$[C_7] = C_{10}$ and adding (C_7, C_{10}) to NEW. At this point NEW contains (C_3, C_6) and (C_7, C_{10}).

Removing (C_3, C_6) from NEW, we call UPDATE(C_2, C_7), which makes TERM$[C_2] = C_{10}$ and TERM$[C_1] = C_{10}$, since PRED$[C_2]$ contains C_1. We add (C_2, C_{10}), and (C_1, C_{10}) to NEW.

We invite you to complete this simulation. □

Theorem 9.10. Algorithm 9.4 correctly answers the question "Is $w \in L(P)$?" in $O(|w|)$ time.

Proof. It can be shown that TEMP will never have on it the same configuration more than once. Thus every call to UPDATE will always terminate. It can also be shown that no pair of configurations is placed on the list NEW more than once, so the algorithm itself will always terminate. The details of these two parts of the proof are left as exercises.

We shall now show that at the conclusion of Algorithm 9.4, TERM$[C_0]$ is a final configuration if and only if $w \in L(P)$. It is easy to show simultaneously, by induction on the number of steps of Algorithm 9.4 executed, that

i) If TERM$[C]$ is set to D, then D is the terminator of C.
ii) If C is added to PRED$[D]$, then $C \Longrightarrow D$.
iii) If (C, D) is placed on NEW, then TERM$[C] = D$.

Thus if Algorithm 9.4 finds that TERM$[C_0]$ is a final configuration, then $w \in L(P)$ is true by Lemma 9.1.

† For simplicity, we assume that P has no moves not implied by Fig. 9.17.

For the converse, we must show that if D is the terminator of C, then TERM[C] is eventually set to D. The proof is by induction on the number of moves in the sequence $C \overset{*}{\vdash} D$. The basis, zero moves, is trivial since $C = D$ and TERM[C] is set to D in step 2 of Algorithm 9.4.

For the inductive step, suppose $C \vdash E \overset{*}{\vdash} D$. There are two cases to consider.

CASE 1. E is a configuration, i.e., the move $C \vdash E$ is not a **push** or **pop** move. Thus in step 3, UPDATE(C, E) is called. If TERM[E] has been set to D at this time, C will be placed on TEMP at line 2 and eventually TERM[C] will be set to D at line 5. If TERM[E] is not yet set to D, then we add C to PRED[E] at line 1 of UPDATE(C, E). By the inductive hypothesis, we eventually set TERM[E] $= D$. If this occurs at line 5 of UPDATE, then C is added to TEMP at line 7, and TERM[C] is set to D on that call of UPDATE. TERM[E] cannot be set to D in step 2 of Algorithm 9.4 since $E \neq D$. (If $E = D$, then TERM[E] would have been set to D when we considered $C \vdash E$ in step 3.) We conclude that TERM[C] is set to D in case 1.

CASE 2. E is an ID such that $C \vdash E$ is a **push** move. Then we can find configurations A, B, and F, with (C, F) below (A, B), such that $A \overset{*}{\vdash} B$ and $F \overset{*}{\vdash} D$, each by sequences of fewer moves than the sequence $C \overset{*}{\vdash} D$. (Configuration A is the "surface" of ID E.) By the inductive hypothesis, TERM[A] is set to B and TERM[F] is set to D.

Suppose the latter occurs before the former. Then (A, B) is placed on NEW eventually, and in step 4, UPDATE(C, F) is called. Since TERM[F] $=$ D at this time, we set TERM[C] to D at line 5.

In the contrary situation, suppose TERM[A] is set to B before TERM[F] is set to D. Then when UPDATE(C, F) is called, TERM[F] $= \emptyset$, in which case C is added to PRED[F]. But then TERM[C] is set to D when TERM[F] is computed. This completes the induction and the proof of correctness for Algorithm 9.4.

Consider the running time of Algorithm 9.4. Steps 1 and 2 require $O(n)$ time, since there are $O(n)$ configurations. Since for each configuration the 2DPDA has at most one possible move, there are at most $O(n)$ pairs of configurations (C, D) such that $C \vdash D$. Thus there are at most $O(n)$ calls of UPDATE in step 3.

A pair (C', D') is placed on the list NEW when $C' \overset{*}{\dashrightarrow} D'$ and the terminator of C' has been found to be D'. Since each configuration has a unique terminator (if it has one at all), no pair is placed on the list NEW more than once. Consequently, the total number of pairs placed on the list NEW does not exceed $O(n)$. For each pair placed on NEW, there are only a bounded number of pairs below it, since if (C, D) is below (C', D'), then C and C' differ in head position by at most one, and similarly for D and D'. Thus UPDATE is called $O(n)$ times.

Consider now the total amount of time spent in the subroutine UPDATE. It can be shown that each configuration appears at most once in the PRED array and no configuration is placed on the list TEMP more than once. Thus lines 4–6 of UPDATE can be "charged" to the configuration B removed from TEMP, and line 7 to the configuration A added to TEMP. Since UPDATE is called at most $O(n)$ times, it follows that the total amount of time used by UPDATE, exclusive of that charged to configurations A and B as above, is $O(n)$. Consequently, Algorithm 9.4 runs in linear time. □

The primary application of the results in this section is to show the existence of linear time algorithms for certain problems. We have seen that some pattern-matching problems can be formulated as language recognition problems. If we can design a 2DPDA to accept the language corresponding to a pattern-matching problem, then we know a linear time algorithm exists for the problem. The fact that the 2DPDA can make n^2 or even 2^n moves on an input of length n often makes it easier to find an algorithm to recognize the language on a 2DPDA than a linear algorithm to solve the pattern-matching problem directly on a RAM.

9.5 POSITION TREES AND SUBSTRING IDENTIFIERS

In the previous section we showed that if a pattern-matching problem can be formulated as a language recognition problem for which we can find a 2DPDA, then the original pattern-matching problem can be solved in linear time. We could use Algorithm 9.4 directly as a linear time algorithm for the original problem. However, the constant factor arising from the simulation would make this approach unappetizing at best. In this section we shall study a data structure that can be used in more practical linear time pattern-matching algorithms. Some pattern-matching problems to which this data structure can be applied are the following.

1. Given a text string x and a pattern string y, determine all occurrences of y in x.
2. Given a text string x, determine a longest repeated substring of x.
3. Given two strings x and y, determine a longest string that is a substring of both x and y.

Definition. A *position* in a string of length n is an integer between 1 and n. We say symbol *a occurs in position i* of string x if $x = yaz$, with $|y| = i - 1$. We say substring u *identifies* position i in string x if $x = yuz$, $|y| = i - 1$, and x cannot be written as $y'uz'$ unless $y' = y$. That is, the only occurrence of u within x begins at position i. For example, the substring *bba* identifies position 2 of the string *abbabb*. The substring *bb* does not identify position 2.

Position	Substring identifier
1	*abba*
2	*bba*
3	*ba*
4	*abb*\$
5	*bb*\$
6	*b*\$
7	\$

Fig. 9.19. Substring identifiers for *abbabb*\$.

For the remainder of the chapter let $x = a_1a_2 \cdots a_n$ be a string over some alphabet I, and let $a_{n+1} = \$$ be a symbol not in I. Then each position i of $x\$ = a_1a_2 \cdots a_{n+1}$ is identified by at least one string, namely $a_ia_{i+1} \cdots a_{n+1}$. We call the shortest string which identifies position i in $x\$$ the *substring identifier for position i in x*, denoted $s(i)$.

Example 9.16. Consider the string *abbabb*\$. The substring identifiers for positions 1 through 7 are tabulated in Fig. 9.19. □

The substring identifiers for the positions in a string $x\$$ can be conveniently represented in terms of a tree called an I-tree, with edges and certain vertices labeled.

Definition. An *I-tree* is a labeled tree T such that for each interior vertex v in T, the edges leaving v have distinct labels in alphabet I. If the edge (v, w) in T is labeled a, we shall call w the *a-son* of v.

A *position tree* for a string $x\$ = a_1 \cdots a_na_{n+1}$, where a_i is in I, $1 \leq i \leq n$, and $a_{n+1} = \$$, is an $(I \cup \{\$\})$-tree such that:

1. T has $n + 1$ leaves labeled $1, 2, \ldots, n + 1$. The leaves of T are in one-to-one correspondence with the positions in $x\$$.
2. The sequence of labels of edges on the path from the root to the leaf labeled i is $s(i)$, the substring identifier for position i.

Example 9.17. The position tree for the string *abbabb*\$ is shown in Fig. 9.20. For example, the path from the root to the leaf labeled 2 spells out *bba*, which is the substring identifier for position 2. □

A few basic properties of substring identifiers are stated in the next lemma.

Lemma 9.3. Let $s(i)$ be the substring identifier for position i of string $x\$ = a_1a_2 \cdots a_{n+1}$.

a) If $s(i)$ has length j, then $s(i - 1)$ has length at most $j + 1$.
b) No substring identifier is a proper prefix of another.

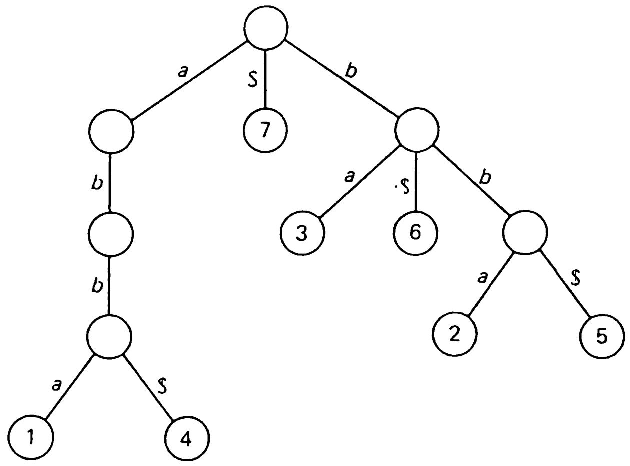


Fig. 9.20 A position tree.

Proof

a) If $s(i-1)$ has length greater than $j+1$, then there is some position $k \neq i-1$ such that $a_{i-1}a_i \cdots a_{i+j-1} = a_k a_{k+1} \cdots a_{k+j}$. Hence $a_i a_{i+1} \cdots a_{i+j-1} = a_{k+1}a_{k+2} \cdots a_{k+j}$, and $a_i a_{i+1} \cdots a_{i+j-1}$ does not identify position i, a contradiction.

b) Easy exercise. □

As a consequence of Lemma 9.3(b), we can be assured that a position tree does in fact exist for each string $x\$$.

A number of pattern-matching problems can be solved with the position tree, including those mentioned at the beginning of this section.

Example 9.18. Consider the basic pattern-matching problem: "Is $y = b_1b_2 \cdots b_p$ a substring of $x = a_1a_2 \cdots a_n$?" Suppose we have constructed the position tree T for $x\$$. To determine whether $y = b_1b_2 \cdots b_p$ is a substring of x, we treat the position tree as the transition diagram of a deterministic finite automaton. That is, we start at the root of T and trace out the longest possible path in the position tree that spells out $b_1b_2 \cdots b_j$ for some $0 \leq j \leq p$. Suppose this path terminates at vertex v. Several cases can occur.

1. If $j < p$ and vertex v is not a leaf, then answer "no." In this case, we know $b_1b_2 \cdots b_j$ is a substring of x but $b_1b_2 \cdots b_jb_{j+1}$ is not.
2. If $j \leq p$ and vertex v is a leaf labeled i, then we know that $b_1b_2 \cdots b_j$ matches the j symbols of x beginning at position i. We must then compare $b_{j+1}b_{j+2} \cdots b_p$ with $a_{i+j}a_{i+j+1} \cdots a_{i+p-1}$. If these two do not match, then answer "no"; otherwise answer "yes" and report that y is a substring of x beginning at position i.

3. If $j = p$ and vertex v is not a leaf, then answer "yes." In this case, y is a substring that begins in two or more positions in x. These positions are given by the labels on the leaves in the subtree of vertex v of the position tree. □

Example 9.19. The position tree can be used to find a longest repeated substring in a given string $x = a_1 a_2 \cdots a_n$ (the two occurrences of the substring may overlap). A longest repeated substring corresponds to an interior vertex of the position tree with the greatest depth. Such a vertex can be located in a straightforward manner.

Consider finding the longest repeated substring in *abbabb*. In the position tree for *abbabb*\$ shown in Fig. 9.20 there is one interior vertex of depth 3 and no interior vertices of greater depth. Thus the string *abb* corresponding to this vertex is the longest repeated substring in *abbabb*. The leaves labeled 1 and 4 tell where the two occurrences of *abb* begin. In this case they happen not to overlap. □

We shall now consider in detail the problem of constructing the position tree. Throughout the remainder of this chapter we use $a_1 \cdots a_n a_{n+1}$ to represent $x\$$, where a_{n+1} is the unique right endmarker \$. We use x_i, $1 \le i \le n + 1$, to represent the suffix $a_i \cdots a_n a_{n+1}$, and $s_i(j)$ to denote the substring identifier for position j in x_i. All positions are with respect to the original string $a_1 \cdots a_n a_{n+1}$.

We shall give an algorithm to construct the position tree for $a_1 \cdots a_n a_{n+1}$ in time proportional to the number of vertices in the resulting tree. We note that a position tree for $a_1 \cdots a_n a_{n+1}$ can have $O(n^2)$ vertices. For example, the position tree for $a^n b^n a^n b^n \$$[†] has $n^2 + 6n + 2$ vertices, as you may check. However, under reasonable assumptions about what is a "random" string (e.g., symbols chosen uniformly and independently from a fixed alphabet), we can show an "average" position tree for a string of length n has $O(n)$ vertices. Although we shall not show it here, there is an $O(n)$ worst-case algorithm to construct a compact form of the position tree directly from a given string. The bibliographic notes contain references to this algorithm.

Let us consider the differences between S_i, the set of substring identifiers for x_i, and S_{i+1}, the set of substring identifiers for x_{i+1}, since we shall construct S_i from S_{i+1} in the algorithm to follow. One obvious difference is that S_i contains $s_i(i)$, the substring identifier for the first position of x_i. Because S_i contains this additional string, it is possible that the substring identifier for some position k in S_{i+1} is no longer the substring identifier for position k in S_i. This situation occurs if and only if $s_{i+1}(k)$, the substring identifier for position k in S_{i+1}, is a prefix of $s_i(i)$. In this situation we must lengthen $s_{i+1}(k)$ to make it $s_i(k)$, the substring identifier for position k in S_i. It is not possible that two

[†] a^n stands for n a's concatenated.

Position p	$s_4(p)$	$s_3(p)$	$s_2(p)$	$s_1(p)$
7	\$	\$	\$	\$
6	$b\$$	$b\$$	$b\$$	$b\$$
5	bb	bb	$bb\$$	$bb\$$
4	a	a	a	$abb\$$
3	—	ba	ba	ba
2	—	—	bba	bba
1	—	—	—	$abba$

Fig. 9.21. Substring identifiers for certain suffixes of $abbabb\$$.

substring identifiers in S_{i+1} need lengthening. For suppose both $s_{i+1}(k_1)$ and $s_{i+1}(k_2)$ did. Then both these strings would be a prefix of $s_i(i)$, and hence one would be a prefix of the other, violating Lemma 9.3(b).

Example 9.20. Let $a_1 \cdots a_n a_{n+1} = abbabb\$$. The substring identifiers for $x_4 = abb\$$, $x_3 = babb\$$, $x_2 = bbabb\$$, and $x_1 = abbabb\$$ are tabulated in Fig. 9.21. Note that S_3 is S_4 with only $s_3(3) = ba$ added. On the other hand, S_2 is S_3 with two changes. We have added $s_2(2) = bba$ to S_2 and we have appended \$ to the end of $s_3(5)$ to get $s_2(5) = bb\$$. To obtain S_1 we have added $s_1(1) = abba$ to S_2 and we have appended $bb\$$ to $s_2(4)$ to make $s_1(4) = abb\$$. □

Consider what is involved in constructing T_i, the position tree representing S_i, from T_{i+1}, the position tree representing S_{i+1}. Given T_{i+1}, we must add to T_{i+1} a leaf labeled i that corresponds to $s_i(i)$. If, for some $i < k \leq n$, $s_{i+1}(k)$ is a prefix of $s_i(i)$, then we must also lengthen the path in T_{i+1} from the root to the leaf labeled k, so the new leaf labeled k in T_i will correspond to $s_i(k)$. Thus the key to efficiently constructing T_i, the position tree for x_i, from T_{i+1} is being able to find quickly the string y such that $a_i y$ is the longest prefix of x_i that is also a substring of x_{i+1} and determining whether $a_i y$ is a prefix of a substring identifier in T_{i+1}.

The construction requires attaching three new structures to a position tree. The first of these new structures is a bit vector (array) that we shall attach to each vertex of a position tree T_i. We shall use B_v to denote this bit vector at vertex v. There will be a component of this vector for each symbol in I. If v is the vertex in T_i that corresponds to string y and $a \in I$, then $B_v[a]$ is 1 if ay is a substring of x_i. Otherwise, $B_v[a]$ is 0.

Next, we attach to each vertex its depth in the position tree. This information can easily be updated as the tree grows, and we shall henceforth assume that it is computed without mentioning that fact specifically.

The final addition to the position tree is a new tree structure placed on the vertices of the position tree. We shall call this tree structure an "auxiliary tree" and in practice it will just be another set of edges defined on the vertices of the position tree.

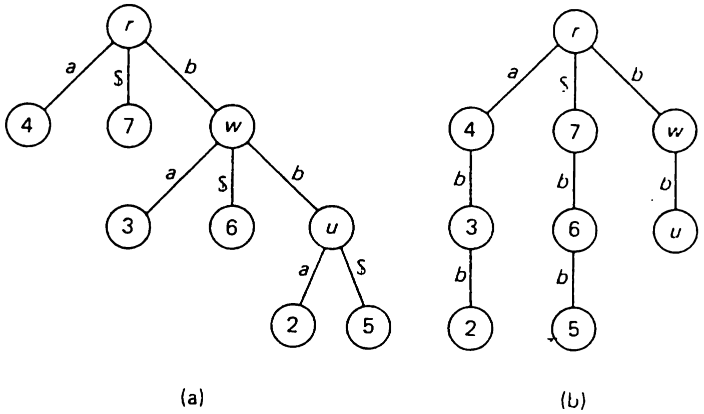


Fig. 9.22 Position tree (a) and its auxiliary tree (b) for *bbabb*$.

Definition. Let T_i be a position tree for $x_i = a_i a_{i+1} \cdots a_n\$$. The *auxiliary tree* for position tree T_i is an $(I \cup \{\$\})$-tree A_i such that:

1. A_i has the same set of vertices as T_i.
2. Vertex w is the a-son of v in A_i if ay is the string corresponding to w in T_i and y is the string corresponding to v in T_i. Thus in A_i the path from the root to w spells out $y^R a$, whereas in T_i the path from the root to w spells out ay.

Example 9.21. The position and auxiliary trees for $x_2 = bbabb\$$ are shown in Fig. 9.22. (The numbers on the leaves are positions with respect to $x = abbabb\$$.) Note that leaves of the auxiliary tree are not necessarily leaves in the position tree, but nevertheless the vertex sets are the same. □

From the definition it is not clear that a position tree has an auxiliary tree; in fact, an arbitrary I-tree may not have one. The following theorem gives the conditions under which a tree possesses an auxiliary tree.

Theorem 9.11. A necessary and sufficient condition for an I-tree T to have an auxiliary tree is that if there is a vertex in T that corresponds to the string ax, where $a \in I$ and $x \in I^*$, then there is also a vertex that corresponds to the string x.

Proof. Exercise. □

Corollary. Every position tree has an auxiliary tree.

Proof. If ax is a prefix of the substring identifier for position i, then by Lemma 9.3(a), x is a prefix of the substring identifier for position $i + 1$. □

We are now ready to give an algorithm that computes T_i, the position tree for x_i, from T_{i+1}, the position tree for x_{i+1}.

Algorithm 9.5. Construction of T_i from T_{i+1}.

Input. A string $a_1 \cdots a_n a_{n+1}$, a position tree T_{i+1}, and an auxiliary tree A_{i+1} for $x_{i+1} = a_{i+1} \cdots a_n a_{n+1}$.

Output. T_i and A_i, the position and auxiliary trees for x_i.

Method

1. Find the leaf labeled $i + 1$ in T_{i+1}. (This was the last leaf added to T_{i+1}.)
2. Traverse the path from this leaf toward the root of T_{i+1} until a vertex u_y is encountered such that $B_{u_y}[a_i] = 1$. (Vertex u_y corresponds to the longest string y such that a_iy is a prefix of x_i and also a substring of x_{i+1} starting at some position k, $i < k \leq n$.) If no such vertex exists, continue to the root.
3. Set $B_v[a_i] = 1$ for each vertex v on the path from the leaf labeled $i + 1$ to the son of u_y on that path, or to the root if u_y does not exist. (Each vertex v on this path corresponds to some prefix z of x_{i+1}. Hence a_iz is clearly a substring of a_ix_{i+1}.)
4. i) If u_y does not exist, go to case 1.
   ii) Otherwise, if u_y has no a_i-son in the auxiliary tree A_{i+1}, go to case 2.
   iii) Otherwise, if u_y has an a_i-son v_y in the auxiliary tree, go to case 3. The vertex v_y corresponds to a_iy in the position tree T_{i+1}.

CASE 1. a_i is a symbol that does not occur in x_{i+1}. Thus the substring identifier for position i is just a_i. To construct T_i from T_{i+1}, do the following.

1. Create a new leaf, labeled i, the a_i-son of the root of T_{i+1}.
2. Make $B_i[a] = 0$ for all $a \in I$.

To construct A_i from A_{i+1}, make the new vertex labeled i the a_i-son of the root of A_{i+1}.

CASE 2. a_i appears in x_{i+1} but only a proper prefix of a_iy appears in T_{i+1} (spelling out a path from the root of T_{i+1} to some leaf v_1). This situation occurs when a substring identifier for some position k, $i < k \leq n$, must be lengthened to become the substring identifier for position k in T_i. Suppose $y = a_{i+1}a_{i+2} \cdots a_j$ and the leaf v_1 corresponds to $a_ia_{i+1} \cdots a_p$ for some $p < j$. Then the leaf v_1 is labeled k and $a_ia_{i+1} \cdots a_p$ is the substring identifier for position k in T_{i+1} (that is, $a_ia_{i+1} \cdots a_p = a_ka_{k+1} \cdots a_l$).

To construct T_i from T_{i+1} we add a subtree to vertex v_1 with two new leaves labeled i and k. The path from v_1 to k in this added subtree spells out $a_{l+1}a_{l+2} \cdots a_{m+1}$ and the path from v_1 to i spells out $a_{p+1}a_{p+2} \cdots a_{j+1}$, where $a_{l+1}a_{l+2} \cdots a_m = a_{p+1}a_{p+2} \cdots a_j$. Thus the path from the root of T_i to leaf k will spell out $a_ka_{k+1} \cdots a_ma_{m+1}$, which is the new substring identifier for position k in x_i, and the path from the root to leaf i will spell out $a_ia_{i+1} \cdots a_ja_{j+1}$, which is the substring identifier for position i in x_i.

Specifically, to construct T_i from T_{i+1}, do the following.

1. By following the path in T_{i+1} from u_y to the root find u_1, the first ancestor of u_y that has an a_i-son in A_{i+1}. Let this a_i-son be v_1. In T_{i+1} vertex v_1 is a leaf labeled k, for some $i < k \leq n$. Remove the label k from vertex v_1.
2. Let $u_1, u_2, \ldots, u_q, u_y$ be the sequence of vertices on the path from u_1 to u_y in T_{i+1}. Suppose the branch from u_h to u_{h+1} is labeled by c_h, $1 \leq h < q$, and the branch from u_q to u_y is labeled c_q, as shown in Fig. 9.23. ($c_1c_2 \cdots c_q = a_{p+1}a_{p+2} \cdots a_j = a_{l+1}a_{l+2} \cdots a_m$ in the discussion above.)
3. In T_i create q new vertices $v_2, v_3, \ldots, v_q, v_y$. Make v_{h+1} the c_h-son of v_h for $1 \leq h < q$. Make v_y the c_q-son of v_q.
4. Let j be i plus the depth of u_y, and let m be k plus the depth of u_y. Give T_i two new leaves labeled i and k; make leaf i the a_{j+1}-son of v_y and leaf k the a_{m+1}-son of v_y.
5. For all $a \in I$, make $B_v[a] = B_{v_1}[a]$ for all v in $\{v_2, v_3, \ldots, v_q, v_y, k\}$.
6. Make $B_i[a] = 0$ for all $a \in I$.

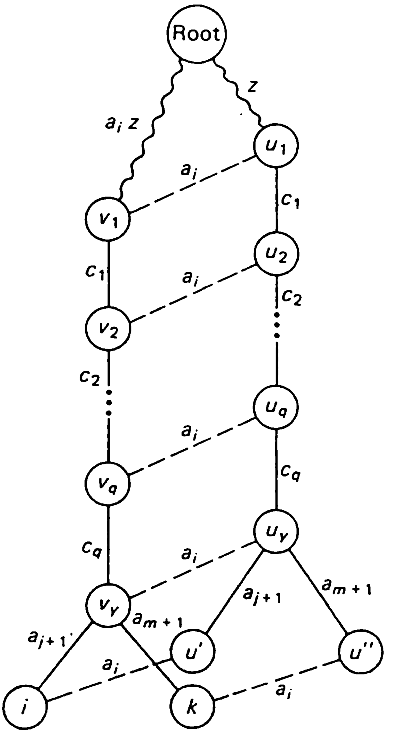


Fig. 9.23 Important parts of T_i in Case 2. The dashed lines indicate edges in A_i.

To construct A_i from A_{i+1} do the following.

7. Make v_y the a_i-son of u_y in A_i.
8. Let u' be the a_{j+1}-son of u_y in T_{i+1}. Make the new leaf labeled i the a_i-son of u' in A_i.
9. Let u'' be the a_{m+1}-son of u_y in T_{i+1}. Make the new leaf labeled k the a_i-son of u'' in A_i.
10. Make v_h the a_i-son of u_h in A_i for $2 \leq h \leq q$.

CASE 3. u_y has an a_i-son v_y in A_{i+1}. There are two situations to consider in this case.

a) v_y is a leaf labeled k in T_{i+1}. This situation occurs when $s_i(i) = a_ia_{i+1} \cdots a_ja_{j+1}$ and $s_{i+1}(k) = a_ka_{k+1} \cdots a_m$, where $a_ka_{k+1} \cdots a_m = a_ia_{i+1} \cdots a_j$; it is a special instance of case 2, in which $v_1 = v_y$. To construct T_i from T_{i+1} we remove the label k from v_y and give v_y an a_{j+1}-son labeled i and an a_{m+1}-son labeled k. The details of this construction are identical to those in case 2 with steps 3, 7 and 10 omitted.

b) v_y is an interior vertex in T_{i+1}. This case occurs when $S_i = S_{i+1} \cup \{s_i(i)\}$. To construct T_i from T_{i+1} we simply give v_y an a_{j+1}-son (which it does not have) and label this leaf i. The detailed algorithm is the following:
   1. Let j be i plus the depth of u_y.
   2. Make a new vertex labeled i the a_{j+1}-son of v_y in T_i. (Note that v_y cannot have an a_{j+1}-son in T_{i+1} by the maximality of y.)
   3. Make $B_i[a] = 0$ for all $a \in I$. Since a_iya_{j+1} is not a substring of x_{i+1}, surely aa_iya_{j+1} is not a substring of x_i for any a.
   4. To construct A_i from A_{i+1}, let vertex u' be the a_{j+1}-son of u_y in T_{i+1}. (In T_{i+1}, u' corresponds to ya_{j+1}.) Make the new vertex i the a_i-son of u' in A_{i+1}. (In A_i, i will correspond to $a_{j+1}y^Ra_i$.)

The relationships between the vertices mentioned in case 3(b) are depicted in Fig. 9.24. □

Example 9.22. Let $a_1 \cdots a_na_{n+1} = abbabb\$$. Let T_2 and A_2 be the trees in Fig. 9.22 (p. 351). From T_2 and A_2 we construct T_1 and A_1 using Algorithm 9.5. Thus $i = 1$ and $a_1 = a$ here.

First we locate the leaf labeled 2 and, since $B_2[a] = 0$, we set $B_2[a] = 1$ and move up to the vertex called u in Fig. 9.22. At vertex u, we find $B_u[a] = 1$, since abb is a substring of $bbabb\$$. Thus vertex u is u_y. We now look for v_y, the a-son of u in A_2. v_y does not exist so case 2 pertains.

Vertex w, the father of u in T_2, does not have an a-son in A_2. However, vertex r, the father of w, does—namely, the vertex 4. Thus at step 1 of case 2 we find that v_1 is the vertex labeled 4 in T_2 and A_2. At step 2 we find $q = 2$, $u_1 = r$, and $u_2 = w$. Also, $c_1 = c_2 = b$. Therefore at step 3, we create a new

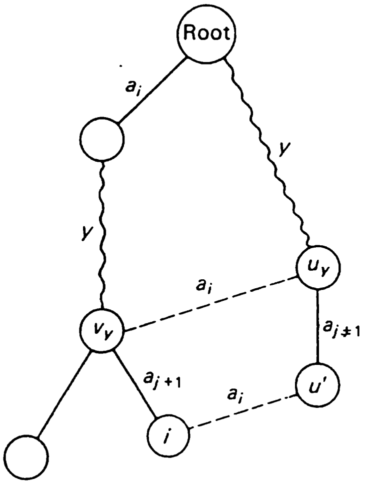


Fig. 9.24 Important parts of T_i in Case 3(b). The two dashed lines labeled a_i are edges in A_i.

vertex v, the b-son of the vertex labeled 4 in T_2 (it has lost this label in T_1). We also create v_y and make it the b-son of v.

At step 4, we compute $j = 3$. Since k, the former label of vertex v_1 is 4, we compute $m = 6$. We find $a_{j+1} = a$ and $a_{m+1} = \$$. Thus new vertices labeled 1 and 4 are made the a- and $\$$-sons of v_y, respectively. The tree T_1 is shown in Fig. 9.25. The remaining steps are left as an exercise. □

Lemma 9.4. If T_{i+1} and A_{i+1} are the position and auxiliary trees for x_{i+1}, then T_i and A_i as constructed by Algorithm 9.5 are the correct position and auxiliary trees for x_i.

Proof. Assume that T_{i+1} is a representation for S_{i+1}, the set of substring identifiers for the positions in x_{i+1}, and that A_{i+1} is the auxiliary tree for T_{i+1}. There are two possibilities:

1. $S_i = S_{i+1} \cup \{s_i(i)\}$.
2. $S_i = S_{i+1} - \{s_{i+1}(k)\} \cup \{s_i(k), s_i(i)\}$ for some $i < k \leq n$.

The first possibility is covered by cases 1 and 3(b) of Algorithm 9.5. The second possibility is covered by cases 2 and 3(a). The inferences needed to prove that cases 1-3 work correctly are included in the description of the algorithm. □

Thus to construct a position tree for an arbitrary string we can use the following procedure.

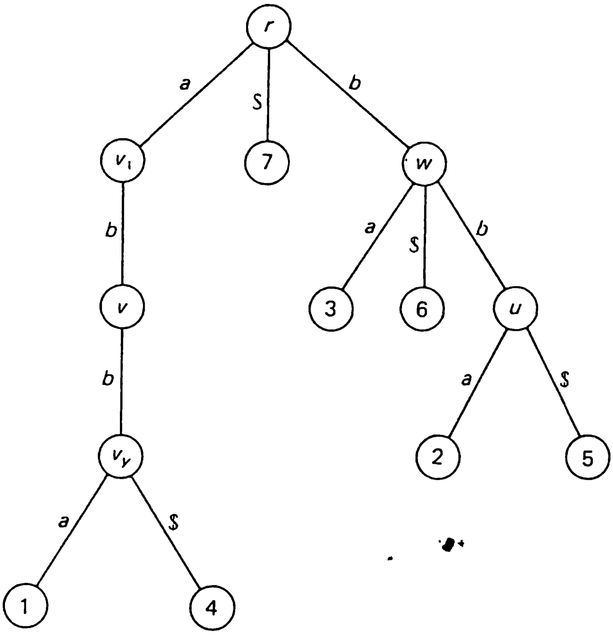


Fig. 9.25 Final position tree T_1.

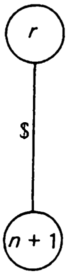


Fig. 9.26 Initialization of the position tree.

Algorithm 9.6. Construction of a position tree.

Input. A string $x\$ = a_1 \cdots a_n a_{n+1}$ with $a_i \in I$, $1 \leq i \leq n$ and $a_{n+1} = \$$.

Output. The position tree T_1 for $x\$$.

Method

1. Let T_{n+1} be the position tree of Fig. 9.26 and let $B_r[a] = B_{n+1}[a] = 0$ for all $a \in I$.
2. Let A_{n+1} be the auxiliary tree which is identical to that of Fig. 9.26.

3. **for** $i \leftarrow n$ **step** -1 **until** 1 **do** use Algorithm 9.5 to construct T_i and A_i from T_{i+1} and A_{i+1}. □

Theorem 9.12. Algorithm 9.6 constructs the position tree for $x\$$ in time proportional to the number of vertices in T_1, the final position tree.

Proof. In step 1 of Algorithm 9.5 we can find leaf $i+1$ in fixed time by keeping a pointer to this leaf when it is added to T_{i+1}. When we add leaf i to T_i, we set this pointer to leaf i.

This work in each execution of steps 2 and 3 is clearly proportional to the length of the path traveled from vertex $i+1$ to u_y, and the work in each execution of step 1 is constant. The time required for execution of any of cases 1-3 is proportional to the number of vertices added to the tree, as you may easily check. Thus the total time spent in all parts of Algorithm 9.5 except for steps 2 and 3 is proportional to the size of T_1.

It remains to show that the sum of the distances between vertices $i+1$ and u_y (or the root if u_y does not exist) in T_{i+1} for $1 \le i \le n$ is no greater than the size of T_1. Let these distances be $d_2, d_3, \ldots, d_{n+1}$, and let e_i, for $1 \le i \le n+1$, be the depth of the vertex i in T_i. A simple inspection of the cases 1-3 shows that

$$e_i \le e_{i+1} - d_{i+1} + 2. \tag{9.1}$$

If we sum both sides of (9.1) from i equal to 1 through n, we find that

$$\sum_{i=2}^{n+1} d_i \le 2n + e_{n+1} - e_1. \tag{9.2}$$

From (9.2) we immediately conclude that the time spent for steps 2 and 3 of Algorithm 9.5 is $O(n)$, and hence at most proportional to the size of T_1. □

As we mentioned, a position tree for a string of length n can have $O(n^2)$ vertices. Thus any pattern-matching algorithm in which such a tree is constructed would be $O(n^2)$ in time complexity. However, we may "compact" a position and auxiliary tree by condensing all chains in the position tree into a single vertex. A *chain* is a path all of whose vertices have exactly one son. It is not hard to show that a compact position tree for a string of length n has at most $4n-2$ vertices. The compact tree can represent the same information as the original position tree, and thus can be used in the same pattern-matching algorithms.

Algorithm 9.5 can be modified to produce the compact position and auxiliary tree in linear time. We do not give the modified algorithm here because of its similarity to Algorithm 9.5 and the fact that in many applications we may expect the size of the position tree to be linearly proportional to the length of the input string. The bibliographic notes give sources for the algorithm with compaction of the position tree, along with some of its applications.

EXERCISES

9.1 Give regular expressions and transition diagrams of finite automata for the following regular sets of strings over alphabet $I = \{a, b\}$.
a) All strings beginning and ending in a
b) All strings without two consecutive a's
*c) All strings with an odd number of a's and an even number of b's
*d) All strings not containing the substring aba

9.2 Prove that the set accepted by the NDFA of Fig. 9.1 (p. 320) is the set $(a + b)^*aba$.

9.3 Construct NDFA's accepting the following regular sets.
a) $(a + b)^*(aa + bb)$
b) $a^*b^* + b^*a^*$
c) $(a + \epsilon)(b + \epsilon)(c + \epsilon)$

9.4 Show that the complement of a regular set is a regular set.

*9.5 Show that the following sets of strings are not regular sets.
a) $\{a^nb^n \mid n \geq 1\}$
b) $\{ww \mid w \in \{a, b\}^*\}$
c) $\{w \mid w \in \{a, b\}^*$ and $w = w^R\}$ (i.e., the set of palindromes)

9.6 Let $x = a_1a_2 \cdots a_n$ be a given string and α a regular expression. Modify Algorithm 9.1 (p. 327) to find the least k and, having found k, the (a) least j and (b) greatest j such that $a_ja_{j+1} \cdots a_k$ is in the set denoted by α. [*Hint:* Associate an integer j with each state in S_i.]

*9.7 Let x and α be as in Exercise 9.6. Modify Algorithm 9.1 to find the least j and, having found j, the greatest k such that $a_ja_{j+1} \cdots a_k$ is in the set denoted by α.

9.8 Let x and α be as in Exercise 9.6. Construct an algorithm to find all substrings of x that are strings in the set denoted by α. What is the time complexity of your algorithm?

*9.9 Let I be an alphabet and $\emptyset$ be a symbol not in I. A *string with don't cares* (SWDC) is a string over $I \cup \{\emptyset\}$. We say SWDC $b_1b_2 \cdots b_n$ *matches* SWDC $a_1a_2 \cdots a_n$ at position i, if for all j, $i \leq j \leq n$, either $a_j = b_{j-i+1}$ or one of a_j and b_{j-i+1} is 0. Show that for a fixed alphabet I the problem of determining the set of positions at which one SWDC matches another is equivalent in asymptotic time complexity to *and-or multiplication*, i.e., computing $c_j = \bigvee_{i=1}^{n} d_i \wedge e_{j-i}$, where the c's, d's, and e's are Boolean.

**9.10 Show that determining the positions at which one SWDC matches another can be done in $O_A(n \log^2 n \log\log n)$ time. [*Hint:* Use Exercise 9.9 and the Schönhage-Strassen integer-multiplication algorithm.]

9.11 Given strings $a_1a_2 \cdots a_n$ and $b_1b_2 \cdots b_n$, find an $O(n)$ algorithm to determine whether there exists a k, $1 \leq k \leq n$, such that $a_i = b_{(k+i) \bmod n}$ for $1 \leq i \leq n$.

9.12 Use Algorithm 9.3 (p. 334) to construct machines for the following strings y.
a) $abbbbabbbabbaba$
b) $abcabcab$

9.13 Prove Theorem 9.8 (p. 334).

9.14 Let $S = \{y_1, y_2, \ldots, y_m\}$ be a set of strings. Design an algorithm to find all occurrences of strings in S as substrings of a text string x. What is the time complexity of your algorithm?

***9.15** Construct 2DPDA's to accept the following languages.

a) $\{xcy_1cy_2 \cdots cy_m \mid m \ge 1,\ x \in \{a, b\}^*,\ y_i \in \{a, b\}^*$ for $1 \le i \le m$, and at least one of $y_1, y_2, \ldots, y_m$ is a substring of $x\}$.

b) $\{xyy^R \mid x$ and y are I^*, with $|y| \ge 1\}$.

c) $\{x_1cx_2 \cdots cx_n \mid x_i \in \{a, b\}^*$ for $1 \le i \le n$ and all x_i's are distinct$\}$.

d) $\{x_1cx_2 \cdots cx_kdy_1cy_2 \cdots cy_l \mid$each x_i and y_i in $\{a, b\}^*$ and $x_i = y_j^R$ for some i and $j\}$.

9.16 Consider the 2DPDA P with rules:

$\delta(s_0, a, Z_0) = \delta(s_0, a, A) = (s_0, +1, \textbf{push } A),$
$\delta(s_0, \$, A) = (s_1, -1),$
$\delta(s_1, a, A) = \delta(s_1, a, Z_0) = (s_1, -1, \textbf{pop}),$
$\delta(s_1, ¢, Z_0) = (s_f, 0).$

a) List all the surface configurations of P with input aa.

b) Use Algorithm 9.4 (p. 343) to compute the array TERM.

***9.17** Suppose a 2DPDA has the ability to go from position i (n is the input length) to each of the following in a single move.

a) To position $n - $i

b) To position $i/2$

c) To position $n/2$

d) To position $\log n$

Show that none of these modifications increases the recognitive power of a 2DPDA. What other capabilities can you add without increasing the accepting power of a 2DPDA?

****9.18** Construct a 2DPDA to accept the language

$$\{w_1w_1^Rw_2w_2^Rw_3w_3^Ru \mid w_1, w_2, w_3 \in \{a, b\}^+ \text{ and } u \in \{a, b\}^*\}.$$

[*Hint:* Let x be an input string. Write a subroutine to find the shortest w_1 such that $w_1w_1^Ry = x$. Then apply this subroutine to y, using the subroutine recursively to determine the symbol immediately to the left of y.]

****9.19** Give linear time algorithms to recognize the following languages.

a) $\{ww^R \mid w \in I^*\}^*$

b) $\{ww^Ru \mid w, u \in I^*, |w| \ge 1\}$

c) $\{w_1w_2 \cdots w_n \mid w_i \in \{a, b\}^*$, and each w_i is a nonempty even length palindrome$\}$

9.20 Construct a linear algorithm which takes strings $a_1a_2 \cdots a_n$ and $b_1b_2 \cdots b_n$ and finds the largest l such that $a_1a_2 \cdots a_l$ is a substring of $b_1b_2 \cdots b_n$. How can the algorithm be used to test whether a given string is a palindrome?

The next four exercises discuss some variants of pushdown automata.

***9.21** A *k-headed* 2DPDA is a 2DPDA with k independent read heads on its input tape. Show that we can determine in $O(n^k)$ time whether an input string of length n is accepted by a k-headed 2DPDA.

9.22 A *one-way* pushdown automaton is one that never moves its input head to the left. A *nondeterministic* pushdown automaton is one that has zero or more choices for its next move. Thus there are four families of pushdown automata: 1DPDA, 1NPDA, 2DPDA, and 2NPDA. A language accepted by a 1NPDA is called a *context-free language.*

a) Show that $\{wcw^R | w \in \{a, b\}^*\}$ can be accepted by a 1DPDA.

b) Show that $\{ww^R | w \in \{a, b\}^*\}$ can be accepted by a 1NPDA. (This language cannot be accepted by any 1DPDA.)

c) Show that $\{wwx | w \text{ and } x \in \{a, b\}^*, |w| \geq 1\}$ can be accepted by a 2NPDA. (This language cannot be accepted by any 1NPDA.)

*9.23 Show that a language L is generated by a Chomsky normal form context-free grammar (p. 74) if and only if L is accepted by a 1NPDA.

*9.24 Show that we can determine in $O(n^3)$ time whether an input string of length n is accepted by a 2NPDA.

9.25 Find the position trees for these strings:

a) *baaaab*$.

b) *abababa*$.

9.26 Show that the position tree for $a^nb^na^nb^n\$$ has $n^2 + 6n + 2$ vertices.

*9.27 Show that a position tree for a random input string x has $O(|x|)$ vertices if all positions are chosen independently and uniformly from a fixed alphabet.

9.28 For the position trees of Exercise 9.25 find:

a) The auxiliary trees.

b) The bit vectors B_v for each vertex v.

9.29 Show that every position tree has an auxiliary tree.

9.30 Complete the proof of Lemma 9.4 (p. 355) by showing that T_i and A_i are correctly constructed from T_{i+1} and A_{i+1}.

*9.31 Show how the position tree for x may be used to test whether any of $y_1, y_2, \ldots, y_m$ is a substring of x in time proportional to

$$|y_1| + |y_2| + \cdots + |y_m|.$$

9.32 Design an algorithm to find a longest common substring of two given strings x and y. What is the time complexity of your algorithm?

9.33 Given a string x and a string $b_1b_2 \cdots b_p$, design an efficient algorithm to find for each i, $1 \leq i \leq p$, a longest substring of x that is a prefix of $b_ib_{i+1} \cdots b_p$.

9.34 Given two strings $x = a_1a_2 \cdots a_n$ and $y = b_1b_2 \cdots b_p$ over alphabet I, design an efficient algorithm to find a shortest representation for y as a string $c_1c_2 \cdots c_m$, where each c_i is a symbol in I or a symbol denoting a substring of x. For example, if $x = abbabb$ and $y = ababbaa$, then $[1:2][4:6]\,aa$ is a representation for y of length 4. The symbol $[i:j]$ denotes the substring $a_ia_{i+1} \cdots a_j$ of x. [*Hint:* Use Exercise 9.33.]

9.35 Prove that a compact position tree for a string of length n has at most $4n - 2$ vertices.

**9.36 Design an $O(n)$ algorithm to construct a compact position tree for a string of length n.

9.37 A string $x = a_1a_2 \cdots a_n$ is a *subsequence* of string $y = b_1b_2 \cdots b_p$ if $a_1a_2 \cdots a_n = b_{i_1}b_{i_2} \cdots b_{i_n}$ for some $i_1 < i_2 < \cdots < i_n$ (i.e., x is y with zero or more symbols deleted). Design an $O(|x| \cdot |y|)$ algorithm to find a longest common subsequence of x and y.

9.38 Given two strings x and y, design an algorithm to determine a shortest sequence of insertions and deletions of single symbols that will transform x into y.

Research Problems

9.39 Can the time bound in Exercise 9.10 be improved?

9.40 Is $O(|\alpha| \cdot |x|)$ time necessary to determine whether a string in the language denoted by regular expression α is a substring of x?

9.41 A k-headed 2DPDA can be simulated in $O(n^k)$ time on a RAM (Exercise 9.21). Can every context-free language be accepted by a two-headed 2DPDA? If so, then every context-free language could be accepted in $O(n^2)$ time on a RAM.

9.42 Is there a context-free language that cannot be accepted by a 2DPDA?

9.43 Is there a language that can be accepted by a 2NPDA that cannot be accepted by a 2DPDA?

9.44 Can the time bound in Exercise 9.37 be improved? See Aho, Hirschberg, and Ullman [1974] for bounds using the decision tree model.

BIBLIOGRAPHIC NOTES

The equivalence between finite automata and regular expressions is from Kleene [1956]. Nondeterministic finite automata were studied by Rabin and Scott [1959], who showed their equivalence to deterministic ones. The regular expression pattern-matching algorithm (Algorithm 9.1) is an abstraction of an algorithm by Thompson [1968]. Ehrenfeucht and Zeiger [1974] discuss the complexity of the NDFA as a pattern-specifying device. The linear match of one string against another (Algorithm 9.3) is from Morris and Pratt [1970]. The linear simulation of 2DPDA's is by Cook [1971a], as is its extension in Exercise 9.21. Properties of 2DPDA's have been studied by Gray, Harrison, and Ibarra [1967]. An $O(n^3)$ simulation for two-way nondeterministic PDA's (Exercise 9.24) is in Aho, Hopcroft, and Ullman [1968]. Exercise 9.15(b) is due to D. Chester and Exercise 9.19(c) to V. Pratt. A solution to Exercise 9.19(c) is contained in Knuth and Pratt [1971].

The material in Section 9.5 on position trees is due to Weiner [1973], as is the notion of compact position trees. Solutions to Exercises 9.20 and 9.31–9.36, along with several other applications, can be found there and in Knuth [1973b]. The paper by Karp, Miller, and Rosenberg [1972] also contains some interesting pattern-matching algorithms concerning repeated substrings. Exercises 9.9 and 9.10 on matching strings with don't cares are due to Fischer and Paterson [1974]. Wagner and Fischer [1974] contains a solution to Exercise 9.38. That paper and Hirschberg [1973] discuss algorithms for the common subsequence problem, Exercise 9.37.

NP-COMPLETE PROBLEMS

CHAPTER 10

How much computation should a problem require before we rate the problem as being truly difficult? There is general agreement that if a problem cannot be solved in less than exponential time, then the problem should be considered completely intractable. The implication of this "rating scheme" is that problems having polynomial-time-bounded algorithms are tractable. But bear in mind that although an exponential function such as 2^n grows faster than any polynomial function of n, for small values of n an $O(2^n)$-time-bounded algorithm can be more efficient than many polynomial-time-bounded algorithms. For example, 2^n itself does not overtake n^{10} until n reaches 59. Nevertheless, the growth rate of an exponential function is so explosive that we say a problem is *intractable* if all algorithms to solve that problem are of at least exponential time complexity.

In this chapter we shall give evidence that a certain class of problems, the class of nondeterministic polynomial-time complete ("NP-complete" for short) problems, is quite likely to contain only intractable problems. This class of problems includes many "classical" problems in combinatorics, such as the traveling salesman problem, the Hamilton circuit problem, and integer linear programming, and all problems in the class can be shown "equivalent," in the sense that if one problem is tractable, then all are. Since many of these problems have been studied by mathematicians and computer scientists for decades, and no polynomial-time-bounded algorithm has been found for even one of them, it is natural to conjecture that no such polynomial algorithms exist, and consequently, to regard all the problems in this class as being intractable.

We shall also consider a second class of problems, called the "polynomial-space complete" problems, which are at least as hard as the NP-complete problems, yet still not provably intractable. In Chapter 11 we exhibit certain problems which we can actually prove are intractable.

10.1 NONDETERMINISTIC TURING MACHINES

For reasons which will soon become clear, the key notion behind the theory of NP-complete problems is the nondeterministic Turing machine.† We have already discussed nondeterministic finite automata, for which each move could take an automaton to one of several states. In analogy, a nondeterministic Turing machine has a finite number of moves from which to choose at each step. An input string x is deemed accepted if at least one sequence of moves with x as input leads to an accepting instantaneous description (ID).

On a given input x, we can think of a nondeterministic Turing machine M as executing all possible sequences of moves in parallel until either an accepting ID is reached or no more moves are possible. That is to say, after i moves

† The reader not familiar with Turing machines is encouraged to review Section 1.6.

we can think of a number of "copies" of M being in existence. Each copy represents an ID in which M can be after i moves. On the $(i + 1)$st move a copy C replicates itself into j copies if, in ID C, the Turing machine has j choices for a next move.

Thus the possible sequences of moves that M can make on input x can be arranged into a tree of ID's. Each path from the root to a leaf in the tree represents a sequence of possible moves. If σ is a shortest sequence of moves that terminates in an accepting ID, then as soon as M has made $|\sigma|$ moves, M halts and accepts the input x. The time "spent" in processing x is the length of σ. If on input x no sequence of moves leads to an accepting ID, then M rejects x and the time spent in processing x is left undefined.

It is often convenient to think of M as "guessing" only the moves in the sequence σ and verifying that σ indeed terminates in an accepting ID. However, since a deterministic machine cannot normally guess an accepting sequence of moves in advance, a deterministic simulation of M would require tracing out the tree of all possible sequences of moves on x, in some order, until a shortest sequence that terminated in an accepting ID was found. If no sequence of moves leads to acceptance, then a deterministic simulation of M could run forever unless there is some *a priori* bound on the length of a shortest accepting sequence. Thus it is natural to suspect that nondeterministic Turing machines are capable of performing tasks that cannot be done by any deterministic machine of equal time or space complexity. It is, however, a major open question whether there are languages that are accepted by a nondeterministic Turing machine of fixed time or space complexity but by no deterministic Turing machine of the same complexity.

Definition. A *k-tape nondeterministic Turing machine* (NDTM for short) M is a seven-tuple $(Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ where all components have the same meaning as for the ordinary deterministic Turing machine, except that here the next-move function δ is a mapping from $Q \times T^k$ to subsets of $Q \times (T \times \{L, R, S\})^k$. That is, given a state and list of k tape symbols, δ returns a finite set of choices of next move; each choice is a new state, with k new tape symbols and k moves of the tape heads. Note that the NDTM M may choose any of these moves, but it cannot choose a next state from one and new tape symbols from another, or make any other combination of moves.

Instantaneous descriptions of an NDTM are defined exactly as for a eterministic Turing Machine (*DTM*). An NDTM $M = (Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ makes a *move* by determining the current state, say q, and the symbols scanned y each of the k tape heads, say $X_1, X_2, \ldots, X_k$. Then it selects some element $(r, (Y_1, D_1), \ldots, (Y_k, D_k))$ from the set $\delta(q, X_1, X_2, \ldots, X_k)$. This articular element specifies that the new state is to be r, Y_i is to be printed n place of X_i on the ith tape and the ith head is to move in the direction indicated by D_i, for $1 \leq i \leq k$. If ID C can become ID D for some choice of next move, then we write $C \vdash_M D$. (The subscript M will often be omitted from $\vdash$.) Note that there may be several D's such that $C \vdash D$ for NDTM M, but if M is deterministic, there is at most one such D for each C.

We write $C_1 \overset{*}{\vdash} C_k$ if $C_1 \vdash C_2 \vdash \cdots \vdash C_k$ for some $k > 1$, or if $C_1 = C_k$. The NDTM M *accepts* string w if $(q_0w, q_0, q_0, \ldots, q_0) \overset{*}{\vdash} (\alpha_1, \alpha_2, \ldots, \alpha_k)$, where α_1 (and hence $\alpha_2, \alpha_3, \ldots, \alpha_k$) has the final state symbol q_f somewhere within. The *language accepted by* M, denoted $L(M)$, is the set of strings accepted by M.

Example 10.1. Let us design an NDTM to accept strings of the form

$$10^{i_1}10^{i_2} \cdots 10^{i_k}$$

such that there is some set $I \subseteq \{1, 2, \ldots, k\}$ for which $\Sigma_{j \in I} i_j = \Sigma_{j \notin I} i_j$. That is, a string w is to be accepted if the list† of integers $i_1, i_2, \ldots, i_k$ represented by w can be partitioned into two sublists such that the sum of the integers on one list equals the sum of the integers on the other. This problem is known as the partition problem. It is a problem which has been shown NP-complete when the integers are encoded in binary and the size of the problem is the length of the list of binary integers.‡

We shall design a three-tape NDTM M to recognize this language. It scans its input tape from left to right, and each time a block of 0's 0^{i_j} is reached, i_j 0's will be appended to either tape 2 or tape 3, nondeterministically. When the end of the input is reached, the NDTM will check whether it has placed an equal number of 0's on tapes 2 and 3 and if so will accept. Thus, if any sequence of choices to put the various i_j's into one set (tape 2) or the other (tape 3) leads to equal sums, the NDTM will accept. The sequences of moves leading to unequal strings on tapes 2 and 3 are of no concern, as long as at least one sequence of choices works. Formally, let

$$M = (\{q_0, q_1, \ldots, q_5\}, \{0, 1, b, \$\}, \{0, 1\}, \delta, b, q_0, q_5),$$

where δ, the next-move function, is shown in Fig. 10.1.

Figure 10.2 shows two of the many sequences of moves which may be made by the NDTM on input 1010010. The first leads to acceptance, the second does not. Since there is at least one sequence of moves leading to acceptance, the NDTM accepts 1010010. □

† It is a list rather than a set because there may be repetitions.

‡ As we shall see, the encoding used to represent a problem is all-important. It is not hard to show that the language accepted by the NDTM of Example 10.1 is in fact of time complexity $O_{\mathrm{TM}}(n^2)$, where n is the length of the input and a DTM is used. However, if we encode the input in binary, the length of the input becomes the sum of the logarithms of $i_1, \ldots, i_k$ and the same strategy yields an $O_{\mathrm{TM}}(c^n)$ algorithm, where n is now the length of the binary input and $c > 1$.

Definition. We say that an NDTM M is of *time complexity* $T(n)$ if for every accepted input string of length n there is some sequence of at most $T(n)$ moves leading to the accepting state. M is of *space complexity* $S(n)$ if for every accepted input of length n there is some sequence of moves leading to acceptance in which at most $S(n)$ different cells are scanned on any one tape.

xample 10.2. The NDTM of Example 10.1 is of time complexity $2n+2$ he worst case is an input of n 1's) and space complexity $n+1$. Other rea-)nable encodings of the partition problem also yield this same complexity. or example, let $B(i)$ be the binary encoding of the integer i. Let

$${}_1 = \Big\{\#B(i_1)\#B(i_2)\ \ldots\ \#B(i_k) \mid \text{there exists a set } I \subseteq \{1, 2, \ldots, k\} \text{ such that } \sum_{j\in I} i_j = \sum_{j\in I} i_j\Big\}$$

'here # is a special marker symbol. To recognize L_1 we can design a new IDTM M_1 whose operation is similar to the NDTM M of Fig. 10.1. How- ver, instead of copying 0's onto tape 2 or 3, M_1 stores a binary number on ipes 2 and 3. Each new binary number encountered on the input is added) the number on one tape or the other.

To process the list of integers $i_1, i_2, \ldots, i_k$, M_1 would use the input string $= \#B(i_1)\#B(i_2)\ \ldots\ \#B(i_k)$. To process the same problem, M would use ıe input string $w = 10^{i_1}10^{i_2}\ \ldots\ 10^{i_k}$ which can be exponentially longer than x. 'hus, although M_1 is in some sense faster by an exponential factor than the IDTM of Fig. 10.1, its time and space complexities are still $O(n)$, where n ; the length of the input, since the input has been shortened accordingly. □

We can show by means of a simulation that any language accepted by an IDTM is also accepted by a DTM, but it seems that a heavy price must be 'aid in terms of time complexity. The smallest upper bound we can show or such a simulation is exponential. That is, if $T(n)$ is a reasonable time omplexity function (reasonable in the sense of being "time-constructible," term we shall define in Chapter 11), then for each $T(n)$-time-bounded NDTM M we can find a constant c and a DTM M' such that $L(M) = L(M')$ and M' s of time complexity $O_{\mathrm{TM}}(c^{T(n)})$.

A proof of this result can be obtained by constructing a DTM M' to simu- ate M by an exhaustive enumeration algorithm. There is some constant d uch that M has no more than d choices of next move in any situation. Thus ı sequence of up to $T(n)$ moves of M can be represented by a string over the ılphabet $\Sigma = \{0, 1, \ldots, d-1\}$ of length up to $T(n)$. M' simulates the be- ıavior of M on an input x of length n as follows. M' successively generates ıll strings in Σ^* of length at most $T(n)$ in lexicographic order. There are no nore than $(d+1)^{T(n)}$ such strings. As soon as a new string w is generated, M' simulates σ_w, the sequence of moves of M represented by w. If σ_w causes

	Current symbol			(New symbol, head move)				
State	Tape 1	Tape 2	Tape 3	Tape 1	Tape 2	Tape 3	New State	Comments
q_0	1	b	b	1,S	$,R	$,R	q_1	Mark left ends of tapes 2 and 3 with $, then go to state q_1.
q_1	1	b	b	1,R	b,S	b,S	q_2	Here we choose whether to write the next block on tape 2 (q_2) or tape 3 (q_3).
				1.R	b.S	b.S	q_3	
q_2	0	b	b	0.R	0,R	b,S	q_2	Copy the block of 0's onto tape 2, then return to state q_1 when 1 is reached on tape 1. If b is reached on tape 1, instead go to state q_4 to compare the lengths of tapes 2 and 3.
	1	b	b	1,S	b,S	b,S	q_1	
	b	b	b	b,S	b,L	b,L	q_4	
q_3	0	b	b	0,R	b,S	0;R	q_3	The same as for state q_2, but write on tape 3.
	1	b	b	1,S	b,S	b,S	q_1	
	b	b	b	b.S	b,L	b,L	q_4	
q_4	b	0	0	b,S	0,L	0,L	q_4	Compare the length of tapes 2 and 3
	b	$	$	b.S	$.S	$.S	q_5	
q_5								Accept

Fig. 10.1. Next moves of an NDTM. Each line represents one choice.

M to accept x, then M' also accepts x. If σ_w does not represent a valid sequence of moves by M, or if σ_w does not cause M to accept x, then M' repeats the process with the next string in Σ^*.

M' can simulate σ_w in time $O_{\text{TM}}(T(n))$. It takes at most $O_{\text{TM}}(T(n))$ time to generate each string w. Thus the entire simulation of NDTM M can take time $O_{\text{TM}}(T(n)(d+1)^{T(n)})$, which is at most $O_{\text{TM}}(c^{T(n)})$ for some constant c. We leave the details of the simulation for an exercise.

We should, however, emphasize that no nontrivial lower bound on the simulation of an NDTM by a DTM is known. That is, we know of no language L which can be accepted by some NDTM of time complexity $T(n)$ but which we can prove is not accepted by any DTM of time complexity $T(n)$. The situation with regard to space complexity is much more pleasant.

Definition. We say a function $S(n)$ is *space-constructible* if there is a DTM M which when given an input of length n will place a special marker symbol on the $S(n)$th tape square of one of its tapes without using more

$(q_0 1010010, q_0, q_0)$	$(q_0 1010010, q_0, q_0)$
$\vdash (q_1 1010010, \$q_1, \$q_1)$	$\vdash (q_1 1010010, \$q_1, \$q_1)$
$\vdash (1q_2 010010, \$q_2, \$q_2)$	$\vdash (1q_3 010010, \$q_3, \$q_3)$
$\vdash (10q_2 10010, \$0q_2, \$q_2)$	$\vdash (10q_3 10010, \$q_3, \$0q_3)$
$\vdash (10q_1 10010, \$0q_1, \$q_1)$	$\vdash (10q_1 10010, \$q_1, \$0q_1)$
$\vdash (101q_3 0010, \$0q_3, \$q_3)$	$\vdash (101q_3 0010, \$q_3, \$0q_3)$
$\vdash (1010q_3 010, \$0q_3, \$0q_3)$	$\vdash (1010q_3 010, \$q_3, \$00q_3)$
$\vdash (10100q_3 10, \$0q_3, \$00q_3)$	$\vdash (10100q_3 10, \$q_3, \$000q_3)$
$\vdash (10100q_1 10, \$0q_1, \$00q_1)$	$\vdash (10100q_1 10, \$q_1, \$000q_1)$
$\vdash (101001q_2 0, \$0q_2, \$00q_2)$	$\vdash (101001q_3 0, \$q_3, \$000q_3)$
$\vdash (1010010q_2, \$00q_2, \$00q_2)$	$\vdash (1010010q_3, \$q_3, \$0000q_3)$
$\vdash (1010010q_4, \$0q_4 0, \$0q_4 0)$	$\vdash (1010010q_4, q_4\$, \$000q_4 0)$
$\vdash (1010010q_4, \$q_4 00, \$q_4 00)$	Halt, no next ID
$\vdash (1010010q_4, q_4\$00, q_4\$00)$	
$\vdash (1010010q_5, q_5\$00, q_5\$00)$	
Accept	

Fig. 10.2. Two legal sequences of moves for an NDTM.

than $S(n)$ cells on any tape. Most common functions, e.g., polynomials, 2^n, $n!$, $\lceil n \log (n+1) \rceil$, are space-constructible.

We can show that if $S(n)$ is a constructible space complexity function, and M is an NDTM of space complexity $S(n)$, then there is a DTM M' such that $L(M) = L(M')$ and M' is of $O(S^2(n))$ space complexity.

One strategy by which M' can simulate M is an interesting application of divide-and-conquer. If a k-tape NDTM $M = (Q, T, I, \delta, \text{b}, q_0, q_f)$ is of constructible complexity $S(n)$, then there is some constant c such that the number of distinct ID's which M need enter when started with an input of length n is at most $c^{S(n)}$. A more precise bound is $\|Q\| \times (\|T\| + 1)^{kS(n)} \times (S(n))^k$, where the first factor represents the number of states, the second bounds the number of possible tape contents, and the last bounds the number of possible head positions. Thus, if $C_1 \overset{*}{\underset{M}{\vdash}} C_2$, then there is some way for M to go from ID C_1 to ID C_2 in at most $c^{S(n)}$ moves. We can test whether $C_1 \overset{*}{\underset{M}{\vdash}} C_2$ in up to $2i$ moves by testing, for all C_3, whether $C_1 \overset{*}{\underset{M}{\vdash}} C_3$ in up to i moves and $C_3 \overset{*}{\underset{M}{\vdash}} C_2$ in up to i moves.

```
procedure TEST($C_1$, $C_2$, $i$):
if $i = 1$ then
    if $C_1 \vdash C_2$ or $C_1 = C_2$ then return true
    else return false
else
    begin
        for each ID $C_3$ with no more than $S(n)$ cells used on any tape do
            if TEST($C_1$, $C_3$, $\lceil i/2 \rceil$) and TEST($C_3$, $C_2$, $\lfloor i/2 \rfloor$) then
              return true;
        return false
    end
```

Fig. 10.3. The procedure TEST.

The strategy behind M', then, is to determine by the following algorithm if a given initial ID C_0 goes to some accepting ID.

Algorithm 10.1. Deterministic simulation of an NDTM.

Input. An NDTM M of space complexity $S(n)$, where $S(n)$ is space-constructible, and an input string w of length n.

Output. "Yes" if $w \in L(M)$, "no" otherwise.

Method. The recursive procedure TEST(C_1, C_2, i) in Fig. 10.3 determines whether $C_1 \overset{*}{\underset{M}{\vdash}} C_2$ in at most i steps. If so, its value is **true**, otherwise its value is **false**. Throughout the algorithm C_1 and C_2 are ID's in which no more than $S(n)$ cells are used on any tape.

The complete algorithm is to call TEST(C_0, C_f, $c^{S(n)}$), for each accepting ID C_f, where C_0 is the initial ID of M with input w. If any one of these calls is found to have the value **true**, then answer "yes," otherwise answer "no." □

Theorem 10.1. If M is an NDTM of constructible space complexity $S(n)$, then there is a DTM M' of space complexity $O(S^2(n))$, such that $L(M) = L(M')$.

Proof. The proof is a Turing machine implementation of Algorithm 10.1. We know from Section 2.5 how to simulate the recursive procedure TEST on a RAM. We can use the same stacking strategy on a Turing machine. Since the space-consuming arguments to TEST are ID's of length $O(S(n))$, stack frames of this size must be laid out on a tape; the fact that $S(n)$ is space-constructible assures us we can do this. An examination of TEST shows that each time TEST is called, its third argument is essentially half the third argument of the calling instance of TEST. Thus it may be shown that the number of stack frames at any one time does not exceed $1 + \log \lceil c^{S(n)} \rceil$, which is $O(S(n))$. Since $O(S(n))$ cells per frame are used, the total number of cells used by the Turing machine for the stack is $O(S^2(n))$. The remaining details of the Turing machine construction are left for an exercise. □

In order to simplify proofs, it is often desirable to restrict our attention to single-tape Turing machines. The next lemma allows us to do this if we are willing to pay by an increase in computing time.

Lemma 10.1. If L is accepted by a k-tape NDTM $M = (Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ of time complexity $T(n)$, then L is accepted by a single-tape NDTM of time complexity $O(T^2(n))$.

Proof. We construct a single-tape NDTM M_1 of time complexity $O(T^2(n))$ which accepts L. The strategy involved is to think of M_1's tape as if it had $2k$ "tracks" as shown in Fig. 10.4. That is, the tape symbols for M_1 are $2k$-tuples whose odd-numbered components are symbols from T and whose even-numbered components are either the blank b or a special marker symbol #. The k odd-numbered tracks correspond to the k tapes of M and each even-numbered track contains all b's except for one occurrence of #. The # on track $2j$ marks the location of the tape head of M on tape j, which corresponds to track $2j - 1$. In Fig. 10.4 we have shown the jth tape head scanning cell i_j of the jth tape for each j, $1 \leq j \leq k$.

M_1 simulates one move of M as follows. Initially, suppose that the tape head of M_1 is at the cell containing the leftmost tape head of M.

1. The tape head of M_1 moves right, until it has passed over all k head markers on the even-numbered tracks. As it goes, M_1 records in its own state the symbols scanned by each of M's tape heads.
2. The action of M_1 in step 1 is deterministic. Now M_1 makes a nondeterministic "branch." Specifically, on the basis of the state of M, which M_1 has recorded in its own state, and of the tape symbols scanned by

Tape 1 of M	X_{11}	X_{12}	. . .	X_{1i_1}	. . .
Head for tape 1	b	b	. . .	#	. . .
Tape 2 of M	X_{21}	X_{22}	. . .	X_{2i_2}	. . .
Head for tape 2	b	b	. . .	#	. . .
			.		
Tape k of M	X_{k1}	X_{k2}	. . .	X_{ki_k}	. . .
Head for tape k	b	b	. . .	#	. . .

Fig. 10.4. The tape of M_1.

M, which M_1 has just determined, M_1 nondeterministically selects a legal move of M. For each legal move of M in this situation, M_1 has one next state which it may enter.

3. Having selected a move of M to simulate, M_1 changes the state of M recorded in its own state, according to the selected move. Then M_1 moves its tape head left until it has again passed over the k head markers. Each time a head marker is found, M_1 changes the tape symbol on the track above, and moves the head marker at most one cell left or right, in conformity with the selected move.

At this point, M_1 has simulated one move of M. Its tape head is at most two cells to the right of the leftmost head marker, so that marker may be found and the cycle repeated.

M_1 may accept if M accepts, since M_1 records M's state. If M accepts a string w of length n, it does so with a sequence of no more than $T(n)$ moves. Clearly, in a sequence of $T(n)$ moves, the tape heads of M cannot get more than $T(n)$ cells apart, so M_1 can simulate one move of the sequence in at most $O(T(n))$ of its own moves. Thus M_1 accepts w by a sequence of at most $O(T^2(n))$ moves. It follows that M_1 accepts language L and is of time complexity $O(T^2(n))$. □

Corollary 1. If L is accepted by a k-tape DTM of time complexity $T(n)$, then L is accepted by a single-tape DTM of time complexity $O(T^2(n))$.

Proof. In the proof above, if M is deterministic, then so is M_1. □

Corollary 2. If L is accepted by a k-tape NDTM of space complexity $S(n)$, then L is accepted by a single-tape NDTM of space complexity $S(n)$.

Corollary 3. If L is accepted by a k-tape DTM of space complexity $S(n)$, then L is accepted by a single-tape DTM of space complexity $S(n)$.

10.2 THE CLASSES $\mathscr{P}$ AND $\mathscr{NP}$

We now introduce two important classes of languages.

Definition. We define $\mathscr{P}$-TIME to be the set of all languages which can be accepted by DTM's of polynomial time complexity. That is,

$$\mathscr{P}\text{-TIME} = \{L \mid \text{there exists a DTM } M \text{ and a polynomial } p(n) \text{ such that } M \text{ is of time complexity } p(n) \text{ and } L(M) = L\}.$$

We define $\mathscr{NP}$-TIME to be the set of all languages which can be accepted by NDTM's of polynomial time complexity. We shall frequently abbreviate $\mathscr{P}$-TIME and $\mathscr{NP}$-TIME to $\mathscr{P}$ and $\mathscr{NP}$, respectively.

First, we observe that although $\mathscr{P}$ and $\mathscr{NP}$ have been defined in terms of Turing machines, we could have used any of a number of models of computation. Intuitively, we can think of $\mathscr{P}$ as being that class of languages which can be recognized in polynomial time. For example, we showed that under the logarithmic cost, if a language L is of time complexity $T(n)$ on a Turing machine, then the time complexity of L on a RAM or RASP lies between $k_1T(n)$ and $k_2T^4(n)$, for positive constants k_1 and k_2. Thus under the logarithmic cost criterion, L can be accepted by a Turing machine of polynomial time complexity if and only if L has a polynomial algorithm on a RAM or RASP.

It is also possible to define a "nondeterministic" RAM or RASP by adding to its repertoire an instruction

$$\text{CHOICE } (L_1, L_2, \ldots, L_k)$$

which causes one of the statements labeled $L_1, L_2, \ldots, L_k$ to be nondeterministically selected and executed. Thus one can also define the class $\mathscr{NP}$ by polynomial-time-bounded nondeterministic RAM's or RASP's under the logarithmic cost criterion.

We can, therefore, imagine a nondeterministic computer, such as a RAM or RASP, which can make many different possible sequences of moves starting from a given initial ID. Such a device appears capable of recognizing in polynomial time many languages that are apparently not recognizable in polynomial time by deterministic algorithms. Of course, any attempt to directly simulate a nondeterministic device N by a deterministic device D which executes all possible sequences of moves uses much more time than does any one copy of N, as D must keep track of a profusion of copies of N. The best we can say from the results of the previous section is that if L is in $\mathscr{NP}$, then L is accepted by some DTM of time complexity $k^{p(n)}$ for some constant k and polynomial p, k, and p, each depending on L.

On the other hand, no one has yet been able to prove that there is a language in $\mathscr{NP}$ that is not in $\mathscr{P}$. That is, it is not known whether $\mathscr{P}$ is properly contained in $\mathscr{NP}$. However, we can show that certain languages are as "hard" as any in $\mathscr{NP}$, in the sense that if we had a deterministic polynomial-time-bounded algorithm to recognize one of these languages, then we could find a deterministic polynomial-time-bounded algorithm to recognize any language in $\mathscr{NP}$. These languages are called "NP-complete."

Definition. A language L_0 in $\mathscr{NP}$ is *nondeterministic polynomial-time complete* (*NP-complete* for short) if the following condition is satisfied: If we are given a deterministic algorithm of time complexity $T(n) \geq n$ to recognize L_0, then for every language L in $\mathscr{NP}$ we can effectively find a deterministic algorithm of time complexity $T(p_L(n))$, where p_L is a polynomial that depends on L. We say L is *reducible* to L_0.

One way to prove that a language L_0 is NP-complete is to show that L_0 is in $\mathscr{NP}$ and that every language L in $\mathscr{NP}$ can be "polynomially transformed" to L_0.

Definition. We say that a language L is *polynomially transformable* to L_0 if there is a deterministic polynomial-time-bounded Turing machine M which will convert each string w in the alphabet of L into a string w_0 in the alphabet of L_0, such that w is in L if and only if w_0 is in L_0.

If L is transformable to L_0, and L_0 is accepted by some deterministic algorithm A of time complexity $T(n) \geq n$, then we can determine whether w is in L by letting M transform input w into w_0 and then using A to determine whether w_0 is in L_0. If M is $p(n)$-time-bounded, then $|w_0| \leq p(|w|)$. Thus there is an algorithm to determine whether w is in L which takes time $p(|w|) + T(p(|w|)) \leq T(2p(|w|))$. If T is a polynomial (that is, L_0 is in $\mathscr{P}$) then the algorithm to accept L would be polynomial-time-bounded, so L would also be in $\mathscr{P}$.

Some authors actually define a language L_0 to be NP-complete if L_0 is in $\mathscr{NP}$ and every language in $\mathscr{NP}$ is polynomially transformable to L_0. This definition appears more restrictive than the earlier one, although it is not known whether the class of NP-complete languages is different under the two definitions. The "reduces"definition implies that if M_0 is a deterministic $T(n)$-time-bounded Turing machine for an NP-complete language L_0, then every language in $\mathscr{NP}$ can be recognized in $T(p(n))$ time, for some polynomial p, by a deterministic Turing machine that can call M_0 as a subroutine, zero or more times. The "transforms" definition implies that M_0 can be used only once, and then only in a restricted way. Although we shall adopt the broader definition, all of our proofs of NP-completeness reflect the narrower definition.

Under either definition it should be clear that if there is a deterministic polynomial-time-bounded algorithm to recognize L_0, then all languages in $\mathscr{NP}$ can be recognized in polynomial time. Thus either all NP-complete languages are in $\mathscr{P}$ or none are. The former is true if and only if $\mathscr{P} = \mathscr{NP}$. Unfortunately, at this time we can only conjecture that $\mathscr{P}$ is properly contained in $\mathscr{NP}$.

10.3 LANGUAGES AND PROBLEMS

We have defined $\mathscr{P}$ and $\mathscr{NP}$ to be classes of languages. The reason for doing so is twofold. First, it simplifies notation. Second, problems from diverse disciplines such as graph theory and number theory can often be couched as language recognition problems. For example, consider a problem which requires a "yes" or "no" answer for each instance. We can encode each instance of such a problem as a string and reformulate the original problem as one of recognizing the language consisting of all strings representing those instances of the problem whose answer is "yes." We have already seen such encodings in Chapter 9 where we formulated a number of pattern-matching problems in terms of language recognition problems. However, we must be careful in selecting the encoding because the time complexity of a problem can depend on the encoding used.

To make the problem–language relationship more explicit, we define some common "standard" representations for problems. In particular, we make the following assumptions.

1. Integers will be represented in decimal.
2. Graphs of n vertices will have their vertices represented by the integers $1, 2, \ldots, n$, encoded in decimal as in assumption 1. An edge is represented by the string (i_1, i_2), where i_1 and i_2 are the decimal representations for the vertex pair indicating the edge.
3. Boolean expressions with n propositional variables will be represented by strings in which $*$ represents "and," $+$ represents "or," $\neg$ represents "not,"† and the integers $1, 2, \ldots, n$ represent the propositional variables. The $*$ is omitted when possible. Parentheses will be used if needed.

We may then say that a problem is in $\mathscr{P}$ or $\mathscr{NP}$ if and only if a standard encoding of the problem is in $\mathscr{P}$ or $\mathscr{NP}$, respectively.

Example 10.3. Consider the clique problem for undirected graphs. A *k-clique* in a graph G is a *complete subgraph* (every pair of distinct vertices in the subgraph is connected by an edge) of G with k vertices. The *clique problem* is, Given a graph G and an integer k, does G contain a k-clique?

The instance of the clique problem with the graph G of Fig. 10.5 and $k = 3$ could be encoded by the string:

$$3(1, 2)(1, 4)(2, 3)(2, 4)(3, 4)(3, 5)(4, 5).$$

The first integer represents the value of k. Then follow those pairs of vertices connected by edges, with v_i represented by i. That is, the edges, in the order listed, are $(v_1, v_2), (v_1, v_4), \ldots, (v_4, v_5)$.

The language L representing the clique problem is the set of strings of the form

$$k(i_1, j_1)(i_2, j_2) \cdots (i_m, j_m)$$

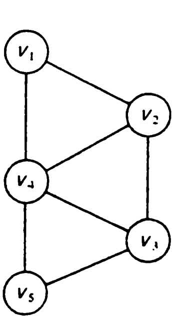


Fig. 10.5. An undirected graph.

† We shall frequently use $\bar{\alpha}$ as an abbreviation for $\neg(\alpha)$. If α consists of a single *literal* (variable or complemented variable), then the parentheses can be omitted.

such that the graph with edges (i_r, j_r) for $1 \le r \le m$ has a k-clique. Other languages could also represent the clique problem. For example, the constant k could be required to follow rather than precede the graph. Or we could use binary integers instead of decimal. However, for any two such languages, there should exist a polynomial p such that a string w representing an instance of the clique problem in one language can be transformed into the string representing the same instance of the problem in the other language in time $p(|w|)$. Thus, as far as membership in $\mathcal{P}$ or $\mathcal{NP}$ is concerned, the exact choice of language to represent the clique problem is unimportant, as long as a "standard" encoding is used. □

The clique problem is in $\mathcal{NP}$. A nondeterministic Turing machine can first "guess" which k vertices will be in the complete subgraph and then verify that there is an edge between each pair of these vertices in $O(n^3)$ steps, where n is the length of this encoding of the clique problem. The "power" of nondeterminism is illustrated here, since all subsets of k vertices are checked "in parallel" by independent copies of the device. For the graph in Fig. 10.5, there are three 3-cliques, namely $\{v_1, v_2, v_4\}$, $\{v_2, v_3, v_4\}$, and $\{v_3, v_4, v_5\}$.

We shall see later that the clique problem is NP-complete. At the present time there is no known way to solve the clique problem in deterministic polynomial time.

Example 10.4. The Boolean expression $(p_1 + p_2) * p_3$ can be represented by the string $(1 + 2)3$, where integer i represents variable p_i. Consider the language L consisting of all strings representing *satisfiable* Boolean expressions (those for which some assignment of 0's and 1's to the variables gives the expression the value 1). We claim that L is in $\mathcal{NP}$. A nondeterministic algorithm to accept L begins by "guessing" a satisfying assignment of 0's and 1's to the propositional variables in an input string, if such an assignment exists. Then, the value (0 or 1) of each variable is substituted for the variable wherever it occurs in the input string. Some shifting of the string will be needed to close up gaps when the single symbol 0 or 1 is substituted for a decimal representation of a propositional variable. Then the resulting expression is evaluated to verify that it has the value 1.

The evaluation can be done in time proportional to its length by a number of parsing algorithms (see Aho and Ullman [1972]). Even without using an efficient parsing algorithm, the reader should have little trouble evaluating a proposition in $O(n^2)$ steps. Hence there is a nondeterministic Turing machine of polynomial time complexity to accept satisfiable Boolean expressions, and thus the problem of determining whether a Boolean expression is satisfiable is in $\mathcal{NP}$. Note again how nondeterminism has been used to "parallelize" the problem, since "guessing" a correct solution really means trying out all possible solutions in parallel. This problem will also be shown to be NP-complete. □

Often we are interested in optimization problems, such as finding a largest clique in a graph. Many such problems can be transformed in polynomial time to language recognition problems. For example, a maximal clique in a graph G can be found as follows. Let n be the length of the representation of G. For each k from 1 to n we determine whether there is a clique of size k. Once the size m of a maximal clique is determined, we delete the vertices one by one until removing a vertex v destroys all remaining cliques of size m. Then consider the subgraph G' consisting of all vertices in G adjacent to v and call the process recursively to find a clique C of size $m - 1$ in G'. A maximal clique for G is C plus the vertex v.

We leave the reader to convince himself that the time to find a maximal clique by the above method is a polynomial function of both n, the length of the representation of G, and the time to determine whether there is a clique of size k in G.

An optimization problem of the form "find the largest k such that $P(k)$ is true" for some proposition P, where the number of possible k's is exponential in the length n of the problem description, can often be solved by binary search (Section 4.3). If $P(k)$ is true implies $P(i)$ is true for $i < k$ and the range of k is between 0 and c^n for some constant c, then the largest k such that $P(k)$ is true can be found by binary search in $\log c^n = n \log c$ tests of propositions $P(k)$. Again the reader should convince himself that the optimum value of k can be found in an amount of time that is a polynomial in n and the maximum time to decide $P(k)$.

We leave the development of techniques of this nature to the ingenuity of the reader and henceforth deal only with yes–no problems.

10.4 NP-COMPLETENESS OF THE SATISFIABILITY PROBLEM

We can show that a problem, or more accurately, its language representation L_0, is NP-complete by showing that L_0 is in $\mathcal{NP}$ and that every language in $\mathcal{NP}$ is polynomially transformable to L_0. Because of the "power" of nondeterminism, the easy part of such an NP-completeness proof is usually in showing the given problem to be in $\mathcal{NP}$. Examples 10.3 and 10.4 are typical of this step. The greatest difficulty is in showing that every problem in $\mathcal{NP}$ is polynomially transformable to the given problem. However, once we have proved a problem L_0 to be NP-complete, a new problem L can be proven NP-complete by showing that L is in $\mathcal{NP}$ and that L_0 is polynomially transformable to L.

We shall show that the satisfiability problem for Boolean expressions is NP-complete. Then we shall show that some fundamental problems in propositional calculus and graph theory are NP-complete by showing that they are in $\mathcal{NP}$ and that the satisfiability problem (or some other already proven NP-complete problem) is polynomially transformable to them.

The following definitions are needed to discuss some important NP-complete problems on undirected graphs.

Definition. Let $G = (V, E)$ be an undirected graph.

1. A *vertex cover* of G is a subset $S \subseteq V$ such that each edge of G is incident upon some vertex in S.
2. A *Hamilton circuit* is a cycle of G containing every vertex of V.
3. G is *k-colorable* if there exists an assignment of the integers 1, 2, . . . , k, called "colors," to the vertices of G such that no two adjacent vertices are assigned the same color. *The chromatic number* of G is the smallest integer k such that G is k-colorable.

The following definitions are needed to present some important NP-complete problems on directed graphs.

Definition. Let $G = (V, E)$ be a directed graph.

1. A *feedback vertex set* is a subset $S \subseteq V$ such that every cycle of G contains a vertex in S.
2. A *feedback edge set* is a subset $F \subseteq E$ such that every cycle of G contains an edge in F.
3. A *directed Hamilton circuit* is a cycle containing every vertex of V.

Theorem 10.2. The following problems are in $\mathcal{NP}$.

1. (*Satisfiability.*) Is a Boolean expression satisfiable?
2. (*Clique.*) Does an undirected graph have a clique of size k?
3. (*Vertex cover.*) Does an undirected graph have a vertex cover of size k?
4. (*Hamilton circuit.*) Does an undirected graph have a Hamilton circuit?
5. (*Colorability.*) Is an undirected graph k-colorable?
6. (*Feedback vertex set.*) Does a directed graph have a feedback vertex set with k members?
7. (*Feedback edge set.*) Does a directed graph have a feedback edge set with k members?
8. (*Directed Hamilton circuit.*) Does a directed graph have a directed Hamilton circuit?
9. (*Set cover.*) Given a family of sets $S_1, S_2, \ldots, S_n$† does there exist a subfamily of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that

$$\bigcup_{j=1}^{k} S_{i_j} = \bigcup_{j=1}^{n} S_j?$$

† We encode finite sets by strings of the form $\{i_1, i_2, \ldots, i_m\}$, where the i's are decimal (or binary) integers representing the set elements. If there are n elements among all the sets. we represent the jth element by integer j.

10. (*Exact cover.*) Given a family of sets $S_1, S_2, \ldots, S_n$ does there exist a set cover consisting of a subfamily of pairwise disjoint sets?

Proof. We showed that problems 1 and 2 are in $\mathcal{NP}$ in Examples 10.4 and 10.3, respectively. Similarly, each of the other parts requires the design of a nondeterministic polynomially-time-bounded Turing machine (or nondeterministic RAM program, if the reader prefers) which "guesses" a solution and checks that it is indeed a solution. We leave the details for exercises. □

We shall now prove that every language in $\mathcal{NP}$ is polynomially transformable to the satisfiability problem, thereby establishing that satisfiability of Boolean expressions is an NP-complete problem.

Theorem 10.3. The problem of determining whether a Boolean expression is satisfiable is NP-complete.

Proof. We already know the satisfiability problem is in $\mathcal{NP}$. Thus we need only show that every language L in $\mathcal{NP}$ is polynomially transformable to the satisfiability problem. Let M be a nondeterministic Turing machine of polynomial-time complexity that accepts L, and let w be an input to M. From M and w we can construct a Boolean expression w_0 such that w_0 is satisfiable if and only if M accepts w. The crux of the proof is in showing that for each M there is a polynomial-time-bounded algorithm to construct the Boolean expression w_0 from w. The polynomial depends on the machine M.

Every language in $\mathcal{NP}$ is accepted by a nondeterministic single-tape Turing machine of polynomial-time complexity, according to Lemma 10.1. Thus we may assume M has but a single tape. Suppose M has states $q_1, q_2, \ldots, q_s$ and tape symbols $X_1, X_2, \ldots, X_m$. Let $p(n)$ be the time complexity of M.

Suppose an input w to M is of length n. If M accepts w, then it does so within $p(n)$ moves. Thus if M accepts w, there is at least one sequence of ID's $Q_0, Q_1, \ldots, Q_q$ such that Q_0 is an initial ID, $Q_{i-1} \vdash Q_i$ for $1 \leq i \leq q$, Q_q is an accepting ID, $q \leq p(n)$, and no ID has more than $p(n)$ tape cells.

We shall construct a Boolean expression w_0 that "simulates" a sequence of ID's entered by M. Each assignment of true (represented by 1) and false (represented by 0) to the variables of w_0 represents at most one sequence of ID's of M, possibly not a legal sequence. The Boolean expression w_0 will take on the value 1 if and only if the assignment to the variables represents a sequence $Q_0, Q_1, \ldots, Q_q$ of ID's leading to acceptance. That is, w_0 will be satisfiable if and only if M accepts w. The following variables are used as the propositional variables in w_0. We give them with their intended interpretation.

1. $C\langle i, j, t\rangle$ is 1 if and only if the ith cell on M's input tape contains the tape symbol X_j at time t. Here $1 \leq i \leq p(n)$, $1 \leq j \leq m$, and $0 \leq t \leq p(n)$.
2. $S\langle k, t\rangle$ is 1 if and only if M is in state q_k at time t. Here $1 \leq k \leq s$ and $0 \leq t \leq p(n)$.

3. $H\langle i, t\rangle$ is true if and only if at time t the tape head is scanning tape cell i. Here $1 \le i \le p(n)$ and $0 \le t \le p(n)$.

There are thus $O(p^2(n))$ propositional variables, and these can be represented by binary numbers with at most c log n bits for some constant c depending on p. It is convenient in what follows to maintain the fiction that each propositional variable can be represented by a single symbol, rather than c log n symbols. This loss of a factor of c log n cannot affect the question of whether any function is polynomial-time-bounded.

From these propositional variables we shall construct a Boolean expression w_0 patterned after the sequence of ID's $Q_0, Q_1, \ldots, Q_q$. In the construction we shall make use of the predicate $U(x_1, x_2, \ldots, x_r)$ that has the value 1 when exactly one of the arguments $x_1, x_2, \ldots, x_r$ has value 1. U can be expressed as a Boolean expression of the form:

$$U(x_1, x_2, \ldots, x_r) = (x_1 + x_2 + \cdots + x_r) \left(\prod_{\substack{i,j \\ i \neq j}} (\neg x_i + \neg x_j) \right). \quad (10.1)$$

The first factor of (10.1) asserts that at least one of the x_i's is true. The remaining $r(r-1)/2$ factors assert that no two x_i's are true. Note that when written out formally, $U(x_1, \ldots, x_r)$ is of length $O(r^2)$.†

If M accepts w, then there is an accepting sequence of ID's $Q_0, Q_1, \ldots, Q_q$ entered by M in processing w. To simplify the discussion we can assume without loss of generality that M has been modified so that whenever it reaches the accepting state, it continues "running" without moving its head or leaving the accepting state and that each ID in the sequence is padded out with blanks to length $p(n)$. Therefore, we shall construct w_0 as the product of seven expressions $A, B, \ldots, G$ to assert that $Q_0, Q_1, \ldots, Q_q$ is an accepting sequence of ID's, where each Q_i is of length $p(n)$ and $q = p(n)$. Asserting that $Q_0, Q_1, \ldots, Q_{p(n)}$ is an accepting sequence of ID's then is tantamount to asserting that:

1. the tape head is scanning exactly one cell in each ID,
2. each ID has exactly one tape symbol in each tape cell,
3. each ID has exactly one state,
4. at most one tape cell, the cell scanned by the tape head, is modified from one ID to the next,
5. the change in state, head location, and tape cell contents between successive ID's is allowed by the move function of M,
6. the first ID is an initial ID, and
7. the state in the last ID is the final state.

† Recall that we are counting one symbol per variable. Strictly speaking $O(r^2 \log r)$, hence at most $O(r^3)$, symbols would be required.

We now construct Boolean expressions A–G to mirror statements 1 through 7 above.

1. A asserts that at each unit of time M is scanning exactly one cell. Let A_t assert that at time t exactly one cell is scanned. Then $A = A_0 A_1 \cdots A_{p(n)}$, where

$$A_t = U(H\langle 1, t\rangle, H\langle 2, t\rangle, \ldots, H\langle p(n), t\rangle).$$

Note that when the shorthand U is expanded, A is of length $O(p^3(n))$ and can be written down in that time.

2. B asserts that each tape cell contains exactly one symbol at each unit of time. Let B_{it} assert that the ith tape cell contains exactly one symbol at time t. Then

$$\mathrm{B} = \prod_{i,t} B_{it},$$

where

$$B_{it} = U(C\langle i, 1, t\rangle, C\langle i, 2, t\rangle, \ldots, C\langle i, m, t\rangle).$$

The length of each B_{it} is independent of n, since m, the size of the tape alphabet, depends only on the Turing machine M. Thus B is of length $O(p^2(n))$.

3. C asserts that at each time t, M is in one and only one state:

$$C = \prod_{0 \leq t \leq p(n)} U(S\langle 1, t\rangle, S\langle 2, t\rangle, \ldots, S\langle s, t\rangle).$$

Since s, the number of states of M, is a constant, C is of length $O(p(n))$.

4. D asserts that the contents of at most one tape cell can be changed at any time t:

$$D = \prod_{i,j,t} [(C\langle i, j, t\rangle \equiv^{\dagger} C\langle i, j, t+1\rangle) + H\langle i, t\rangle].$$

The expression $(C\langle i, j, t\rangle \equiv C\langle i, j, t+1\rangle) + H\langle i, t\rangle$ asserts that either

a) the tape head is scanning cell i at time t or

b) the jth symbol is in cell i at time $t+1$ if and only if it was there at time t.

Since A and B assert that the tape head is scanning only one cell at time t and that cell i contains only one symbol, either the tape head is scanning cell i at time t or the contents of cell i do not change at time t. When $\equiv$ is replaced by its denoted expression, D is of length $O(p^2(n))$.

5. E asserts that each successive ID of M is obtained from the previous ID by one transition allowed by the next-move function δ of M. Let E_{ijkt}

† We use $x \equiv y$ to stand for $xy + \bar{x}\bar{y}$, that is, x if and only if y.

assert one of the following:

a) that the ith cell does not contain symbol j at time t,
b) that the tape head is not scanning cell i at time t,
c) that M is not in state k at time t, or
d) that the next ID of M is obtained from the previous ID by a transition allowed by the move function of M.

Then

$$E = \prod_{i,j,k,t} E_{ijkt},$$

where

$$E_{ijkt} = \neg C\langle i, j, t\rangle + \neg H\langle i, t\rangle + \neg S\langle k, t\rangle + \sum_{l} [C\langle i, j_l, t+1\rangle S\langle k_l, t+1\rangle H\langle i_l, t+1\rangle].$$

Here l ranges over all possible moves of M when scanning symbol X_j in state q_k. That is, for each triple (q, X, d) in $\delta(q_k, X_j)$, there is one value of l for which $X_{j_l} = X$, $q_{k_l} = q$, and i_l is $i-1$, i, or $i+1$ as d is L, S, or R, respectively. Here, δ is the move function of M. Since M is nondeterministic, there may be more than one such triple (q, X, d), but a finite number in any case. Thus E_{ijkt} is of bounded length independent of n. Therefore, E is of length $O(p^2(n))$.

6. F asserts that the initial conditions are satisfied:

$$F = S\langle 1, 0\rangle H\langle 1, 0\rangle \prod_{1\le i\le n} C\langle i, j_i, 0\rangle \prod_{n<i\le p(n)} C\langle i, 1, 0\rangle,$$

where $S\langle 1, 0\rangle$ asserts that at time $t=0$, M is in state q_1, which we take to be the initial state; $H\langle 1, 0\rangle$ asserts that at time $t=0$, M is scanning the leftmost tape cell; $\Pi_{1\le i\le n} C\langle i, j_i, 0\rangle$ asserts that the first n tape cells initially contain the input string w; and $\Pi_{n<i\le p(n)} C\langle i, 1, 0\rangle$ asserts that the remaining tape cells are initially blank. We take X_1 to be the symbol for the blank. Clearly, F is of length $O(p(n))$.

7. G asserts that M eventually enters the final state. Since we have required that M remain in the final state once it has reached it, we have $G = S\langle s, p(n)\rangle$. We take q_s to be the final state.

The Boolean expression w_0 is the product $ABCDEFG$. Since each of the seven factors of w_0 requires at most $O(p^3(n))$ symbols, w_0 itself has $O(p^3(n))$ symbols. Since we have been counting propositional variables as single symbols, when in fact they are represented by strings of length $O(\log n)$, we really have a bound on $|w_0|$ of $O(p^3(n) \log n)$, which is in turn bounded by $cnp^3(n)$ for some constant c. Thus the length of w_0 is a polynomial function of the length of w. It should be clear that w_0 can be written down in time proportional to its length, given w and the polynomial p.

Given an accepting sequence of ID's $Q_0, Q_1, \ldots, Q_q$ we can obviously find an assignment of 0's and 1's to the propositional variables $C\langle i, j, t\rangle$, $S\langle k, t\rangle$, and $H\langle i, t\rangle$ that will make w_0 have the value **true**. Conversely, given an assignment of values to the variables of w_0 that makes w_0 true, we can easily find an accepting sequence of ID's. Thus w_0 is satisfiable if and only if M accepts w.

We put no restrictions on L, the language accepted by M, other than that L be in $\mathcal{NP}$. We have therefore shown that any language in $\mathcal{NP}$ is polynomially transformable to the satisfiability problem for Boolean expressions. We conclude that the satisfiability problem is NP-complete. □

We have in fact shown more in Theorem 10.3 than is stated. A Boolean expression is said to be in *conjunctive normal form* (CNF) if it is the product of sums of literals, where a *literal* is either x or $\neg x$ for some variable x. For example, $(x_1 + x_2)(x_2 + \bar{x}_1 + x_3)$ is in CNF; $x_1x_2 + x_3$ is not. The expression w_0 constructed in Theorem 10.3 is practically in CNF, and we can put it in CNF without expanding its length by more than a constant factor.

Corollary. The satisfiability problem for Boolean expressions in CNF is NP-complete.

Proof. It suffices to show that each of the expressions $A, \ldots, G$ defined in the proof of Theorem 10.3 is either already in CNF or can be manipulated into such an expression, by use of laws of Boolean algebra, without increasing the length of the expression by more than a constant factor. We observe that U, defined in Eq. (10.1), is already in CNF. It follows that A, B, and C are in CNF. F and G are trivially in CNF, since they are products of single literals.

D is the product of expressions of the form $(x \equiv y) + z$. If we replace the $\equiv$ sign we obtain the expression

$$xy + \bar{x}\bar{y} + z. \tag{10.2}$$

We observe that (10.2) is equivalent to

$$(x + \bar{y} + z)(\bar{x} + y + z). \tag{10.3}$$

By substituting (10.3) for (10.2) wherever it appears in D, we can obtain an equivalent expression that is in CNF and at most twice as long as the original.

Finally, the expression E is the product of the expressions E_{ijkt}. Since the length of each E_{ijkt} is bounded independently of n, each E_{ijkt} can be expressed by a conjunctive normal form expression whose length is independent of n. Thus converting E to CNF increases the length of E by at most a constant factor.

We have thus shown that the expression w_0 can be put in CNF with but a constant-factor increase in its length. □

We have just shown that the satisfiability problem for CNF expressions is NP-complete. We can show that even with a more stringent condition, the satisfiability problem for Boolean expressions is NP-complete. An expression is said to be in *k-conjunctive normal form* (k-CNF) if it is the product of sums of at most k literals. The *k-satisfiability problem* is to determine whether an expression in k-CNF is satisfiable. For $k = 1$ or 2 we can find deterministic polynomial algorithms to test for k-satisfiability. The situation (presumably) changes at $k = 3$, as seen in the next theorem.

Theorem 10.4. 3-satisfiability is NP-complete.

Proof. We shall show that satisfiability for CNF expressions is polynomially transformable to 3-satisfiability. Given a product of sums, replace each sum $(x_1 + x_2 + \cdots + x_k)$, $k \geq 4$, with

$$(x_1 + x_2 + y_1)(x_3 + \bar{y}_1 + y_2)(x_4 + \bar{y}_2 + y_3) \cdots (x_{k-2} + \bar{y}_{k-4} + y_{k-3})(x_{k-1} + x_k + \bar{y}_{k-3}) \quad (10.4)$$

for new variables $y_1, y_2, \ldots, y_{k-3}$. For example, for $k = 4$, expression (10.4) is $(x_1 + x_2 + y_1)(x_3 + x_4 + \bar{y}_1)$.

There is some assignment to the new variables which makes the replacing expression have value 1 if and only if one of the literals $x_1, x_2, \ldots, x_k$ has value 1, that is, if and only if the original expression has value 1. Assume $x_i = 1$. Then set y_j to 1 for $j \leq i - 2$ and set y_j to 0 for $j > i - 2$. The replacing expression has value 1. Conversely, assume that some assignment for the y_i's makes the resulting expression have value 1. If $y_1 = 0$, then either x_1 or x_2 must have value 1. If $y_{k-3} = 1$, then either x_{k-1} or x_k must have value 1. If $y_1 = 1$ and $y_{k-3} = 0$, then for some i, $1 \leq i \leq k - 4$, $y_i = 1$ and $y_{i+1} = 0$, implying x_{i+2} must have value 1. In any event some x_i must have value 1.

The length of each replacing expression is bounded by a constant multiple of the length of the replaced expression. In fact, given any CNF expression E, we can find, by applying the above transformations to each sum, a 3-CNF expression E' which is satisfiable if and only if the original expression is satisfiable. Moreover, we can find E' in time proportional to the length of E, since the transformations are straightforward to apply. □

10.5 ADDITIONAL NP-COMPLETE PROBLEMS

We now proceed to show that each of the problems mentioned in Theorem 10.2 is NP-complete by directly or indirectly transforming satisfiability to them. The tree in Fig. 10.6 shows the sequence of transformations we shall actually make. If P is the father of P' in Fig. 10.6, then we show that P is polynomially transformable to P'.

Theorem 10.5. CNF-satisfiability is polynomially transformable to the clique problem. Therefore, the clique problem is NP-complete.

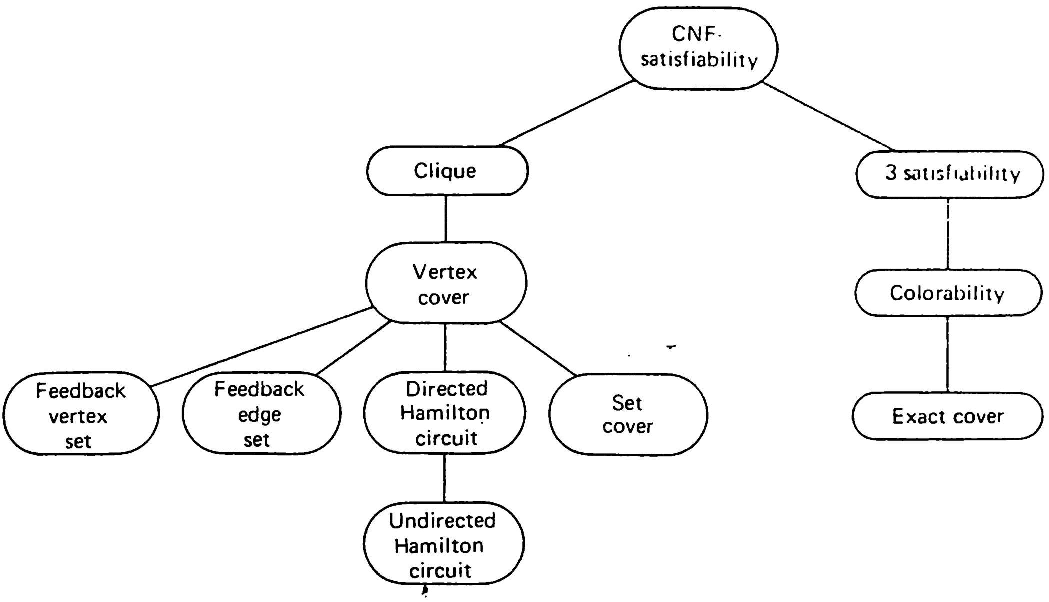


Fig. 10.6 Order of transformations for various hard problems.

Proof. Let $F = F_1F_2 \cdots F_q$ be an expression in CNF, where the F_i's are the factors: each F_i is of the form $(x_{i1} + x_{i2} + \cdots + x_{ik_i})$, where x_{ij} is a literal. We shall construct an undirected graph $G = (V, E)$ whose vertices are pairs of integers $[i, j]$, for $1 \leq i \leq q$ and $1 \leq j \leq k_i$. The first component of the pair represents a factor, and the second a literal within the factor. Thus each vertex of the graph corresponds to a particular literal of a particular factor in a natural way.

The edges of G are those pairs $([i, j], [k, l])$ such that $i \neq k$ and $x_{ij} \neq \bar{x}_{kl}$. Intuitively, $[i, j]$ and $[k, l]$ are adjacent in G if they correspond to different factors and it is possible to assign values to the variables in literals x_{ij} and x_{kl} in such a way that both literals have value 1 (giving F_i and F_k the value 1). That is, either $x_{ij} = x_{kl}$, or x_{ij} and x_{kl} are complemented or uncomplemented versions of different variables.

The number of vertices in G is clearly less than the length of F, and the number of edges is at most the square of this number. Thus G can be encoded as a string whose length is bounded by a polynomial in the length of F. More importantly, such an encoding can be computed in time bounded by a polynomial in the length of F. We shall show that G has a clique of size q if and only if F is satisfiable. It then follows that given a polynomial-time-bounded algorithm for the clique problem we could construct a polynomial-time algorithm for the CNF-satisfiability problem by converting an expression F to a graph G such that G has a clique of size q if and only if F is satisfiable. The proof that G has a clique of size q if and only if F is satisfiable follows.

IF. Assume that F is satisfiable. Then there exists a 0–1 assignment of values to the variables such that $F = 1$. For this assignment each factor of F has value 1. Each factor F_i contains at least one literal with value 1. Let one such literal in F_i be x_{im_i}.

We claim that the set of vertices $\{[i, m_i] | 1 \leq i \leq q\}$ forms a clique of size q. Otherwise there exist i and j, $i \neq j$, such that there is no edge between vertices $[i, m_i]$ and $[j, m_j]$. This implies that $x_{im_i} = \bar{x}_{jm_j}$ by definition of the edge set of G. But this is impossible since $x_{im_i} = x_{jm_j} = 1$ by the way the x_{im_i}'s were selected.

ONLY IF. Assume G has a clique of size q. Each vertex in the clique must have a distinct first component, since two vertices with the same first component are not connected by an edge. Since there are exactly q vertices in the clique, there is a one-to-one correspondence between the vertices of the clique and the factors of F. Let the vertices of the clique be $[i, m_i]$ for $1 \leq i \leq q$. Let $S_1 = \{y | x_{im_i} = y$, where $1 \leq i \leq q$ and y is a variable$\}$, and let $S_2 = \{y | x_{im_i} = \bar{y}$, where $1 \leq i \leq q$ and y is a variable$\}$. That is, S_1 and S_2 denote the sets of uncomplemented and complemented variables, respectively, represented by the vertices of the clique. Then $S_1 \cap S_2 = \emptyset$, otherwise there would be an edge between some $[s, m_s]$ and $[t, m_t]$, where $x_{sm_s} = \bar{x}_{tm_t}$. By setting the variables of S_1 to 1 and those of S_2 to 0, the value of each F_i is made 1. Thus F is satisfiable. □

Example 10.5. Consider the expression $F = (y_1 + \bar{y}_2)(y_2 + \bar{y}_3)(y_3 + \bar{y}_1)$. The literals are

$$\begin{array}{lll} x_{11} = y_1, & x_{21} = y_2, & x_{31} = y_3, \\ x_{12} = \bar{y}_2, & x_{22} = \bar{y}_3, & x_{32} = \bar{y}_1. \end{array}$$

The construction of Theorem 10.5 yields the graph of Fig. 10.7. For example, $[1, 1]$ is not connected to $[1, 2]$ because the first components are the same, and $[1, 1]$ is not connected to $[3, 2]$ since $x_{11} = y_1$ and $x_{32} = \bar{y}_1$. $[1, 1]$ is connected to the other three vertices.

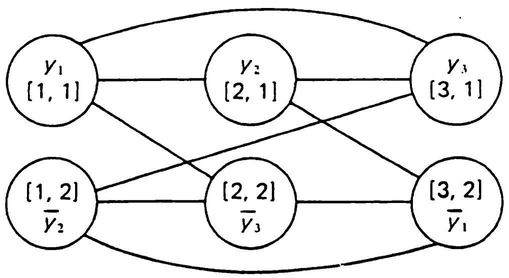


Fig. 10.7 Graph constructed by Theorem 10.5.

F has three factors, and it happens that there are two cliques of size 3 in the graph of Fig. 10.7, namely $\{[1, 1], [2, 1], [3, 1]\}$ and $\{[1, 2], [2, 2], [3, 2]\}$. In the first case, literals y_1, y_2, and y_3 are represented. These are all uncomplemented variables, and the clique corresponds to the assignment $y_1 = y_2 = y_3 = 1$ to F. The other clique corresponds to the other assignment that makes F true, namely, $y_1 = y_2 = y_3 = 0$. □

Theorem 10.6. The clique problem is polynomially transformable to the vertex cover problem. Therefore, the vertex cover problem is NP-complete.

Proof. Given an undirected graph $G = (V, E)$, consider the complement graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(v, w) \mid v, w \in V, v \neq w, \text{ and } (v, w) \notin E\}$. We claim a set $S \subseteq V$ is a clique in G if and only if $V - S$ is a vertex cover of $\overline{G}$. For if S is a clique in G, no edge in $\overline{G}$ connects two vertices in S. Thus every edge in $\overline{G}$ is incident upon at least one vertex in $V - S$, implying $V - S$ is a vertex cover of $\overline{G}$. Similarly, if $V - S$ is a vertex cover of $\overline{G}$, then every edge of $\overline{G}$ is incident upon at least one vertex of $V - S$. Thus no edge of $\overline{G}$ connects two vertices in S. Therefore every pair of vertices of S is connected in G, and S is a clique in G.

To find whether there exists a clique of size k, we construct $\overline{G}$ and determine whether it has a vertex cover of size $\|V\| - k$. Surely, given a standard representation of $G = (V, E)$ and k, we can find a representation of $\overline{G}$ and $\|V\| - k$ in time which is a polynomial in the length of the representation for G and k. □

Example 10.6. Graph G of Fig. 10.8(a) has cliques $\{1, 2, 5\}$ and $\{1, 4, 5\}$ of size 3. $\overline{G}$ in Fig. 10.8(b) has the corresponding vertex covers $\{3, 4\}$ and $\{2, 3\}$ of size 2. G has cliques $\{2, 3\}$ and $\{3, 4\}$ of size 2 (among others). $\overline{G}$ has the corresponding vertex covers $\{1, 4, 5\}$ and $\{1, 2, 5\}$ of size 3. □

Theorem 10.7. The vertex cover problem is polynomially transformable to the feedback vertex set problem. Therefore, the feedback vertex set problem is NP-complete.

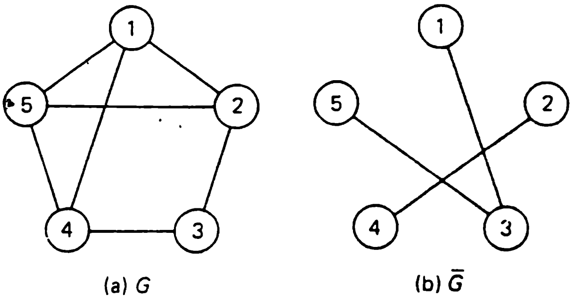


Fig. 10.8 (a) A graph and (b) its complement.

Proof. Let $G = (V, E)$ be an undirected graph. Let D be the directed grapt formed by replacing each edge of G by two directed edges. Specifically, le: $D = (V, E')$, where $E' = \{(v, w), (w, v) | (v, w) \in E\}$. Since every edge in E has been replaced by a cycle in D, a set $S \subseteq V$ is a feedback vertex set foi D (every cycle of D contains a vertex in S) if and only if S is a vertex covei for G. Also, the representation of D can easily be found from G in polynomial time. □

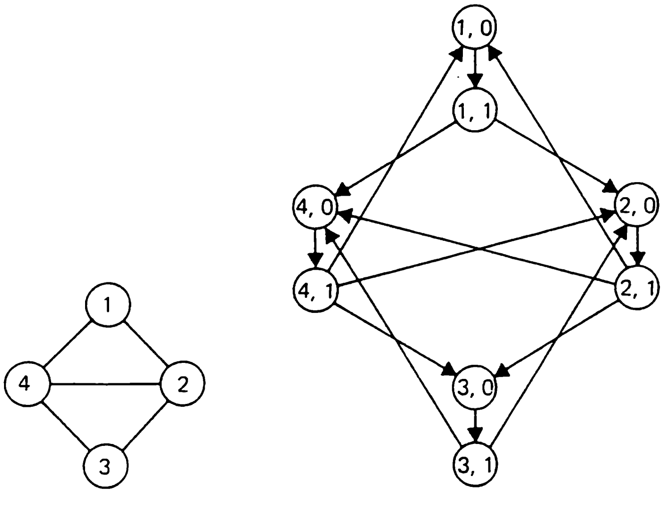
Theorem 10.8. The vertex cover problem is polynomially transformable to the feedback edge set problem. Therefore, the feedback edge set problem is NP-complete.

Proof. Let $G = (V, E)$ be an undirected graph. Let $D = (V \times \{0, 1\}, E')$ be a directed graph, where E' consists of:

i) $[v, 0] \to [v, 1]$† for each $v \in V$, and

ii) $[v, 1] \to [w, 0]$ and $[w, 1] \to [v, 0]$ for each undirected edge $(v, w) \in E$. See Fig. 10.9.

Let $F \subseteq E'$ be a set of edges of D containing at least one edge from each cycle in D. Replace each edge in F of the form $[v, 1] \to [w, 0]$ by the edge



(a) G (b) D

Fig. 10.9 An undirected graph (a) and the corresponding directed graph (b) used in Theorem 10.8. Vertex cover $\{2, 4\}$ corresponds to feedback edge set $\{(2,0) \to (2,1), (4,0) \to (4,1)\}$.

† We use $x \to y$ for the directed edge (x, y) here and subsequently in this chapter. The convention improves readability.

$[w, 0] \rightarrow [w, 1]$. Let the resulting set be F'. Then we claim $\|F'\| \leq \|F\|$, and F' contains at least one edge from each cycle. (The only edge out of $[w, 0]$ goes to $[w, 1]$, so $[w, 0] \rightarrow [w, 1]$ is in any cycle containing $[v, 1] \rightarrow [v, 0]$.)

Without loss of generality, assume

$$F' = \{[v_i, 0] \rightarrow [v_i, 1] \mid 1 \leq i \leq k\}$$

for some k. Then every cycle of D contains an edge $[v_i, 0] \rightarrow [v_i, 1]$ for some i, $1 \leq i \leq k$. However, note that if (x, y) is any edge of G, then $[x, 1]$, $[y, 0]$, $[y, 1]$, $[x, 0]$, $[x, 1]$ is a cycle in D. Thus every edge of G is incident upon some v_i, $1 \leq i \leq k$, and hence $\{v_1, \ldots, v_k\}$ is a vertex cover for G.

Conversely, we may easily show that given a vertex cover S of size k, the set $\{[v, 0] \rightarrow [v, 1] \mid v \in S\}$ is a feedback edge set of D. To find whether there is a vertex cover for G with k vertices, construct D in polynomial time and determine whether there is a feedback edge set of D with k members. □

Theorem 10.9. The vertex cover problem is polynomially transformable to the Hamilton circuit problem for directed graphs. Hence the Hamilton circuit problem for directed graphs is NP-complete.

Proof. Given an undirected graph $G = (V, E)$ and an integer k we show how to construct in time polynomial in $\|V\|$ a directed graph $G_D = (V_D, E_D)$ such that G_D has a Hamilton circuit if and only if a set of k vertices of V covers the edges of G.

Let $V = \{v_1, v_2, \ldots, v_n\}$. Denote an edge (v_i, v_j) by e_{ij}.† Let $a_1, a_2, \ldots, a_k$ be new symbols. The vertices of G_D will consist of one vertex corresponding to each a_i plus four vertices for each edge of G. More precisely:

$$V_D = \{a_1, a_2, \ldots, a_k\} \cup \{[v, e, b] \mid v \in V,\ e \in E,\ b \in \{0, 1\}, \text{ and } e \text{ is incident upon } v\}.$$

Before formally describing the edges of G_D we give an intuitive explanation. An edge (v_i, v_j) of G is represented by a subgraph with four vertices as shown in Fig. 10.10.

If a Hamilton circuit enters the subgraph representing edge e_{ij} at A, then it must leave by D, since if it leaves by C either $[v_j, e_{ij}, 0]$ or $[v_i, e_{ij}, 1]$ cannot appear on the circuit. In going from A to D, the path may visit all four vertices in the subgraph, or it may visit only the two leftmost. In the latter case the Hamilton circuit must at some point go from B to C visiting the two rightmost vertices.

G_D can be thought of as consisting of $\|V\|$ lists, one list for each vertex. The list for vertex v contains all edges incident upon v in some specific order. Each a_j, $1 \leq j \leq k$, has an edge to the vertices $[v_i, e_{il}, 0]$ such that e_{il} is the first edge on the list for v_i. There is an edge from $[v_i, e_{il}, 1]$ to each a_j if e_{il}

† Note that e_{ij} and e_{ji} denote the same edge and are to be regarded as the same symbol.

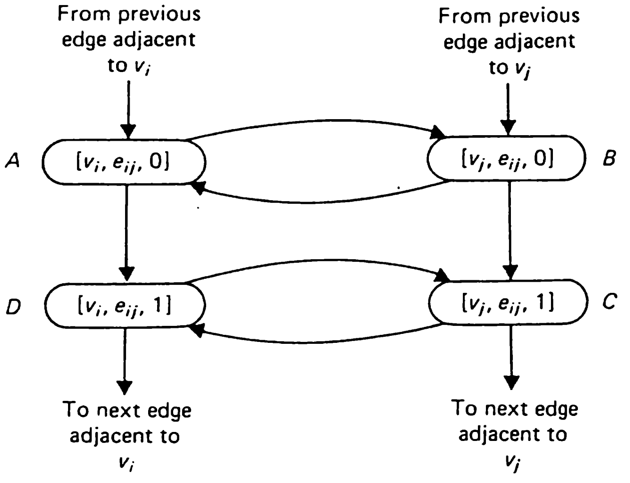

From previous edge adjacent to v_i
From previous edge adjacent to v_j
A
$[v_i, e_{ij}, 0]$
$[v_j, e_{ij}, 0]$
B
D
$[v_i, e_{ij}, 1]$
$[v_j, e_{ij}, 1]$
C
To next edge adjacent to v_i
To next edge adjacent to v_j

Fig. 10.10 Representation of an edge (v_i, v_j).

is the last edge on the list for v_i. There is an edge from $[v_i, e_{il}, 1]$ to $[v_i, e_{im}, 0]$ if edge e_{im} immediately follows e_{il} on the list for v_i. These edges, in addition to the edges implied by Fig. 10.10, form the set E_D. We now show that G_D has a Hamilton circuit if and only if G has a set of k vertices covering all edges.

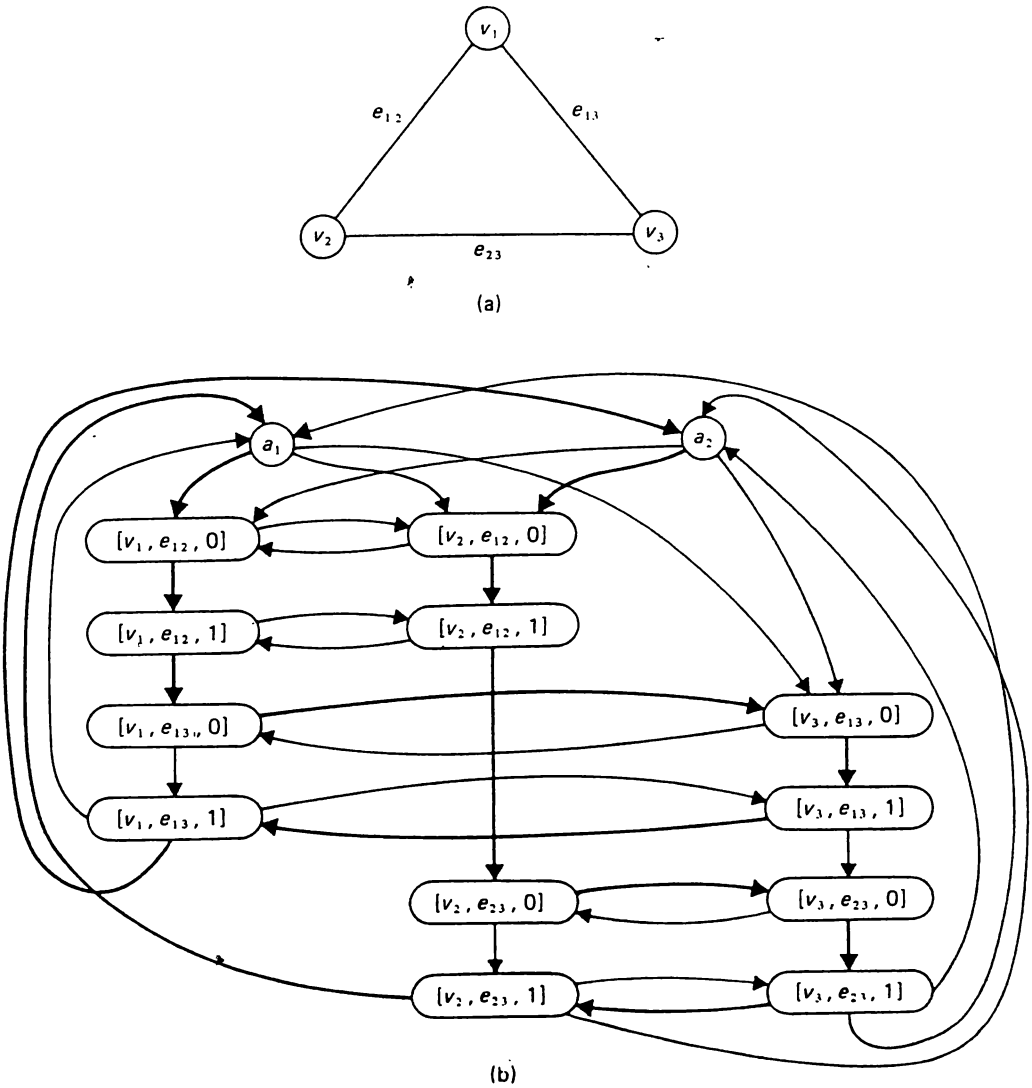
IF. Assume vertices $v_1, v_2, \ldots, v_k$ cover all edges of G. Let $f_{i1}, f_{i2}, \ldots, f_{il_i}$ be the list of edges incident upon v_i, $1 \leq i \leq k$. Consider the cycle from vertex a_1 to $[v_1, f_{11}, 0]$, $[v_1, f_{11}, 1]$, $[v_1, f_{12}, 0]$, $[v_1, f_{12}, 1]$, $\ldots$, $[v_1, f_{1l_1}, 0]$, $[v_1, f_{1l_1}, 1]$ and then to a_2, then to $[v_2, f_{21}, 0]$, $[v_2, f_{21}, 1]$, $\ldots$ and so on through the edge lists of $v_3, v_4, \ldots, v_k$, then finally back to a_1. This cycle goes through every vertex of G_D except those vertices in the edge lists of vertices other than $v_1, \ldots, v_k$. For each pair of vertices of G_D of the form $[v_j, e, 0]$, $[v_j, e, 1]$, $j > k$, we can extend the cycle, since e is incident upon some v_i, $1 \leq i \leq k$. Replace the edge from $[v_i, e, 0]$ to $[v_i, e, 1]$ in the cycle by the path

$$[v_i, e, 0],\ [v_j, e, 0],\ [v_j, e, 1],\ [v_i, e, 1].$$

The cycle, revised as above for each v_j and e, contains every vertex of G_D and hence is a Hamilton circuit.

ONLY IF. Assume G_D has a Hamilton circuit. The circuit can be broken into k paths, each path starting at a vertex a_i and ending at a vertex a_j. By our previous consideration of possible paths through the subgraph of four vertices representing each edge (as in Fig. 10.10), we see that there exists a vertex v such that each vertex on the path from a_i to a_j is of the form $[v, e, 0]$ or $[v, e, 1]$, where e is an edge of G, or is of the form $[w, e, 0]$ or $[w, e, 1]$, where e is the edge (v, w). Thus the first component of each vertex on the path from a_i to a_j is either the vertex v or a vertex adjacent to v. With the path from a_i to a_j we thus may associate some specific vertex v. For every vertex $[w, e, b]$ of G_D, e must be incident upon one of the k vertices of G associated with a path. Thus the k vertices form a vertex cover of G. We conclude that G_D has a Hamilton circuit if and only if there exist k vertices in G such that every edge of G is incident upon one of these k vertices. ☐

Example 10.7. Let G be the graph with vertices v_1, v_2, and v_3 and with edges e_{12}, e_{13}, and e_{23} as in Fig. 10.11(a). The graph G_D constructed in Theorem



(a)

(b)

Fig. 10.11 Graph with Hamilton circuit: (a) the undirected graph G, (b) the directed graph G_D.

10.9 is shown in Fig. 10.11(b). A Hamilton circuit of G_D, based on the vertex cover $\{v_1, v_2\}$, is shown in boldface. □

Theorem 10.10. The Hamilton circuit problem for directed graphs is polynomially transformable to the Hamilton circuit problem for undirected graphs. Hence the problem of determining whether there is a Hamilton circuit in an undirected graph is NP-complete.

Proof. Let $G = (V, E)$ be a directed graph. Let $U = (V \times \{0, 1, 2\}, E')$ be an undirected graph where E' consists of edges

1. $([v, 0], [v, 1])$ for $v \in V$,
2. $([v, 1], [v, 2])$ for $v \in V$, and
3. $([v, 2], [w, 0])$ if and only if $v \rightarrow w$ is an edge in E.

Note that each vertex of V has been expanded into a path consisting of three vertices. Since a Hamilton circuit in U must include all vertices, the last component of the vertices on the circuit must vary in the order 0, 1, 2, 0, 1, 2, . . . or its reversal. We may assume the former, in which case the type-3 edges, whose second component goes from 2 to 0, represent a directed Hamilton circuit in G. Conversely, a Hamilton circuit in G may be converted to an undirected Hamilton circuit in U by replacing each edge $v \rightarrow w$ by the path $[v, 0]$, $[v, 1]$, $[v, 2]$, $[w, 0]$ in U. □

Theorem 10.11. The vertex cover problem is polynomially transformable to the set cover problem. Therefore, the set cover problem is NP-complete.

Proof. Let $G = (V, E)$ be an undirected graph. For $1 \leq i \leq \|V\|$, let S_i be the set of all edges incident upon v_i. Clearly, $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ is a set cover for the S_i's if and only if $\{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ is a vertex cover for G. □

Theorem 10.12. The 3-satisfiability problem is polynomially transformable to the colorability problem.

Proof. Given an expression F in 3-CNF with n variables and t factors, we show how to construct, in time polynomial in $\mathrm{MAX}(n, t)$, an undirected graph $G = (V, E)$ with $3n + t$ vertices, such that G can be colored with $n + 1$ colors if and only if F is satisfiable.

Let $x_1, x_2, \ldots, x_n$ and $F_1, F_2, \ldots, F_t$ be the variables and factors of F, respectively. Let $v_1, v_2, \ldots, v_n$ be new symbols. Without loss of generality assume $n \geq 4$ since any expression in 3-CNF with three or fewer distinct variables can be tested directly for satisfiability in time linear in the length of the expression without use of the transformation to colorability. The vertices of G are:

1. x_i, $\bar{x}_i$, and v_i, for $1 \leq i \leq n$, and
2. F_i, for $1 \leq i \leq t$.

The edges of G are:

1. all (v_i, v_j) such that $i \neq j$,
2. all (v_i, x_j) and $(v_i, \bar{x}_j)$ such that $i \neq j$,
3. $(x_i, \bar{x}_i)$ for $1 \leq i \leq n$,
4. (x_i, F_j) if x_i is not a term of factor F_j, and $(\bar{x}_i, F_j)$, if $\bar{x}_i$ is not a term of F_j.

The vertices $v_1, v_2, \ldots, v_n$ form a complete subgraph of n vertices and hence require n distinct colors. Each x_j and $\bar{x}_j$ is connected to each v_i, $i \neq j$, and hence x_j and $\bar{x}_j$ cannot be the same color as any of the v's, except possibly v_j. Since x_j and $\bar{x}_j$ are adjacent, they cannot be the same color, so G cannot be colored with $n + 1$ colors unless one of x_j and $\bar{x}_j$ is the same color as v_j and the other is a new color which we refer to as the *special color*.

Think of that one of x_j or $\bar{x}_j$ colored with the special color as being assigned the value 0. Now consider the color assigned to the F_j vertices. F_j is adjacent to at least $2n - 3$ of the $2n$ x_i's and $\bar{x}_i$'s. Since we assume $n \geq 4$, for each j there exists an i such that F_j is adjacent to both x_i and $\bar{x}_i$. Since one of x_i or $\bar{x}_i$ is colored with the special color, F_j cannot be colored with the special color.

If F_j contains some literal y, where $\bar{y}$ has been assigned the special color, then F_j is not adjacent to any vertex colored the same as y and hence may be assigned the same color as y. Otherwise, a new color is needed. Thus all the F_j's can be colored with no additional colors if and only if there is an assignment of the special color to the literals such that each factor contains some literal y where $\bar{y}$ has been assigned the special color—that is, if and only if one can assign values to the variables so that each factor contains a y assigned the value 1 ($\bar{y}$ assigned the value 0), i.e., if and only if F is satisfiable. □

Example 10.8. Let F be $(x_1 + x_2)(\bar{x}_1 + x_3)$. Note that there are two rather than three terms per factor. This change does not alter the construction of the graph G in Theorem 10.12, but enables us to deal with expressions involving only three variables and graphs which can be colored with four colors. (Observe that the analog to Theorem 10.12 for 2-CNF is true, but uninteresting. Since there is a polynomial-time-bounded algorithm for 2-CNF, satisfiability for 2-CNF is polynomial-time-transformable to every problem.) The resulting graph is shown in Fig. 10.12. Colors are marked A, B, C, and S (special). The 4-coloring corresponds to the solution $x_1 = \bar{x}_2 = x_3 = 0$. □

Theorem 10.13. The colorability problem is polynomially transformable to the exact cover problem. Hence the exact cover problem is NP-complete.

Proof. Let $G = (V, E)$ be an undirected graph, and let k be an integer. We construct sets whose elements are chosen from

$$S = V \cup \{[e, i] \mid e \in E \text{ and } 1 \leq i \leq k\}.$$

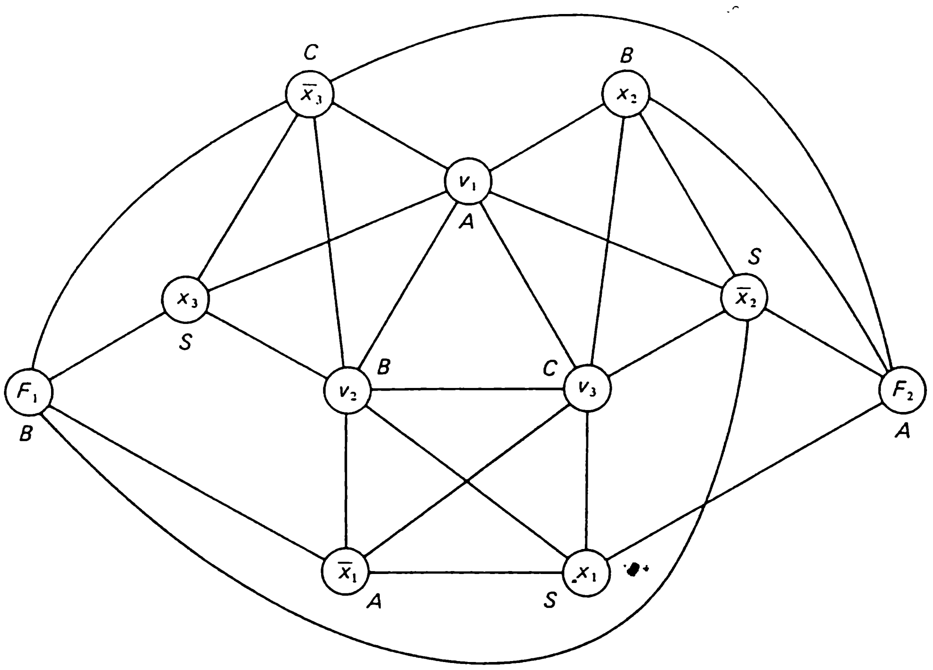


Fig. 10.12 Graph with chromatic number 4.

For each $v \in V$ and $1 \leq i \leq k$, define

$$S_{vi} = \{v\} \cup \{[e, i] | e \text{ incident upon } v\}. \tag{10.5}$$

For each edge e and $1 \leq i \leq k$, define

$$T_{ei} = \{[e, i]\}. \tag{10.6}$$

We can relate k-colorings of the graph G to exact covers for the collection of sets defined by (10.5) and (10.6), as follows. Suppose G has a k-coloring, in which vertex v is colored c_v, where c_v may be taken to be an integer from 1 to k. Then we may easily check that the collection of sets consisting of S_{vc_v} for each v, and those singleton sets T_{ei} such that $[e, i] \notin S_{vc_v}$ for any v, forms an exact cover. In proof, note that if $e = (v, w)$ is an edge, then $c_v \neq c_w$, so $S_{vc_v} \cap S_{wc_w} = \emptyset$. Certainly, if x and y are not adjacent, then S_{xc_x} and S_{yc_y} are disjoint, and no T_{ei} is selected unless it is disjoint from all other selected sets.

Conversely, suppose the sets have an exact cover $\mathcal{S}$. Then for each v, there is a unique c_v such that S_{vc_v} is in $\mathcal{S}$. This follows since each v must be in exactly one set of $\mathcal{S}$. We claim that each vertex v may be colored c_v to obtain a k-coloring of G. Suppose not. Then there is some edge $e = (v, w)$ such that $c_v = c_w = c$. Then S_{vc_v} and S_{wc_w} each contain $[e, c]$, so $\mathcal{S}$ is not an exact cover as supposed. □

10.6 POLYNOMIAL-SPACE-BOUNDED PROBLEMS

There is another natural class of difficult problems which contains $\mathcal{NP}$. This class, which we call $\mathcal{P}$-SPACE, is the class of languages accepted by polynomial-space-bounded deterministic Turing machines.

Recall, by Theorem 10.1, that if L is accepted by an NDTM of space complexity $p(n)$ for some polynomial p, then it is accepted by a DTM of space complexity $p^2(n)$. Thus $\mathcal{P}$-SPACE also includes all languages accepted by polynomial-space-bounded NDTM's. Since every polynomial-time-bounded NDTM is polynomial-space-bounded, we immediately have $\mathcal{NP}$-TIME $\subseteq$ $\mathcal{P}$-SPACE (Fig. 10.13). Moreover, since $\mathcal{P}$-TIME $\subseteq$ $\mathcal{NP}$-TIME $\subseteq$ $\mathcal{P}$-SPACE, if $\mathcal{P}$-TIME $=$ $\mathcal{P}$-SPACE, then $\mathcal{P}$-TIME $=$ $\mathcal{NP}$-TIME, so it is even more unlikely that every language in $\mathcal{P}$-SPACE has a deterministic polynomial-time algorithm than that every language in $\mathcal{NP}$-TIME has a deterministic polynomial-time algorithm.

We can exhibit a fairly natural language L_r which is *complete for $\mathcal{P}$-SPACE*. That is, L_r is in $\mathcal{P}$-SPACE and if we are given a $T(n)$ time-bounded deterministic TM that accepts L_r, then for each language L in $\mathcal{P}$-SPACE we can find a $T(p_L(n))$-time-bounded deterministic TM that accepts L, where p_L is a polynomial that depends on L. The same holds if "deterministic" is replaced by "nondeterministic." Thus, if L_r is in $\mathcal{P}$-TIME, then $\mathcal{P}$-TIME $=$ $\mathcal{NP}$-TIME $=$ $\mathcal{P}$-SPACE. If L_r is in $\mathcal{NP}$-TIME, then $\mathcal{NP}$-TIME $=$ $\mathcal{P}$-SPACE. The language L_r is the set of regular expressions whose complements denote nonempty sets.

Lemma 10.2. Let $M = (Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ be a $p(n)$-space-bounded single-tape DTM where p is some polynomial. Then there is another polynomial p' such that if $x \in I^*$ and $|x| = n$, then we can construct a regular expression R_x in time $p'(n)$, such that the complement of R_x denotes the empty set if and only if M does not accept x.

Proof. The proof bears a strong resemblance to that of Theorem 10.3. There we used Boolean functions as the language with which to describe computa-

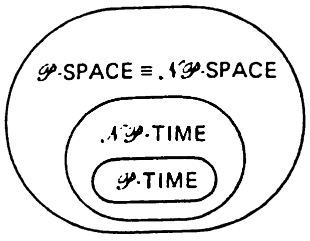


Fig. 10.13 Space time relationships.

tions of a Turing machine. Here we use the language of regular expressior There is a certain conciseness to regular expressions, and we are able to use regular expression to express facts about ID sequences that are much long than the regular expression itself.

The regular expression R_x can be thought of as representing the sequenc of ID's of M which do not demonstrate acceptance of x. For our purpos we use the alphabet $\Delta = T \cup \{[qX] | q \in Q$ and $X \in T\}$. The symbol $[qX$ in $Q \times T$ represents a cell of M's input tape which is scanned by the tap head and holds symbol X, while M is in state q. If M accepts x, it does so b a sequence of moves in which at most $p(|x|)$ tape cells are used. Thus we ma represent an ID by a string w_i of exactly $p(n)$ symbols, all but one of whic are from T; the remaining symbol is of the form $[qX]$. A sequence of move by M can be represented by a string $\#w_1\#w_2\# \cdots w_k\#$ for some $k \geq 1$, wher $\#$ is a new punctuation symbol and each w_i is a string in Δ^* representing a ID, with $|w_i| = p(n)$. Each w_i is padded out with blanks if necessary to mak it of length $p(n)$.

We wish to construct R_x to represent those strings in $(\Delta \cup \{\#\})^*$ whicl do not represent accepting computations of M with input x. Thus R_x will represent all strings in $(\Delta \cup \{\#\})^*$ if and only if M does not accept x. A strin y in $(\Delta \cup \{\#\})^*$ fails to represent an accepting computation of M only if on or more of the following four conditions are met. We assume $|x| = n$.

1. y is not of the form $\#w_1\#w_2\# \cdots w_k\#$ for some $k \geq 1$, where for each i $1 \leq i \leq k$, $|w_i| = p(n)$, and all but one symbol of w_i is in T; the remaining symbol is of the form $[qX]$.
2. The initial ID w_1 is wrong. That is, y does not begin with $\#[q_0a_1]a_2 \cdots a_n$bb . . . b#, where $x = a_1 \cdots a_n$ and the number of blanks is $p(n) - n$.
3. M never enters the final state. That is, no symbol of the form $[q_fX]$ appears in y.
4. y has two consecutive ID's, the second of which does not follow from the first by one move of M.

We shall write R_x as $A + B + C + D$ where $A, \ldots, D$ are regular expressions representing the sets defined by conditions $1, \ldots, 4$, respectively.

It is useful to develop a shorthand for writing regular expressions. If we have an alphabet $S = \{c_1, c_2, \ldots, c_m\}$, then we use S itself to denote the regular expression $c_1 + c_2 + \cdots + c_m$. Also, we use S^i to denote the regular expression $(S)(S) \cdots (S)$ (i times), that is, strings of length i over alphabet S. Note that the length of the regular expression represented by S is $2m - 1$; the length of the regular expression represented by S^i is $i(2m + 1)$. Similarly, if $S_1 - S_2 = \{d_1, d_2, \ldots, d_r\}$, then we use $S_1 - S_2$ to denote the regular expression $d_1 + d_2 + \cdots + d_r$.

Given these conventions, the regular expression A can be written

$$\begin{aligned} A = {} & \Delta^* + \Delta^*\#\,\Delta^* + \Delta(\Delta + \#)^* + (\Delta + \#)^*\Delta \\ & + (\Delta + \#)^*\#T^*\#(\Delta + \#)^* \\ & + (\Delta + \#)^*\#\Delta^*(Q \times T)\Delta^*(Q \times T)\Delta^*\#(\Delta + \#)^* \\ & + (\Delta + \#)^*\#\Delta^{p(n)+1}\Delta^*\#(\Delta + \#)^* \\ & + A_0 + A_1 + \cdots + A_{p(n)-1}, \end{aligned} \tag{10.7}$$

where the A_i's are defined subsequently. The first seven terms of (10.7) represent, respectively:

i) strings with no #,
ii) strings with only one #,
iii) strings not beginning with #,
iv) strings not ending with #,
v) strings with no symbol of $Q \times T$ between two #'s,
vi) strings with more than one symbol of $Q \times T$ between two #'s, and
vii) strings with more than $p(n)$ symbols of Δ between #'s.

The remaining terms A_i are given by $A_i = (\Delta + \#)^*\#\,\Delta^i\#(\Delta + \#)^*$, and together denote the set of strings with fewer than $p(n)$ symbols of Δ between two #'s.

It is left to the reader to check that the regular expression of (10.7) does in fact represent the strings meeting condition 1. Also, note that the first six terms are of fixed length, depending only on M. The length of the seventh term is clearly $O(p(n))$. The term A_i is of length $O(i)$, so the sum of their lengths, and hence the length of (10.7), is $O(p^2(n))$, the constant of proportionality depending only on M, not x. Moreover, it is easy to check that expression (10.7) can easily be written down in time which is polynomial in n.

At this point, we observe that for B, C, and D we need write only expressions which represent all the strings which satisfy condition 2, 3, or 4, but which violate condition 1, i.e., we need only generate strings of the form $\#w_1\# \cdots w_k\#$, where $|w_i| = p(n)$, and w_i is in $T^*(Q \times T)T^*$. For simplicity, however, we shall write expressions defining sets of strings which satisfy conditions 2, 3, and 4 and violate condition 1, and also including some strings which satisfy condition 2, 3, or 4 as well as condition 1. Since those in the latter class are already in R_x by the definition of expression A, their presence or absence in B, C, and D is immaterial.

Assuming the input string $x = a_1a_2 \cdots a_n$, we express B as $B_1 + B_2 + \cdots + B_{p(n)}$, where

$$\begin{aligned} B_1 &= \#(\Delta - [q_0a_1])\Delta^{p(n)-1}\#(\Delta + \#)^*, \\ B_i &= \#\Delta^{i-1}(T - \{a_i\})\Delta^{p(n)-i}\#(\Delta + \#)^* \end{aligned}$$

for $1 < i \leq n$, and

$$B_i = \#\Delta^{i-1}(T - \{\mathrm{b}\})\Delta^{p(n)-i}\#(\Delta + \#)^*$$

for $n < i \leq p(n)$. Thus B_i differs from the initial ID in at least the ith input symbol. Clearly, B is of length $O(p^2(n))$, the constant of proportionality depending only on M.

We write C as $((\Delta + \#) - (\{q_f\} \times T))^*$. Certainly the length of this expression depends only on M.

Finally, we may construct D as follows. Suppose we are given a string y which does represent a legal computation of M. Then it is of the form $\#w_1\#w_2\# \cdots w_k\#$, where for each i, $|w_i| = p(n)$ and w_{i+1} is an ID that results from ID w_i by one move of M. Observe that given any three consecutive symbols $c_1c_2c_3$ of y, we can uniquely determine the symbol, call it $f(c_1c_2c_3)$, which must appear $p(n) + 1$ symbols to the right of c_2. For example, if c_1, c_2, and c_3 are each in T, then c_2 may not change at the next ID, so $f(c_1c_2c_3) = c_2$. If c_1 and c_2 are in T, $c_3 = [qX]$ in $Q \times T$, and $\delta(q, X) = (p, X, L)$, then $f(c_1c_2c_3) = [pc_2]$. The remaining rules defining f, including those where c_1 or c_2 is in $Q \times T$, or one of the c's is $\#$, should be obvious to the reader and are left as exercises.

D is therefore the sum of terms $D_{c_1c_2c_3}$ for all c_1, c_2, and c_3 in $\Delta \cup \{\#\}$, where

$$D_{c_1c_2c_3} = (\Delta + \#)^*c_1c_2c_3(\Delta + \#)^{p(n)-1}\bar{f}(c_1c_2c_3)(\Delta + \#)^*$$

and $\bar{f}(c_1c_2c_3) = \Delta \cup \{\#\} - f(c_1c_2c_3)$. We see that the length of D is $O(p(n))$, and that D can be written down in time $O(p(n))$, the constant of proportionality depending only on M.

Thus the regular expression R_x can be constructed in time $p'(n)$ where p' is a polynomial of the order of p^2. Moreover, R_x denotes $(\Delta \cup \{\#\})^*$ if and only if x is not accepted by M. □

We note that in Lemma 10.2 we constructed a regular expression whose alphabet $\Delta \cup \{\#\}$ was of a size that depends on M. We wish to talk about the "language of regular expressions whose complements (with respect to their alphabets) denote nonempty sets." Since a language must have a finite alphabet, we shall encode a regular expression over an arbitrary alphabet Σ as follows.

1. $+$, $*$, $\emptyset$, ϵ, and parentheses denote themselves.
2. The ith symbol of the alphabet Σ (in arbitrary order) is denoted $\#w\#$, where w is the decimal representation of i.

Thus only 17 symbols are needed to encode any regular expression over any alphabet. Let $\Gamma = \{+, *, \emptyset, \epsilon, (,), \#, 0, 1, \ldots, 9\}$ be this alphabet.

We define the language $L_r \subseteq \Gamma^*$ to be the set of encodings of those regular expressions R such that the complement of R denotes a nonempty set. If R

is over alphabet Σ, then the complementation is with respect to Σ. Note that if R is of length $n \geq 2$, its encoding is of length at most $3n \log n$.

Lemma 10.3. L_r is in $\mathscr{P}$-SPACE.

Proof. By Theorem 10.1 it suffices to construct a polynomial-space-bounded NDTM M to accept L_r. Let R be a regular expression whose encoding is presented to M as an input string. By Theorem 9.2 we may, given a regular expression of length n, construct a nondeterministic finite automaton A with at most $2n$ states that accepts the language denoted by the regular expression. The NDTM M operates by first building the automaton A from the encoding of R presented to M. Observe that in the construction in Theorem 9.2 the next state function of A can be constructed in time no greater than $O(n^2)$.

Suppose the complement of the set denoted by R is not empty. Then there is a string $x = a_1 \cdots a_m$ such that x is not in the language denoted by R. M guesses x symbol by symbol. M uses an array of $2n$ bits to keep track of S_i, the set of states which A may be in after having read $a_1 \cdots a_i$, $1 \leq i \leq m$. To begin, S_0 is the set of states reachable from the initial state of A on ϵ-transitions only. After M has guessed $a_1a_2 \cdots a_i$, A will be in some set of states S_i. When M guesses the next input symbol a_{i+1}, it first computes $T_{i+1} = \{s' | s' \in \delta_A(s, a_{i+1})$ and $s \in S_i\}$, where δ_A is the next state function of A, and then computes S_{i-1} by adding to T_{i-1} each state in $\delta_A(s', \epsilon)$ for s' in T_{i-1}. (Note the similarity to Algorithm 9.1.)

When M guesses $a_1a_2 \cdots a_m$, it will determine that S_m contains no final state of A. On finding a set of states without a final state, M accepts its own input string, the encoding of the regular expression R. Thus M accepts the encoding of R if and only if the complement of the set denoted by R is nonempty. □

We can now show that the language L_r consisting of the encodings of those regular expressions which denote sets whose complements are not empty is polynomial-space complete.

Theorem 10.14. If L_r is accepted by a DTM of time complexity $T(n) \geq n$, then for each $L \in \mathscr{P}$-SPACE, there is a polynomial p_L such that L is accepted in time $T(p_L(n))$.

Proof. Let M be a DTM accepting L. By Lemma 10.2 and the discussion following, there is a polynomial-time-bounded algorithm, say of time complexity $p_1(n)$, to construct the regular expression R_x from an input string x. R_x is over a fixed 17-symbol alphabet and is surely no longer than $p_1(|x|)$. By Lemma 10.2, the complement of R_x is empty if and only if x is in L. We can, by hypothesis, test whether the complement of R_x is empty in $T(|R_x|) = T(p_1(|x|))$ steps. The total time taken to construct R_x is $p_1(x)$ and to test R_x is $T(p_1(|x|))$. Since $T(n) \geq n$, choosing $p_L(n) = 2p_1(n)$ is sufficient to prove the theorem. □

Corollary. L_r is in $\mathscr{P}$-TIME if and only if $\mathscr{P}$-TIME $=$ $\mathscr{P}$-SPACE.

Proof. If $\mathscr{P}$-TIME $=$ $\mathscr{P}$-SPACE, then L_r is in $\mathscr{P}$-TIME by Lemma 10.3. The converse follows from Theorem 10.14 with $T(n)$ a polynomial. □

Theorem 10.15. If L_r is accepted by an NDTM of time complexity $T(n) \geq n$, then for each $L \in \mathscr{P}$-SPACE, there is a polynomial p_L such that L is accepted by an NDTM in time $T(p_L(n))$.

Proof. Analogous to the proof of Theorem 10.14. □

Corollary. L_r is in $\mathscr{NP}$-TIME if and only if $\mathscr{NP}$-TIME $=$ $\mathscr{P}$-SPACE.

EXERCISES

10.1 Give all the legal sequences of moves of the NDTM in Fig. 10.1 (p. 368) on input 10101. Does the NDTM accept this input?

10.2 Informally describe an NDTM or a nondeterministic Pidgin ALGOL program that will accept the following.

a) The set of strings $10^{i_1}10^{i_2} \cdots 10^{i_k}$ such that $i_r = i_s$ for some $1 \leq r < s \leq k$.

b) The set of strings xcy such that x and y are in $\{a, b\}^*$ and x is a subword of y.

c) The same as (b), but x is a subsequence (p. 361) of y.

10.3 Describe a RAM program that will simulate a nondeterministic RAM program.

10.4 Let M be an $m \times n$ matrix of 0's and 1's. Write a Pidgin ALGOL program to find a smallest $s \times n$ submatrix S of M such that if $M[i, j] = 1$, then for some $1 \leq k \leq s$, $S[k, j] = 1$. What is the time complexity of your program?

10.5 Show that the following functions are space-constructible by informally describing a DTM which places a marker symbol in the appropriate square of one of its tapes.

a) n^2

b) $n^3 - n^2 + 1$

c) 2^n

d) $n!$

10.6 Show that if a single-tape NDTM of space complexity $S(n)$ has s states and t tape symbols, and it accepts a word of length n, then it accepts the word in a sequence of at most $sS(n)t^{S(n)}$ moves.

***10.7** Show how to evaluate a Boolean expression in time $O(n)$ on a RAM.

10.8 Show that the language consisting of those Boolean functions that are not tautologies† is NP-complete.

10.9 Show that the *subgraph isomorphism problem* (Is a given undirected graph G isomorphic to some subgraph of a given undirected graph G'?) is NP-complete.

† A *tautology* is a Boolean expression that has the value 1 for all assignments of values to its variables.

[*Hint:* Use the fact that the clique problem or that the Hamilton circuit problem is NP-complete.]

10.10 Show that the *knapsack problem* (Given a sequence of integers $S = i_1, i_2, \ldots, i_n$ and an integer k, is there a subsequence of S that sums to exactly k?) is NP-complete. [*Hint:* Use the exact cover problem.]

***10.11** Show that the *partition problem* (the knapsack problem of Exercise 10.10 with $\Sigma_{j=1}^{n} i_j = 2k$) is NP-complete.

10.12 Show that the *traveling salesman problem* (Given a graph G with integer weights on its edges, find a cycle which includes each vertex and whose edges have weights summing to at most k) is NP-complete.

****10.13** Show that the problem of determining whether a regular expression without *'s (i.e., using only operators + and ·) does not denote all strings of some fixed length is NP-complete. [*Hint:* Transform the CNF-satisfiability problem to the regular expression problem.]

****10.14** Show that the problem of determining whether a regular expression over the alphabet {0} does not denote 0* is NP-complete.

****10.15** Let $G = (V, E)$ be an undirected graph and let v_1 and v_2 be two distinct vertices. A *cutset* for v_1 and v_2 is a subset $S \subseteq E$ such that every path between v_1 and v_2 contains an element of S. Show that the problem of determining whether a graph has a cutset of size k for v_1 and v_2 such that each part has the same number of vertices is NP-complete.

****10.16** The *one-dimensional package placement problem* is the following. Let $G = (V, E)$ be an undirected graph and k a positive integer. Does there exist an ordering $v_1, v_2, \ldots, v_n$ for V such that

$$\sum_{(v_i, v_j) \in E} |i - j| \leq k?$$

Show this problem to be NP-complete.

****10.17** Prove that the colorability problem is NP-complete even if k is restricted to 3 and the maximum degree of any vertex is 4.

****10.18** Prove Exercise 10.17 for planar graphs. [*Hint:* Show that the planar graph of Fig. 10.14 can be colored with three colors only if v_1 and v_1' are the same color and v_2 and v_2' are the same color. Combine this result with your answer for Exercise 10.17.]

***10.19** Show that the clique problem is NP-complete by directly representing computations of an NDTM instead of using the transformation from 3-CNF-satisfiability.

***10.20** Consider a directed graph with two designated vertices s and t. Assign to each edge an integer "capacity." One can construct a maximal flow from s to t by repeatedly finding a path from s to t and increasing the flow along the path by the maximum allowed by the capacities of the edges. Show that the problem of finding the smallest set of paths on which to increase flow is NP-complete. [*Hint:* Consider a graph of the form shown in Fig. 10.15 and relate the problem to the knapsack problem.]

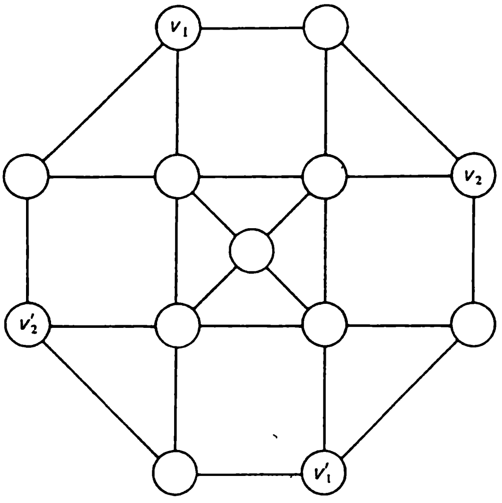


Figure 10.14

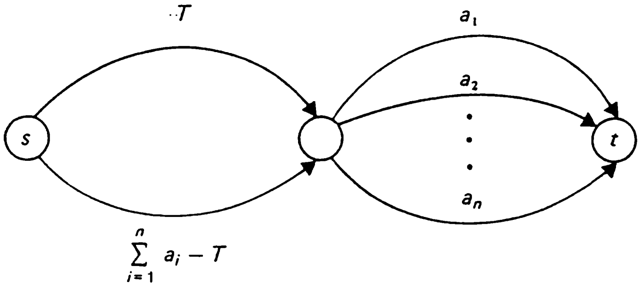


Figure 10.15

**10.21 The *equal execution time scheduling problem* is as follows. Given a set of jobs $S = \{J_1, \ldots, J_n\}$, a time limit t, a number of processors p, and a partial order $<$ on S, does there exist a *schedule* for S requiring at most t time units, that is, a mapping f from S to $\{1, 2, \ldots, t\}$ such that at most p jobs are mapped to any one integer, and if $J < J'$, then $f(J) < f(J')$? Show that this problem is NP-complete.

10.22 Show each of the problems listed in Theorem 10.2 (p. 378) to be in $\mathcal{NP}$-TIME.

10.23 Let $p_1(x)$ and $p_2(x)$ be polynomials. Show that there is a polynomial which for all values of x exceeds $p_1(p_2(x))$.

10.24 Write out E_{ijkt} as on p. 382 if $\delta(q_k, x_j) = \{(q_3, X_4, L), (q_9, X_2, R)\}$.

**10.25 Give a polynomial algorithm to test for 2-satisfiability.

*10.26 Certain subcases of the graph isomorphism problem such as for planar graphs are known to be easy. Other subcases are as hard as the general problem. Show that the isomorphism problem for rooted directed acyclic graphs is of the same complexity as the isomorphism problem for arbitrary graphs.

*10.27 A *context-sensitive language* is one which is accepted by an NDTM of space complexity $n+1$ (called a *linear-bounded automaton*—see Hopcroft and Ullman [1969]). Show that the problem of determining whether a given linear-bounded automaton accepts a given input is polynomial-space complete. That is, the problem of membership in an arbitrary context-sensitive language is complete for $\mathcal{P}$-SPACE.

10.28 Show that the problem of determining whether two regular expressions are equivalent is polynomial-space complete.

**10.29 Describe an algorithm for accepting L_r, the set of encodings of regular expressions R such that R denotes a nonempty set, which can be implemented on a linear-space-bounded DTM.

Research Problems

10.30 The obvious open problems are to resolve whether $\mathcal{P}$-TIME = $\mathcal{NP}$-TIME or $\mathcal{NP}$-TIME = $\mathcal{P}$-SPACE. In view of the amount of work that has been done searching for polynomial-time algorithms for NP-complete problems, it is likely that the problem is at least as hard as some classical problems of mathematics, such as Fermat's conjecture (Is $x^n + y^n = z^n$ solvable in integers for $n \geq 3$?) or the four-color problem.

10.31 Failing 10.30, it would even be of interest to obtain a nontrivial result giving a function $T(n)$ such that every language in $\mathcal{NP}$-TIME was of (deterministic) time complexity $T(n)$. Even $T(n) = 2^n$ has not been shown.

BIBLIOGRAPHIC NOTES

More information about nondeterministic Turing machines can be found in Hopcroft and Ullman [1969]. Theorem 10.1, relating the space complexity of deterministic and nondeterministic TM's, is by Savitch [1970].

The key theorem that satisfiability is NP-complete is due to Cook [1971b]. A large number of classical NP-complete problems were exhibited by Karp [1972], who clearly demonstrated the importance of the concept. Since then, additions to the family of known NP-complete problems have been made by Sahni [1972], Sethi [1973], Ullman [1973], Rounds [1973], Ibarra and Sahni [1973], Hunt and Rosenkrantz [1974], Garey, Johnson, and Stockmeyer [1974], Bruno and Sethi [1974], and many others. It is interesting to note that prior to Cook's pioneering paper, several workers showed some NP-complete problems to be polynomially related to one another without realizing the extent of the class. For example, Dantzig, Blattner, and Rao [1966] related the traveling salesman problem to the shortest-path problem with negative weights. Divetti and Grasselli [1968] showed the relation between the set cover problem and the feedback edge set problem.

Complete problems for $\mathcal{P}$-SPACE were first considered in Meyer and Stockmeyer [1972]. The first space-complete language appeared implicitly in Savitch [1971], where a language (the set of threadable mazes) complete for log n space was defined. Jones [1973] and Stockmeyer and Meyer [1973] treat restricted forms of polynomial-time reducibility between problems. Book [1972 and 1974] shows certain complexity classes incomparable.

Exercises 10.8 and 10.9 are from Cook [1971b]. Exercises 10.10–10.12 are from Karp [1972]. Exercise 10.13 is from Stockmeyer and Meyer [1973] and Hunt [1973a]. Exercises 10.14 and 10.28 are from Stockmeyer and Meyer [1973]. Exercises 10.15–10.18 are from Garey, Johnson, and Stockmeyer [1974], and Fig. 10.14 is an improvement suggested by M. Fischer. A proof that planar 3-colorability is NP-complete appears in Stockmeyer [1973]. Exercise 10.20 is from Even [1973], 10.21 from Ullman [1973], 10.25 from Cook [1971b], and the reduction needed for Exercise 10.27 is from Karp [1972]. The proof of Theorem 10.13 was suggested to us by S. Even.

SOME PROVABLY INTRACTABLE PROBLEMS

CHAPTER 11

In this chapter we develop proofs that the emptiness-of-complement problem for two classes of extended regular expressions is intractable, i.e., any algorithm to solve the problem for either class requires at least exponential time. For one of the classes we derive a lower bound for the problem which is substantially greater than exponential. In particular we show that the problem requires more than

$$2^{2^{\cdot^{\cdot^{\cdot^{2^{n}}}}}}$$

time for any finite number of 2's. Before proving the lower bounds we first consider hierarchy results showing that "the more time or space allowed for computation, the more languages there are that can be recognized."

11.1 COMPLEXITY HIERARCHIES

In Chapter 10 we showed certain problems to be complete for nondeterministic polynomial time or complete for polynomial space. To prove particular problems to be complete, we showed how to express an arbitrary problem in $\mathscr{NP}$-TIME or $\mathscr{P}$-SPACE in terms of the specific problem. The technique of proof was essentially one of simulation. For example, we showed that the satisfiability problem for Boolean expressions is NP-complete, and we showed that the emptiness-of-complement problem for regular expressions is polynomial-space complete, obtaining our result in both cases by a direct embedding of Turing machine computations into instances of these problems. We showed other problems to be complete by reducing to them a problem already known to be complete for the appropriate class of problems. Thus we demonstrated that both $\mathscr{NP}$-TIME and $\mathscr{P}$-SPACE have "hardest" problems, ones whose complexity is at least as great as any problem in the class.

However compelling the circumstantial evidence may be, no one has yet been able to find a problem in $\mathscr{NP}$-TIME or $\mathscr{P}$-SPACE that can be shown not to be in $\mathscr{P}$-TIME. Moreover, the techniques of Chapter 10 seem sufficient only to show that a problem is at least as hard as any other problem in some class. To actually prove that a problem is not in $\mathscr{P}$-TIME we need a technique to show that there exists at least one language not accepted by any deterministic Turing machine of polynomial-time complexity. The chief technique used for this purpose is diagonalization. Although the technique does not seem powerful enough to show $\mathscr{P}$-TIME $\neq$ $\mathscr{NP}$-TIME, it has been used to establish hierarchy results for both space and time complexity, for both deterministic and nondeterministic Turing machines. Each hierarchy theorem is of the following form: Given two "well-behaved" functions $f(n)$ and $g(n)$, where $f(n)$ grows "faster" than $g(n)$, there is a language of complexity $f(n)$ but not of complexity $g(n)$.

11.2 THE SPACE HIERARCHY FOR DETERMINISTIC TURING MACHINES

In this section we prove a space hierarchy theorem for deterministic Turing machines. Similar hierarchies for time and for nondeterministic Turing machines exist but since they are not required for our purposes here, they are left as exercises. We recall from Corollary 3 to Lemma 10.1 that if a language L is accepted by a k-tape DTM of space-complexity $S(n)$, then L is accepted by a single-tape DTM of space complexity $S(n)$. Thus we may restrict ourselves to single-tape DTM's.

To obtain the space hierarchy result we need to *enumerate* the DTM's, that is, assign an ordering to DTM's so that for each nonnegative integer i there is a unique DTM associated with i. Furthermore we require that each DTM appear infinitely often in the enumeration. We shall enumerate only the single-tape DTM's with input alphabet $\{0, 1\}$ because that will be all we need. The method of enumeration for all Turing machines is a simple extension.

We can, without loss of generality, make the following assumptions about the representation of a single-tape DTM.

1. The states are named $q_1, q_2, \ldots, q_s$ for some s, with q_1 the initial state and q_s the accepting state.
2. The input alphabet is $\{0, 1\}$.
3. The tape alphabet is $\{X_1, X_2, \ldots, X_t\}$ for some t, where $X_1 = b$, $X_2 = 0$, and $X_3 = 1$.
4. The next-move function δ is a list of quintuples of the form $(q_i, X_j, q_k, X_l, D_m)$, meaning that $\delta(q_i, X_j) = (q_k, X_l, D_m)$, i.e., q_i and q_k are states, X_j and X_l are tape symbols, and D_m is the direction, L, R or S, if $m = 0$, 1, or 2, respectively. We assume this quintuple is encoded by the string $1\,0^i\,1\,0^j\,1\,0^k\,1\,0^l\,1\,0^m\,1$.
5. The Turing machine itself is encoded by concatenating in any order the codes for each of the quintuples in its next-move function. Additional 1's may be prefixed to the string if desired. The result will be some string of 0's and 1's, beginning with 1, which we can interpret as an integer.

Any integer which cannot be decoded is deemed to represent the trivial Turing machine with an empty next-move function. Every single-tape DTM will appear infinitely often in the enumeration, since given a DTM, we may prefix 1's at will to find larger and larger integers representing the same set of quintuples.

We can now design a four-tape DTM M_0 which treats its input string x both as an encoding of a single-tape DTM M and also as the input to M. We shall design M_0 so that for each DTM M of given complexity there is at least one input string which M_0 accepts but which M rejects, or vice versa. One of the capabilities possessed by M_0 is the ability to simulate a Turing machine, given its specification. We shall have M_0 determine whether the Turing machine M accepts the input string x without using more than $S_1(|x|)$ cells for some function S_1. If M accepts x in space $S_1(|x|)$, then M_0 does not. Otherwise, M_0 accepts x. Thus, for all i, either M_0 disagrees with the behavior of the ith DTM on that input x which is the binary representation of i, or the ith DTM uses more than $S_1(|x|)$ cells on input x.

We say that M_0 *diagonalizes* over all DTM's of space complexity $S_1(n)$, since if we imagined an infinite two-dimensional table in which the ijth entry indicated whether the ith Turing machine accepted input j, then M_0's action would disagree with certain of the Turing machines along the diagonal of the table. In particular, M_0 would disagree with Turing machines accepting their inputs in space $S_1(n)$. By Corollary 3 to Lemma 10.1, there is some single-tape DTM M_0' equivalent to M_0 and of the same space complexity. Since M_0' is itself in the table (that is, M_0' is the kth DTM for some value of k) and M_0' cannot disagree with itself, we may conclude that M_0 and M_0' are not of space complexity $S_1(n)$. The actual construction of M_0 is made complicated by our desire that M_0 should be of space complexity $S_2(n)$, where $S_1(n)$ and $S_2(n)$ are almost the same.

Definition. Let $f(n)$ be any function. Then $\inf_{n\to\infty} f(n)$ is the limit as n goes to infinity of the greatest lower bound of $f(n)$, $f(n+1)$, $f(n+2), \ldots$.

Example 11.1. Since $(n^2+1)/n^2$ is monotonically decreasing

$$\inf_{n\to\infty} \frac{n^2+1}{n^2} = \lim_{n\to\infty} \frac{n^2+1}{n^2} = 1.$$

For a second example, let $f(n) = 1 - 1/n$ if n is not a power of 2 and $f(n) = n$ if n is a power of 2. Then $\inf_{n\to\infty} f(n) = 1$, since the greatest lower bound of $f(n), f(n+1), f(n+2), \ldots$ is either $1 - 1/n$ if n is not a power of 2 or $1 - 1/(n+1)$ if n is a power of 2. □

Theorem 11.1. Let $S_1(n) \geq n$ and $S_2(n) \geq n$ be two space-constructible functions with

$$\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0.$$

Then there is a language L accepted by a DTM of space complexity $S_2(n)$ but by no DTM of space complexity $S_1(n)$.†

† In the literature, one often finds space complexity for Turing machines defined so as to ignore the number of cells scanned on the input tape; the input tape is not permitted to have its symbols changed, however. Under this model, it makes sense to talk about space complexity functions less than n, and the conditions $S_1(n) \geq n$ and $S_2(n) \geq n$ can

Proof. Let M_0 be a four-tape DTM which operates as follows on an input string x of length n.

1. M_0 marks off $S_2(n)$ cells on each tape. After doing so, if any tape head of M_0 attempts to move off the marked cells, M_0 halts without accepting.
2. If x is not the encoding of some single-tape DTM, M_0 halts without accepting.
3. Otherwise let M be the DTM encoded by x. M_0 determines t, the number of tape symbols used by M, and s, its number of states. The third tape of M_0 can be used as "scratch" memory to calculate t. Then M_0 lays off on its second tape $S_1(n)$ blocks of $\lceil \log t \rceil$ cells each, the blocks being separated by single cells holding a marker #, i.e., there are $(1 + \lceil \log t \rceil)\, S_1(n)$ cells in all, provided $(1 + \lceil \log t \rceil)\, S_1(n) \leq S_2(n)$. Each tape symbol occurring in a cell of M's tape will be encoded as a binary number in the corresponding block of the second tape of M_0. Initially, M_0 places its input, in binary coded form, in the blocks of tape 2, filling the unused blocks with the code for the blank.
4. On tape 3, M_0 sets up a block of $\lceil \log s \rceil + \lceil \log S_1(n) \rceil + \lceil \log t \rceil\; S_1(n)$ cells, initialized to all 0's, provided again that this number of cells does not exceed $S_2(n)$. Tape 3 is used as a counter to count up to $sS_1(n)t^{S_1(n)}$.
5. M_0 simulates M, using tape 1, its input tape, to determine the moves of M and using tape 2 to simulate the tape of M. The moves of M are counted in binary in the block of tape 3, and tape 4 is used to hold the state of M. If M accepts, then M_0 halts without accepting. M_0 accepts if M halts without accepting, if the simulation of M attempts to use more than the allotted cells on tape 2, or if the counter on tape 3 overflows, i.e., the number of moves made by M exceeds $sS_1(n)t^{S_1(n)}$.

The Turing machine M_0 described above is of space complexity $S_2(n)$, and it accepts some language L. Suppose L were accepted by some DTM M_i of space complexity $S_1(n)$. By Corollary 3 to Lemma 10.1 we may assume that M_i is a single-tape Turing machine. Let M_i have s states and t tape symbols. By stringing together the quintuples of M_i, with leading 1's if necessary, we can find a string w of length n representing M_i, where n is sufficiently large so that $S_2(n)$ is larger than $\mathrm{MAX}[(1 + \lceil \log t \rceil)S_1(n),\ \lceil \log s \rceil + \lceil \log S_1(n) \rceil + \lceil \log t \rceil\; S_1(n)]$. The fact that $\inf_{n\to\infty} [S_1(n)/S_2(n)] = 0$ guarantees that such an n can be found. Then, when given w for input, M_0 has enough room to simulate M_i. M_0 accepts w if and only if M_i does not accept it. But we assumed that M_i accepted L, i.e., M_i agreed with M_0 on all inputs.

be removed from the hypothesis of the theorem. Since we deal only with large space complexities here, the result for complexities less than n is not worth the added details, and we omit it. In this theorem the restriction that $S_1(n)$ is space-constructible can be relaxed.

We thus conclude that M_i does not exist, i.e., L is not accepted by any DTM of space complexity $S_1(n)$. □

A typical application of Theorem 11.1 is to show, for example, that there is a language accepted by a DTM of space complexity, say, $n^2 \log n$ but by no DTM of space complexity n^2. Some other applications will be seen in the remainder of this chapter.

11.3 A PROBLEM REQUIRING EXPONENTIAL TIME AND SPACE

In Section 10.6 we considered the emptiness-of-complement problem for regular expressions and showed it to be complete for polynomial space. Since the class of regular sets is closed under both intersection and complementation, adding the intersection operator $\cap$ and complementation operator $\neg$ to the notation of regular expressions does not increase the class of sets that can be described. However, the use of the intersection and complementation operators greatly shortens the length of the expressions needed to describe certain regular sets. Because of this ability to be concise, even more time is required to solve some problems for extended regular expressions than is required for "ordinary" regular expressions, when time is measured as a function of the length of the given expression.

Definition. An *extended regular expression* over an alphabet Σ is defined as follows:

1. ϵ, $\emptyset$, and a, for a in Σ, are extended regular expressions denoting $\{\epsilon\}$, the empty set, and $\{a\}$, respectively.
2. If R_1 and R_2 are extended regular expressions denoting the languages L_1 and L_2, respectively, then $(R_1 + R_2)$, $(R_1 \cdot R_2)$, (R_1^*), $(R_1 \cap R_2)$, and $(\neg R_1)$ are extended regular expressions, denoting $L_1 \cup L_2$, L_1L_2, L_1^*, $L_1 \cap L_2$, and $\Sigma^* - L_1$, respectively.

Redundant pairs of parentheses may be deleted from extended regular expressions if we assume the operators have the following increasing order of precedence.

$$+ \quad \cap \quad \neg \quad \cdot \quad *$$

Furthermore, the operator $\cdot$ is usually omitted. For example, $a \neg b * + c \cap d$ means

$$((a \cdot (\neg(b^*))) + (c \cap d)).$$

If an extended regular expression has no complementation signs, then it is said to be *semiextended*. The *emptiness-of-complement problem* for semiextended regular expressions is to determine whether the complement of the set denoted by a given expression R is empty (or equivalently, to determine whether R denotes all strings over the input alphabet). For example.

the regular expression

$$b^* + b^*aa^*(\epsilon + b(a+b)(a+b)^*) + b^*aa^*b$$

denotes $(a+b)^*$ since

$$b^* + b^*aa^*(\epsilon + b(a+b)(a+b)^*) = \neg b^*aa^*b.$$

We now proceed to prove that the emptiness-of-complement problem for semiextended regular expressions is not in $\mathscr{P}$-SPACE, and hence not in $\mathscr{NP}$-TIME because $\mathscr{NP}$-TIME is contained in $\mathscr{P}$-SPACE. In fact, we shall show that there exist semiextended regular expressions of length n for which at least $c'c^{\sqrt{n/\log n}}$ space (and hence time) is required to solve the emptiness-of-complement problem, for some constants $c' > 0$ and $c > 1$.

In Section 11.4 we shall see that the emptiness-of-complement problem for the full class of extended regular expressions is much harder than that for semiextended expressions. In anticipation of these later results, we shall couch many of the results of this section in terms of the full class of extended regular expressions.

In addition to the notational shorthands used in Section 10.6, we shall use $\Sigma_{i=1}^{k} R_i$ to denote the extended regular expression $R_1 + R_2 + \cdots + R_k$. We also use R^+ for RR^*. When we talk of the length of an expression, we are talking about the length of the actual expression denoted by these shorthand conventions.

We now introduce the notion of a yardstick. Let Σ be an alphabet and x any string in Σ^*. Then $\text{CYCLE}(x) = \{zy \mid x = yz \text{ with } y \in \Sigma^* \text{ and } z \in \Sigma^*\}$. Let # be a special marker symbol not in Σ. Then the set CYCLE(x#) is said to be a *yardstick* of length $|x\#|$. That is, a yardstick is the set of all cyclic permutations of the string x#. When no confusion arises, we shall sometimes refer to x# itself as the "yardstick."

We shall use a yardstick to measure the length of ID's in a valid computation of a Turing machine. First we show that some long yardsticks can be denoted by relatively short semiextended regular expressions.

Lemma 11.1. For every $k \geq 1$, there exists a yardstick of length greater than 2^k which can be denoted by a semiextended regular expression R such that $|R| \leq ck^2$ for some constant c independent of k.

Proof. Let $A = \{a_0, a_1, \ldots, a_k\}$ be an alphabet of $k+1$ distinct symbols. Let $x_0 = a_0a_0$, and for $1 \leq i < k$ let $x_i = x_{i-1}a_ix_{i-1}a_i$. Thus

$$x_i = (\cdots((a_0^2a_1)^2a_2)^2 \cdots a_i)^2.$$

The length of x_0 is 2, and the length of x_i is greater than twice the length of x_{i-1}. Thus the length of x_{k-1} is greater than or equal to 2^k, for $k \geq 1$.

Let CYCLE($x_{k-1}a_k$) be the desired yardstick. It remains to be shown that there is a short semiextended regular expression denoting the set CYCLE($x_{k-1}a_k$).

For $0 \leq i \leq k$, let A_i stand for $(a_0 + a_1 + \cdots + a_i)$ and $A - A_i$ stand for $(a_{i+1} + a_{i+2} + \cdots + a_k)$. Let

$$\begin{aligned} R_0 = {} & a_0 a_0 [(A - A_0)^+ a_0^2]^* (A - A_0)^+ \\ & + a_0 [(A - A_0)^+ a_0^2]^* (A - A_0)^+ a_0 \\ & + \quad [(A - A_0)^+ a_0^2]^* (A - A_0)^+ a_0 a_0 (A - A_0)^* \end{aligned}$$

and

$$\begin{aligned} R_i = {} & A_{i-1}^+ a_i A_{i-1}^+ a_i [(A - A_i)^+ (A_{i-1}^+ a_i)^2]^* (A - A_i)^+ A_{i-1}^* \\ & + \quad a_i A_{i-1}^+ a_i [(A - A_i)^+ (A_{i-1}^+ a_i)^2]^* (A - A_i)^+ A_{i-1}^+ \\ & + \quad A_{i-1}^+ a_i [(A - A_i)^+ (A_{i-1}^+ a_i)^2]^* (A - A_i)^+ A_{i-1}^+ a_i A_{i-1}^* \\ & + \quad a_i [(A - A_i)^+ (A_{i-1}^+ a_i)^2]^* (A - A_i)^+ A_{i-1}^+ a_i A_{i-1}^+ \\ & + \quad [(A - A_i)^+ (A_{i-1}^+ a_i)^2]^* (A - A_i)^+ A_{i-1}^+ a_i A_{i-1}^+ a_i (A - A_i)^* \end{aligned}$$

for $1 \leq i \leq k - 1$.

We claim that the semiextended regular expression

$$R = R_0 \cap R_1 \cap \cdots \cap R_{k-1} \cap A_{k-1}^* a_k A_{k-1}^*$$

denotes CYCLE$(x_{k-1} a_k)$. In proof, we show by induction on i that any string in $R_0 \cap R_1 \cap \cdots \cap R_i$ is of the form

$$\begin{aligned} & y_2 [(A - A_i)^+ x_i]^* (A - A_i)^+ y_1 \\ & + [(A - A_i)^+ x_i]^* (A - A_i)^+ x_i (A - A_i)^*, \end{aligned}$$

where $y_1 y_2 = x_i$.

BASIS. For $i = 0$, we observe that R_0 denotes precisely those strings of the form

$$\begin{aligned} & a_0^j [(A - A_0)^+ a_0^2]^* (A - A_0)^+ a_0^{2-j} \\ & + [(A - A_0)^+ a_0^2]^* (A - A_0)^+ a_0 a_0 (A - A_0)^*, \end{aligned}$$

for some $0 \leq j \leq 2$. Since $x_0 = a_0^2$, the result is immediate.

INDUCTIVE STEP. Assume that $R_0 \cap R_1 \cap \cdots \cap R_{i-1}$ denotes all strings of the form

$$\begin{aligned} & y_2' [(A - A_{i-1})^+ x_{i-1}]^* (A - A_{i-1})^+ y_1' \\ & + [(A - A_{i-1})^+ x_{i-1}]^* (A - A_{i-1})^+ x_{i-1} (A - A_{i-1})^*, \end{aligned} \tag{11.1}$$

where $y_1' y_2' = x_{i-1}$.

Consider a string z in the intersection of (11.1) and R_i. Then, each maximal-length substring of z from A_{i-1}^+ must be x_{i-1} unless it is at the beginning or end of the string, in which case it is y_2' or y_1'. Thus each A_{i-1}^+ in R_i can be replaced by x_{i-1} (unless it is at the beginning or end of the regular expression) without changing the intersection of (11.1) and R_i. The A_{i-1}^+ and A_{i-1}^* at the beginning or end of R_i can be replaced by y_2' and y_1', respectively.

Observing that $x_{i-1}a_ix_{i-1}a_i = x_i$, we get

$$\begin{aligned} R_0 \cap R_1 \cap \cdots \cap R_i = {} & y_2'a_ix_{i-1}a_i[(A-A_i)^+x_i]^*(A-A_i)^+y_1' \\ & + a_ix_{i-1}a_i[(A-A_i)^+x_i]^*(A-A_i)^+x_{i-1} \\ & + \quad y_2'a_i[(A-A_i)^+x_i]^*(A-A_i)^+x_{i-1}a_iy_1' \\ & + \quad a_i[(A-A_i)^+x_i]^*(A-A_i)^+x_{i-1}a_ix_{i-1} \\ & + \quad [(A-A_i)^+x_i]^*(A-A_i)^+x_{i-1}a_ix_{i-1}a_i(A-A_i)^*. \end{aligned} \tag{11.2}$$

Finally, (11.2) can be rewritten as

$$\begin{aligned} & y_2[(A - A_i)^+x_i]^*(A - A_i)^+y_1 \\ & + [(A - A_i)^+x_i]^*(A - A_i)^+x_i(A - A_i)^*, \end{aligned}$$

where $y_1y_2 = x_i$. Thus the induction hypothesis follows. Setting $i = k - 1$ and intersecting with $A_{k-1}^*a_kA_{k-1}^*$, we see that R denotes CYCLE$(x_{k-1}a_k)$.

It remains to show that the length of the extended regular expression denoted by R is bounded by ck^2. This result follows immediately from the fact that there exists a constant c_1 such that the length of the regular expressions denoted by each shorthand R_i is bounded by c_1k. That is, A_i and $A - A_i$ denote extended regular expressions of length $O(k)$. Thus each term of the sum defining R_i denotes an extended regular expression of length $O(k)$. Hence the length of the semiextended regular expression R is bounded by ck^2 for some new constant c. □

A construction similar to that of Lemma 11.1 enables us to write a short regular expression denoting all strings in A^* except for $x = x_{k-1}$. This regular expression will find use along with the semiextended regular expression R for CYCLE$(x\#)$ just constructed.

Lemma 11.2. Let alphabet $A = \{a_0, a_1, \ldots, a_k\}$ and the strings x_i be as in Lemma 11.1. Then there is a regular expression of length $O(k^2)$ denoting $\neg x_{k-1}$.

Proof. The string x_{k-1} obeys the following rules.

i) It is nonempty and begins with a_0. The a_0's occur in pairs and each occurrence of a pair is followed by an a_1.
ii) For $1 \leq i \leq k - 2$, the a_i's also occur in pairs. In each pair the two a_i's are separated by a string in $a_0A_{i-1}^*$ and the second a_i in the pair is followed by a_{i+1}. That is to say, an a_i which has, to its left, symbols in A_{i-1} exclusively, or immediately to its left, a string in A_{i-1}^* preceded by a symbol in $A \dot{-} A_i$, is followed by a_0.
iii) There are exactly two instances of a_{k-1}. The first of these is followed by an a_0 and the other is at the right end of the string.

More importantly, x_{k-1} is the only string obeying these rules. We may prove by induction on j that if $b_1b_2 \cdots b_j$ is a prefix of a string satisfying the three rules, then $b_1b_2 \cdots b_j$ is a prefix of x_{k-1}. For example, by rule (i) the first symbol is a_0. Also by rule (i), a lone a_0 cannot end the string or be followed by any symbol but a_0, so the string begins a_0a_0. Next, by rule (i) again, a_0a_0 must have an a_1 following. Then, by rule (ii) with $i = 1$, the string $a_0a_0a_1$ must be followed by a_0, and so on. We leave a formal inductive proof for an exercise. Since x_{k-1} does in fact satisfy the rules, we see that it must be the unique string satisfying them.

We can easily write a regular expression denoting all strings which fail to satisfy one or more of the three rules. Strings not satisfying rule (i) are denoted by†

$$S_1 = \epsilon + (A - a_0)A^* + A^*a_0a_0[(A - a_1)A^* + \epsilon] + \\ [\epsilon + A^*(A - a_0)]a_0[(A - a_0)A^* + \epsilon].$$

In S_1 the first term denotes the empty string, the second those strings not starting with a_0, the third those strings containing a pair of a_0's not followed by a_1, and the last those strings containing an isolated a_0. Those strings not satisfying rule (ii) are denoted by

$$S_2 = \sum_{i=1}^{k-2} [A^*a_iA_{i-1}^*a_i[(A - a_{i+1})A^* + \epsilon] \\ + [\epsilon + A^*(A - A_i)]A_{i-1}^*a_i[A - a_0)A^* + \epsilon]].$$

Finally, those strings which fail to satisfy rule (iii) are

$$S_3 = A^*a_{k-1}A^*a_{k-1}A^+ + (A - a_{k-1})^*a_{k-1}(A - a_{k-1})^* \\ + (A - a_{k-1})^* + (A - a_{k-1})^*a_{k-1}(A - a_0)A^*.$$

In S_3 the first term includes all strings having more than two a_{k-1}'s in addition to those strings having exactly two a_{k-1}'s, the second of which is not at the right end of the string. Thus $S_1 \cup S_2 \cup S_3$ denotes $\neg x_{k-1}$. The length of $S_1 \cup S_2 \cup S_3$ is easily seen to be $O(k^2)$. □

Before proceeding to show that there is no polynomial-space- or polynomial-time-bounded algorithm for the emptiness-of-complement problem for semiextended regular expressions, we first make two preliminary observations.

Definition. A *homomorphism* h from Σ_1^* to Σ_2^* is a function such that for any strings x and y, $h(xy) = h(x)h(y)$. It follows that $h(\epsilon) = \epsilon$ and $h(a_1a_2 \cdots a_n) = h(a_1)h(a_2) \cdots h(a_n)$. Thus the homomorphism h is uniquely defined by the value of $h(a)$ for each a in Σ_1.

A homomorphism h is *length-preserving* if $h(a)$ is a single symbol of Σ_2 for each a in Σ_1. A length-preserving homomorphism merely renames

† We use $A - a_i$ to denote $\sum_{\substack{j=0 \\ j \neq i}}^{k} a_i$.

the symbols, possibly identifying several symbols by renaming them with the same symbol. If w is in Σ_2^*, then $h^{-1}(w) = \{x | h(x) = w\}$. If $L \subseteq \Sigma_2^*$, then $h^{-1}(L) = \{x | h(x) \in L\}$.

Example 11.2. Let $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{0, 1\}$. Define the homomorphism h by $h(a) = 010$, $h(b) = 1$ and $h(c) = \epsilon$. Then $h(abc) = 0101$. Observe that h is not length-preserving. Let L be the language denoted by the extended regular expression $1 + \neg 1^*$, or equivalently by the ordinary regular expression $1 + (0 + 1)^*0(0 + 1)^*$. Then $h^{-1}(L)$ is denoted by the extended regular expression $c^*bc^* + \neg(b + c)^*$ or the ordinary regular expression $c^*bc^* + (a + b + c)^*a(a + b + c)^*$. □

Lemma 11.3. Let h be a length-preserving homomorphism from Σ_1^* to Σ_2^*. Let R_2 be an extended regular expression denoting a set $S \subseteq \Sigma_2^*$. An extended regular expression R_1 denoting $h^{-1}(S)$ can be constructed such that the length of R_1 is bounded by a constant (depending only on h) times the length of R_2, and R_1 contains a complementation sign only if R_2 contains a complementation sign.

Proof. Replace each occurrence of a symbol of Σ_2 in R_2 by a regular expression denoting the set of symbols mapped onto it by h. For example, if $\{a_1, a_2, \ldots, a_r\}$ is the set of symbols mapped onto a, then replace a by $(a_1 + a_2 + \cdots + a_r)$. Let R_1 be the resulting extended regular expression. The proof that R_1 denotes $h^{-1}(S)$ follows easily by induction on the number of occurrences of operators $+$, $\cdot$, $*$, $\cap$, and $\neg$ in R_2. □

We must now talk about representations of sequences of instantaneous descriptions of Turing machines much as we did in Lemma 10.2. That is, given that $M = (Q, T, I, \delta, \text{b}, q_0, q_f)$ is a Turing machine, we represent an ID by a sequence of symbols in T plus one other symbol of the form $[qX]$, for $q \in Q$ and $X \in T$, indicating the state and input head position. If necessary we shall pad out an ID with blanks so that all ID's in the same computation are of the same length.

To help us compare symbols in one ID with "corresponding" symbols in successive ID's, we shall superimpose a yardstick, of the length chosen to represent ID's, upon the ID's themselves. Formally, we use a "two-track" string of symbols, where the upper track is a sequence of ID's and the lower track contains the yardstick repeated. That is, the "two-track" symbols are pairs $[a, b]$, where a is the symbol in the upper track and b the symbol in the lower track. The arrangement is shown in Fig. 11.1, where the yardstick used is CYCLE(x#). (x can be any string not containing #.)

Definition. Given TM M, let $\Delta_1 = T \cup (Q \times T) \cup \{\#\}$ and let Δ_2 be the set of symbols appearing in $x\#$. That is, Δ_1 is the set of symbols that can appear on the upper track and Δ_2 the set of symbols that can appear on the lower track. The "two-track" alphabet is $\Delta_1 \times \Delta_2$. A *valid computa-*

Upper track	#	C_0	#	C_1	#	$\cdots$	#	C_f	#
Lower track	#	x	#	x	#	$\cdots$	#	x	#

Fig. 11.1. Sequence of ID's with yardstick.

tion of M with yardstick CYCLE(x#) *is a string in* $(\Delta_1 \times \Delta_2)^*$ of the form shown in Fig. 11.1, where $C_i \vdash C_{i+1}$ by one move of M for $0 \leq i < f$, C_0 is an initial ID, and in the last ID C_f the state is q_f. The ID's have trailing blanks to make them all of length $|x|$.

Lemma 11.4. Let R_1 be an extended regular expression for CYCLE(x#) and let R_1' be a regular expression over the alphabet of x# denoting all strings but x. An extended regular expression R_2 denoting the invalid computations of M with yardstick CYCLE(x#) can be constructed such that the length of R_2 is linear in $|R_1| + |R_1'|$, the constant of proportionality depending only on M. R_2 contains a complementation sign only if R_1 or R_1' contains a complementation sign.

Proof. A string is not a valid computation with yardstick CYCLE(x#) if and only if either

1. the lower track is not in #(x#)*, or
2. the lower track is in #(x#)* but the upper track is not a valid computation.

If Δ_1 is the alphabet of the upper track and Δ_2 the alphabet of the lower track, let h_1 and h_2 be the homomorphisms mapping $(\Delta_1 \times \Delta_2)^*$ into Δ_1^* and Δ_2^*, respectively, such that $h_1([a, b]) = a$ and $h_2([a, b]) = b$. Then

$$h_2^{-1}[\Delta_2^*\#(R_1' \cap (\Delta_2 - \#)^*)\#\Delta_2^* + (\Delta_2 - \#)\Delta_2^* + \Delta_2^*(\Delta_2 - \#)] + \epsilon$$

denotes all and only the strings whose lower track is not in #(x#)*. The first term of the argument of h_2^{-1} denotes strings with something other than x between two #'s, and the remaining terms denote strings which do not begin and end with #. By Lemma 11.3, there is an extended regular expression of length proportional to $|R_1'|$ that denotes this set.

A given string satisfies condition 2 because two symbols separated by a number of symbols equal to the length of the yardstick do not reflect a move of M, or because one or more of the following format errors occur.

i) The string does not begin and end with # on the upper track.
ii) No state, or else two or more states, appear in the first ID.
iii) The first ID does not contain the initial state as a component of the first symbol.
iv) The accepting state does not appear as a component of any symbol.

v) The first ID is not of the correct length. That is, the first two #'s on the upper track are not in the same cells as the first two #'s on the lower track.

We shall now show how to write the extended regular expression for the sequences in $(\Delta_1 \times \Delta_2)^*$ which fail at some point to reflect legal moves of M. As in Lemma 10.2, we note that in a valid computation three consecutive symbols $c_1c_2c_3$ on the upper track uniquely determine the symbol $|x\#|$ places to the right of c_2. Let this symbol be $f(c_1c_2c_3)$ as before. Let

$$R_{c_1c_2c_3} = (\Delta_1 \times \Delta_2)^*[h_1^{-1}(c_1c_2c_3\Delta_1^*) \cap h_2^{-1}(R_1)] \\ [h_1^{-1}(\Delta_1(\Delta_1 - f(c_1c_2c_3))\Delta_1)](\Delta_1 \times \Delta_2)^* \quad (11.3)$$

Then, let

$$R = \sum_{c_1c_2c_3} R_{c_1c_2c_3}.$$

Note that $h_1^{-1}(c_1c_2c_3\Delta_1^*)$ denotes all strings in $(\Delta_1 \times \Delta_2)^*$ whose upper track begins with $c_1c_2c_3$. $h_2^{-1}(R_1)$ denotes those strings of length $|x\#|$ whose lower track is correct. Thus their intersection is all strings of length $|x\#|$ whose lower track contains a string in CYCLE$(x\#)$ and which begin with $c_1c_2c_3$ on the upper track. With these observations, it should be clear that R includes all strings which satisfy condition 2 because a move illegal for M is made at some point. Also included will be some strings which do not have a lower track in $\#(x\#)^*$; these also satisfy condition 1, and their presence or absence in R is immaterial. Moreover, the length of R is a constant (depending on M) times the length of R_1. To see the truth of this statement, it suffices to note that h_2^{-1} expands regular expressions by a factor depending on M. Also, h_1^{-1} expands expressions by a factor of $3\|\Delta_2\|$ at most, since if $h_1(b) = a$ for exactly k values of b, then $h_1^{-1}(a) = (b_1 + b_2 + \cdots + b_k)$ has length $2k + 1$. Since $1 \leq k \leq \|\Delta_2\|$ is obvious, $|h_1^{-1}(a)| \leq 3\|\Delta_2\|$ follows. Moreover, it should be clear that $\|\Delta_2\| \leq |R_1|$, since each symbol of Δ_2 appears in R_1. Hence the length of the terms $h_1^{-1}(c_1c_2c_3\Delta_1^*)$ and $h_1^{-1}(\Delta_1(\Delta_1 - f(c_1c_2c_3))\Delta_1)$ in (11.3) is $O(|R_1|)$. Finally, the terms for $h_2^{-1}(R_1)$ and $(\Delta_1 \times \Delta_2)^*$ are clearly $O(|R_1|)$, the constant dependent only on M. Thus the length of (11.3), and therefore the length of R, is $O(|R_1| + |R_1'|)$.

The regular expressions for the format errors are easy to write and are left to the reader. In this manner we can construct an extended regular expression R_2 such that the length of R_2 is linear in $|R_1| + |R_1'|$, and R_2 contains a complementation sign only if R_1 or R_1' does. □

Theorem 11.2. Any algorithm to determine whether a semiextended regular expression† denotes all strings over its alphabet is at least of space

† An encoding similar to that in Lemma 10.3 is assumed.

(and hence time) complexity $c'c^{\sqrt{n/\log n}}$ for some constants $c' > 0$, $c > 1$, and an infinity of values of n.

Proof. Let L be an arbitrary language of space complexity 2^n but not of space complexity $2^n/n$.† (By Theorem 11.1 we know that such a language exists.) Let M be a DTM that accepts L.

Suppose we have a DTM M_0 of space complexity $f(n)$ to decide whether the complement of a set denoted by a semiextended regular expression is empty. Then we can use M_0 to recognize the language L as follows. Let $w = a_1a_2 \cdots a_n$ be an input string of length n.

1. Construct a semiextended regular expression R_1 for a yardstick of length $2^n + 1$ or more. By Lemma 11.1 (with $k = n$), there is an expression R_1 of length $O(n^2)$. Moreover, R_1 can be found using only $O(n^2)$ space. Similarly, construct R_1' denoting $\neg x$, where $R_1 = \text{CYCLE}(x\#)$. By Lemma 11.2, R_1' has length $O(n^2)$ and R_1' is easily seen to be constructible using $O(n^2)$ space.
2. Construct a semiextended regular expression R_2 to denote the invalid computations of M with yardstick R_1. By Lemma 11.4, there is an R_2 of length at most c_1n^2, for some constant c_1 depending only on M.
3. Construct a regular expression R_3 to denote all strings in $(\Delta_1 \times \Delta_2)^*$ that do not begin with $\#[q_0a_1]a_2 \cdots a_n\text{b} \cdots \text{b}\#$ in the upper track, where q_0 is the initial state of M. An expression R_3 of length $O(n)$ clearly exists. Thus $|R_2 + R_3| \leq c_2n^2$, for some constant c_2.
4. Apply M_0 to $R_2 + R_3$ to decide whether the complement of $R_2 + R_3$ is empty. If not, there is a valid computation of M with input w, so w is in L. Otherwise, w is not in L.

We can construct a DTM M' of space complexity $f(c_2n^2 \log n)$ to implement this algorithm to recognize L. The factor of $\log n$ in the argument of f is due to the fact that the regular expression $R_2 + R_3$ is over an n-symbol alphabet and thus must be encoded into a fixed alphabet. Since we assume L is of space complexity 2^n but not of space complexity $2^n/n$, we must have $f(c_2n^2 \log n) > 2^n/n$ for at least some values of n. But if $f(c_2n^2 \log n)$ exceeded $2^n/n$ for only a finite number of n's, then there would exist a modified version of the above recognition algorithm which first checks by a finite "table lookup" whether $|w|$ was one of those n's for which $f(c_2n^2 \log n) > 2^n/n$, and if so, whether w was in L. Thus $f(c_2n^2 \log n)$ must exceed $2^n/n$ for an infinity of n's, so $f(m) \geq 2^{c_3\sqrt{m/\log m}}/\sqrt{m/\log m} \geq c(c')^{\sqrt{m/\log m}}$ for an infinity of m's and some constants $c_3 > 0$, $c > 0$, and $c' > 1$. □

† The choice of 2^n is not essential. We could replace 2^n by any exponential function $f(n)$ and $2^n/n$ by any function which grows slightly more slowly than $f(n)$, provided each was space-constructible.

Corollary. The equivalence problem for semiextended regular expressions requires $c'c^{\sqrt{n/\log n}}$ space and time for some constants $c' > 0$ and $c > 1$, and an infinity of n's.

Proof. It is easy to show that the emptiness-of-complement problem is polynomially reducible to the equivalence problem, since a regular expression denoting all strings is short and easy to write down. □

11.4 A NONELEMENTARY PROBLEM

We shall now consider the full class of extended regular expressions. Since we now have the complementation operator, we need only consider the emptiness problem, i.e., given an extended regular expression R, does R denote the empty set? We shall see that with the complementation operator we can represent regular sets even more concisely and that the emptiness problem for extended regular expressions is considerably harder than the emptiness-of-complement problem for semiextended regular expressions.

Let us define the function $g(m, n)$ by:

1. $g(0, n) = n$,
2. $g(m, n) = 2^{g(m-1,n)}$ for $m > 0$.

Thus $g(1, n)$ is 2^n, $g(2, n) = 2^{2^n}$, and

$$g(m, n) = 2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}},$$

a stack of m 2's with the last 2 raised to the nth power. A function $f(n)$ is *elementary* if it is bounded above for all but a finite set of n's by $g(m_0, n)$ for some fixed m_0.

The techniques of Section 11.3 can also be used to show that there is no elementary function $S(n)$ for which the emptiness problem for the full class of extended regular expressions is of space complexity $S(n)$. Before proceeding, we slightly change the definition of a valid computation of a Turing machine $M = (Q, T, I, \delta, \mathrm{b}, q_0, q_f)$ with yardstick CYCLE($x\#$). We delete the first marker # and change the last marker to a new symbol, \$, as shown in Fig. 11.2. The C's are ID's (padded out, if necessary, to be of the same length as x); $C_i \vdash C_{i+1}$ by one move of M, for $0 \leq i < f$, C_0 is an initial ID, and in C_f the state is q_f.

We shall use the following notational conventions. We assume $x \in \Sigma^*$.

1. $\Delta_1 = T \cup (Q \times T) \cup \{\#, \$\}$ is the upper track alphabet.
2. $\Delta_2 = \Sigma \cup \{\#, \$\}$ is the lower track alphabet.

Upper track	C_0	#	C_1	#	$\cdots$	#	C_f	\$
Lower track	x	#	x	#	$\cdots$	#	x	\$

Fig. 11.2. New format for valid computations with yardstick.

3. h_1: $\Delta_1 \times \Delta_2 \to \Delta_1$, where $h_1([a, b]) = a$.
4. h_2: $\Delta_1 \times \Delta_2 \to \Delta_2$, where $h_2([a, b]) = b$.

Lemma 11.5. Let R_1 be an extended regular expression for CYCLE(x#). We may construct an extended regular expression R_2 denoting the set of cyclic permutations of valid computations of a Turing machine M with yardstick CYCLE(x#), such that $|R_2|$ is $O(|R_1|)$, the constant depending only on M.

Proof. A string is a cyclic permutation of a valid computation with yardstick CYCLE(x#) if and only if

1. the lower track is a cyclic permutation of a string of the form $(x\#)^*x\$$,
2. the upper track is a cyclic permutation of a valid computation of M, and
3. the upper and lower tracks have been cyclically permuted by the same amounts.

Let R'_1 be R_1 with \$ substituted for #. Strings satisfying condition 1 can be denoted by $h_2^{-1}(U_1 \cap U_2 \cap U_3)$, where

$$U_1 = \Sigma^*(\# + \$)[(R_1 + R'_1) \cap (\Sigma^*\# + \Sigma^*\$)]^*\Sigma^*,$$
$$U_2 = (\Sigma + \#)^*\$(\Sigma + \#)^*,$$
$$U_3 = (R_1 + R'_1)^*.$$

The subexpression $[(R_1 + R'_1) \cap (\Sigma^*\# + \Sigma^*\$)]$ in U_1 denotes the two strings $x\#$ and $x\$$. Thus U_1 denotes $\Sigma^*(\# + \$)(x\# + x\$)^*\Sigma^*$. The expression U_2 allows only strings with one \$. Thus any string in $U_1 \cap U_2$ is of the form

$$y_2\#x\#x\# \cdots x\#x\$x\# \cdots x\#y_1$$

for some y_1 and y_2 in Σ^*. The expression U_3 forces $|y_2| + |y_1| = |x|$, from which we may easily show $y_1y_2 = x$ for strings in $U_1 \cap U_2 \cap U_3$. Hence $S_1 = U_1 \cap U_2 \cap U_3$ denotes the lower track of all strings satisfying condition 1.

For condition 2, we saw in Lemma 11.4 how to write a semiextended regular expression for the invalid computations of M with yardstick CYCLE(x#). These techniques carry over easily to the format of Fig. 11.2, and an expression E denoting cyclic permutations of invalid computations of M with a correct yardstick on the lower track is left for the reader to construct. Applying the complementation operator to E gives us an extended regular expression S_2. This is the only point at which complementation is used. It should be clear that all strings denoted by $S_1 \cap S_2$ satisfy both conditions 1 and 2.

The expression for condition 3, given that conditions 1 and 2 are satisfied, is easy. One need only check that one symbol has \$ in both tracks. Intersecting this expression with $S_1 \cap S_2$ completes the proof. □

We now proceed to show how longer and longer yardsticks can be expressed with little increase in length, using the construction of Lemma 11.5.

Intuitively, we construct a regular expression for some small-length yardstick, say length n. We use the yardstick to construct a new regular expression for all cyclic permutations of valid computations, with respect to this yardstick, of some Turing machine which accepts exactly one string of length n. The Turing machine is carefully selected so that it makes at least 2^{n+2} moves on the input of length n, and hence the set of cyclic permutations of its valid computations is itself a yardstick of length 2^n or more. By repeating the process with the new yardstick, we create yardsticks of longer and longer lengths, at least 2^n, 2^{2^n}, and so on.

Let M_0 be the particular single-tape Turing machine which behaves as follows.

1. M_0 checks that its input tape begins with a sequence of a's by scanning its tape until it reaches the first blank.
2. If the input tape consists of a string of m a's followed by a blank, then M_0 counts in binary from 0 to $2^{m+1} - 1$ on the portion of its input tape occupied by the a's and the following blank.
3. When M_0 has counted to $2^{m+1} - 1$, it halts and accepts.

We observe the following about M_0. For each n, M_0 accepts exactly one input string of length n, namely a^n. It does so with a unique valid computation consisting of at least 2^{n+2} moves since half its moves add bits and half handle carries. Finally, M_0 scans only $n + 1$ tape cells when presented with an input of length n.

Therefore, consider the valid computations of M_0 with yardstick CYCLE(x#), as in Fig. 11.2. If $|x| = n + 1$, then there will be a unique valid computation whose upper track begins with $[q_0a]a \cdots ab\#$. By Lemma 11.5 we can construct a regular expression R_2 for all cyclic permutations of the valid computation of M_0 with yardstick CYCLE(x#). The expression R_2 denotes a set consisting of the cyclic permutations of a fixed string w in $(\Delta_1 \times \Delta_2)^*$. The string $h_1(w)$ is the valid computation of M_0 on input a^n, where $|x|$, we recall, is $n + 1$. Since M_0 makes at least 2^{n+2} moves on input a^n, we may use R_2 itself as a yardstick of length at least 2^{n+2}.

In summary, suppose we are given a yardstick $R_1 = \text{CYCLE}(x\#)$, where $x \in \Sigma^*$. Then we can construct from R_1 and the particular DTM M_0 a yardstick of length at least $2^{|x\#|}$. By Lemma 11.5 there exists for this yardstick an extended regular expression R_2 of length at most $c_1|R_1|$, where c_1 depends only on M_0.

Lemma 11.6. For all $i \geq 1$ and $m \geq 2$ there is a yardstick of length at least $g(i, m)$† which can be denoted by an extended regular expression of length at most $c(c_1)^{i-1}m^2$, where c_1 is the constant depending on M_0, introduced in the paragraph above, and c is the constant of Lemma 11.1.

Proof. We show, by induction on i, that we can find an extended regular expression R_1 denoting CYCLE($x\#$) for some x, where $|x\#| \geq g(i, m)$ and $|R_1| \leq c(c_1)^{i-1}m^2$. The basis, $i = 1$, follows from Lemma 11.1, with $k = m$, since an extended regular expression of length cm^2 denoting CYCLE($x\#$) can be constructed.

For the inductive step, suppose we have an extended regular expression R_1, where $|R_1| \leq c(c_1)^{i-2}m^2$, R_1 denotes CYCLE($x\#$), and $|x\#| \geq g(i-1, m)$. Then by the above analysis concerning the DTM M_0, we may construct a regular expression R of length at most $c_1|R_1| = c(c_1)^{i-1}m^2$. R denotes CYCLE($y\$$), where $|y\$| \geq 2^{|x\#|} \geq 2^{g(i-1,m)} = g(i, m)$. □

Theorem 11.3. Let $S(n)$ be any elementary function. Then there is no $S(n)$-space-bounded [hence no $S(n)$-time-bounded] DTM to decide whether an extended regular expression denotes the empty set.

Proof. Suppose in contradiction that there is a $g(k_0, n)$-space-bounded TM M_1 which can decide whether an extended regular expression denotes the empty set. By Theorem 11.1, there is a language L accepted by a $g(k_0 + 1, n)$-space-bounded DTM M but by no $[g(k_0 + 1, n)/n]$-space-bounded DTM. On the assumption that M_1 exists, we could design a DTM M_2 to recognize L which behaves as follows.

1. Given an input string $w = a_1a_2 \cdots a_n$ of length n, M_2 constructs an extended regular expression R_1 of length d_1n^2, where d_1 is a constant independent of n, denoting a yardstick of length at least $g(k_0 + 1, n)$. The algorithm implied by Lemma 11.6 is used to construct R_1.
2. Next, M_2 constructs an extended regular expression R_2 denoting the set of valid computations with yardstick R_1 of the $g(k_0 + 1, n)$-space-bounded TM M which accepts L. We may make $|R_2| \leq d_2|R_1|$, for some constant d_2, by Lemma 11.5.
3. Then, M_2 constructs R_3, an extended regular expression denoting the valid computations of M with yardstick R_1 and the initial ID $C_0 = [q_0a_1]a_2 \cdots a_n$bb . . . , where q_0 is the initial state of M. That is,

$$R_3 = R_2 \cap h_1^{-1}([q_0a_1]a_2 \cdots a_n\text{b}^*\#)(\Delta_1 \times \Delta_2)^*,$$

where h_1 is the homomorphism from Lemma 11.5 and $\Delta_1 \times \Delta_2$ is the

† g is defined on p. 419.

alphabet of R_2. By Lemma 11.3, $|R_3| \le |R_2| + d_3 n$ for some constant $d_3 > 0$. Thus

$$|R_3| \le d_1 d_2 n^2 + d_3 n. \tag{11.4}$$

4. Finally, M_2 encodes R_3 into a fixed alphabet as in Theorem 11.2 and uses M_1 to test whether the extended regular expression R_3 denotes the empty set. If so, M_2 rejects w, and if not, M_2 accepts w. Thus M_2 accepts L.

Now it is not hard to see that the most space-consuming step is step 4, in which M_1 requires space $g(k_0, |R_3| \log |R_3|)$ to determine whether R_3 denotes the empty set. Thus M_2 requires this amoont of space as well. Using (11.4) to bound $|R_3|$, we see that M_2 is of space complexity $S(n) = d_4 g(k_0, d_5 n^2 \log n)$ for some constants d_4 and d_5, since for all but a finite number of n, the first term of (11.4), namely $d_1 d_2 n^2$, is greater than the second, $d_3 n$. However, by the argument used in Theorem 11.2, we may show that the space complexity of M_2 must exceed $g(k_0 + 1, n)/n$ for an infinity of n's. Thus, if M_2 exists, we have

$$g(k_0 + 1, n)/n < d_4 g(k_0, d_5 n^2 \log n) \tag{11.5}$$

for an infinite number of values for n. But no matter what values d_4 and d_5 take, there can be only a finite number of n's for which (11.5) holds, as the reader may easily show. We conclude that M_2 does not exist, so M_1 does not exist. Therefore the emptiness problem for extended regular expressions is not decidable by any elementary space-bounded Turing machine. □

EXERCISES

11.1 A function $T(n)$ is said to be *time-constructible* if there is a DTM M which given an input of length n makes exactly $T(n)$ moves before halting. Show that the following are time-constructible functions.

a) n^2

b) 2^n

c) $n!$

***11.2** Show that every time-constructible function is space-constructible and that if $S(n)$ is space-constructible then $c^{S(n)}$ is time-constructible for some integer c.

***11.3** Show that if L is accepted in time $T(n)$ by a k-tape DTM (NDTM), then there is a single-tape DTM (NDTM) of time complexity $O(T^2(n))$ accepting L.

***11.4** (*Time hierarchy for DTM's.*) Show that if $T_1(n)$ and $T_2(n)$ are time-constructible functions and

$$\inf_{n \to \infty} \frac{T_1^2(n)}{T_2(n)} = 0,$$

then there is some language accepted in time $T_2(n)$ but not $T_1(n)$ by a DTM. [*Hint:* By using Exercise 11.3, you need only diagonalize over single-tape

DTM's of time complexity $[T_1(n)]^2$. The diagonalization can be carried out by a multitape DTM.]

The next two exercises are weak forms of hierarchy results for nondeterministic Turing machines.

**11.5 Show that for each integer $k \geq 1$, there is some language accepted by an NDTM of space complexity n^{k+1} but by no NDTM of space complexity n^k.

**11.6 Show the same result as Exercise 11.5 for time complexity.

We tighten the time hierarchy for DTM's in the following two exercises.

**11.7 Show that every language L accepted by a k-tape DTM of time complexity $T(n)$ is accepted by a two-tape DTM of time complexity $O(T(n) \log T(n))$.

11.8 Use Exercise 11.7 to show that if $T_1(n)$ and $T_2(n)$ are time-constructible functions and

$$\inf_{n \to \infty} \frac{T_1(n) \log T_1(n)}{T_2(n)} = 0,$$

then there is some language accepted in time $T_2(n)$ but not in time $T_1(n)$ by a DTM.

*11.9 Show that if L is accepted by a DTM (NDTM) M of space complexity $S(n)$ and time complexity $T(n)$, and c is any constant greater than 0, then L is accepted by a DTM (NDTM) M' of space complexity $\text{MAX}(cS(n), n + 1)$ and time complexity $\text{MAX}(cT(n), 2n)$. [*Hint:* M' must begin by condensing blocks of cells of M to single cells of its own tape.]

11.10 Show that if L is accepted by an NDTM of time complexity $T(n)$, then there is a constant c such that L is accepted by a DTM of time complexity $c^{T(n)}$.

*11.11 Let T_1 and T_2 be functions such that

$$\inf_{n \to \infty} \frac{T_1(n)}{T_2(n)} = 0.$$

Show that there is a language accepted by a RAM in time $T_2(n)$ but not $T_1(n)$, where times are taken under the logarithmic cost function.

11.12 Complete the proof of Lemma 11.2.

11.13 Given an extended regular expression R_1 for CYCLE($x\#$), construct an extended regular expression R_2 for CYCLE(($x\#$)).† The length of R_2 should be bounded by a constant times the length of R_1.

**11.14 Give an $O(n)$ space-bounded nondeterministic algorithm for determining whether a semiextended regular expression denotes a nonempty set. Why does your algorithm break down if you try to determine whether the regular expression denotes all strings? Why must it break down?

11.15 Give an exponential time deterministic algorithm to solve the emptiness-of-complement problem for semiextended regular expressions.

† The meaning of CYCLE applied to a set of strings is the union of CYCLE applied to each string individually.

11.16 Write the regular expressions for the "format errors" in Lemma 11.4.

11.17 Is the function $F(n)$, defined as $F(0) = 1$ and $F(n) = 2^{F(n-1)}$ for $n \geq 1$, elementary? (This function was introduced in Section 4.7.)

***11.18** Give an algorithm to solve the problem of whether an extended regular expression denotes the empty set. What is the time and space complexity of your algorithm?

****11.19** Show that the emptiness problem for extended regular expressions using only the operators $+$, $\cdot$, and $\neg$ is not elementary.

Research Problems

11.20 The natural research area suggested by this chapter is to find problems of practical significance that can be proven intractable. Work in this direction has been done by Fischer and Rabin [1974] on the complexity of elementary arithmetic, Cook and Reckhow [1974] on theorem proving, and Hunt [1974] on language theoretic problems, in addition to several other authors cited in the bibliographic notes.

11.21 Another area of interest is the question of whether the time hierarchy for DTM's and the time and space hierarchies for NDTM's are actually tighter than implied by Exercises 11.5, 11.6, and 11.8. For example, are there time-constructible $T(n)$'s for which no languages are recognized in $T(n) \log T(n)$ that are not recognized in $T(n)$ time? The reader should consult Seiferas, Fischer, and Meyer [1973] for the tightest known hierarchy for NDTM's.

BIBLIOGRAPHIC NOTES

The first extensive studies of space and time hierarchies for Turing machines were in Hartmanis and Stearns [1965] and Hartmanis, Lewis, and Stearns [1965]. Rabin [1963] is an early paper on time complexity and is worth study. The improvement in the time hierarchy given in Exercises 11.7 and 11.8 is from Hennie and Stearns [1966]. For the nondeterministic hierarchies, Exercise 11.5 (space) is by Ibarra [1972] and 11.6 (time) from Cook [1973]. The RAM hierarchy, Exercise 11.11, is by Cook and Reckhow [1973]. Exercise 11.14 is by J. Hopcroft. Exercise 11.19 is from Meyer and Stockmeyer [1973].

The work on exponential lower bounds for "natural" problems began with Meyer [1972] and Meyer and Stockmeyer [1972]. Theorem 11.2 on semiextended regular expressions is from Hunt [1973b]. Theorem 11.3 on extended regular expressions is by Meyer and Stockmeyer [1973]. Other work on intractable problems can be found in Book [1972], Hunt [1973a, 1974], Stockmeyer and Meyer [1973], Meyer [1972], Hunt and Rosenkrantz [1974], Rounds [1973], and Constable, Hunt, and Sahni [1974].

LOWER BOUNDS ON NUMBERS OF ARITHMETIC OPERATIONS

CHAPTER 12

In designing an algorithm to solve a given problem the most basic question to which we would like an answer is, "What is the inherent computational complexity of the given problem?" Knowing the theoretical lower limit on how efficient an algorithm can be, we can better evaluate proposed algorithms and help determine how much further effort should be expended in trying to find a better solution. For example, if a problem is known to be intractable, then we might be content to use heuristic techniques to find approximate solutions.

Unfortunately, determining the inherent computational complexity of a problem is usually a very difficult task. For most practical problems we have to rely on experience to judge the goodness of an algorithm. However, in some cases we can tightly bound from below the number of arithmetic operations required to perform certain calculations. In this chapter we shall present some basic results of this nature. For example, we shall show that the multiplication of an $n \times p$ matrix by a p-vector requires np scalar multiplications, and that the evaluation of an nth-degree polynomial requires n multiplications. Many additional results on lower bounds are contained in the exercises. The reader interested in lower bounds should treat the exercises as an integral part of this chapter.

12.1 FIELDS

In order to obtain precise lower bounds we must know what basic operations are permitted. For specificity we shall assume that all computations are done in a field, such as the field of real numbers, where the basic operations are addition and multiplication of field elements.

Definition. A *field* is an algebraic system $(A, +, \cdot, 0, 1)$ such that

1. the system is a ring with multiplicative identity 1,
2. multiplication is commutative, and
3. each element a in $A - \{0\}$ has a multiplicative inverse a^{-1} such that $aa^{-1} = a^{-1}a = 1$.[†]

Example 12.1. The real numbers with the familiar operations of addition and multiplication form a field. The integers form a ring, but not a field since integers other than ± 1 do not have multiplicative inverses that are integers. However, for p a prime, the integers modulo p do form a (finite) field. □

Consider the problem of evaluating an arbitrary polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ for some value of x. What we want is an algorithm that takes values for the a_i's and x as inputs and produces the corresponding value of $p(x)$ as output. The algorithm is to work for all possible values of its inputs taken from some field. The maximum number of additions, subtractions, and multi-

[†] As is customary, we omit $\cdot$ when no confusion arises.

plications performed by the algorithm, taken over all permissible inputs, is said to be the *arithmetic complexity* of the algorithm.

Note that certain nth-degree polynomials appear to be much easier to evaluate than others. For example, the evaluation of $x^n + 2$ takes on the order of log n operations, whereas intuitively we feel that the evaluation of a "random" nth-degree polynomial would require a linear number of operations. Thus an evaluation algorithm which works only for a specific polynomial may be much faster than one that works for all polynomials. To capture the notion of an algorithm which works for an entire class of problems, we introduce indeterminates to represent input variables.

> **Definition.** An *indeterminate* over an algebraic system is a symbol not in the underlying set. Let $F = (A, +, \cdot, 0, 1)$ be a field and let $x_1, \ldots, x_n$ be indeterminates over F. The *extension* of F by the indeterminates $x_1, \ldots, x_n$, denoted $F[x_1, \ldots, x_n]$, is the smallest commutative† ring $(B, +, \cdot, 0, 1)$ such that B contains $A \cup \{x_1, \ldots, x_n\}$.

Note that there are no "hidden" identities involving indeterminates. Thus every polynomial with coefficients in F and "unknowns" $x_1, x_2, \ldots, x_n$ represents some element of $F[x_1, \ldots, x_n]$. Two polynomials denote the same element of $F[x_1, \ldots, x_n]$ if one can be transformed into the other by use of the laws of a commutative ring. The multiplicative identity 1 of F will also be a multiplicative identity in $F[x_1, \ldots, x_n]$.

Example 12.2. Let F be the field of reals. Then the ring $F[x, y]$ includes x, y, and all real numbers. Since $F[x, y]$ is closed under $+$, we see that $x + y$ and $x + 4$ are in $F[x, y]$. Since $F[x, y]$ is closed under multiplication, it contains $(x + y)(x + 4)$, which is equivalent to $x^2 + xy + 4x + 4y$ under the distributive law of the ring. □

12.2 STRAIGHT-LINE CODE REVISITED

Consider again the question "How many arithmetic operations are required to evaluate an arbitrary polynomial?" We are really asking a question concerning the number of operations $+$ and $\cdot$ needed to construct the expression $\Sigma_{i=0}^{n} a_i x^i$ or one equivalent to it from the indeterminates $a_0, \ldots, a_n$ and x. This observation motivates the following model of computation which is essentially the straight-line program model of Section 1.5.

> **Definition.** Let F be a field. A *computation* with respect to F consists of the following.
>
> 1. A set of indeterminates called *inputs*.
> 2. A set of *variable names*.

† That is, multiplication is commutative.

3. A sequence of steps of the form $a \leftarrow b\ \theta\ c$, where θ is $+, -,$ or $*$, a is a variable not appearing in any previous step. and b and c are eithe inputs, elements of F, or variable names appearing on the left of th arrow at some previous step.

For convenience we write $a \leftarrow b$ for $a \leftarrow b + 0$ and $a \leftarrow -b$ for $a \leftarrow 0 - b$ An element of F appearing in a computation is called a *constant.*

To determine the result of a computation, we need the notion of a *valuation* of a variable in a computation. Informally, we consider a computation t take place step by step; each step assigns an element of $F[x_1, \ldots, x_n]$, wher the x_i's are the inputs, to a new variable. We define the *value* $v(a)$ of variabl or input a as follows. If a is an input or an element of F, then $v(a) = a$. If a i a variable and $a \leftarrow b\ \theta\ c$ is the step with a on the left, then $v(a) = v(b)\ \theta\ v(c)$

A computation *computes E,* a set of expressions in $F[x_1, \ldots, x_n]$, wit respect to the field F, if for each expression e in E there is some variable f i the computation such that $v(f) = e$.

Observe that a computation is with respect to a given underlying fiel For example, computing $x^2 + y^2$, where F is the field of reals, requires tw multiplications even if multiplication by a constant is not counted. Howeve if F is the field of complex numbers, then only one multiplication (excludin multiplications by a constant) is required, namely $(x + iy)(x - iy)$. The under lying field is usually taken to be the field of reals, although we might use th complex numbers, the rational numbers, or some finite field. Which field w use depends on the operations we take as primitive. If we assume that we ca represent real numbers and perform addition and multiplication on reals a primitive operations, then F is taken to be the reals.

Example 12.3. Recall that the computation of the product of two comple numbers $a + ib$ and $c + id$ with respect to the field of reals can be viewed a the computation of the expressions $ac - bd$ and $ad + bc$. The obvious com putation is

$$\begin{aligned}
&f_1 \leftarrow a*c\\
&f_2 \leftarrow b*d\\
&f_3 \leftarrow f_1 - f_2\\
&f_4 \leftarrow a*d\\
&f_5 \leftarrow b*c\\
&f_6 \leftarrow f_4 + f_5
\end{aligned}$$

The value of f_1 is ac. Similarly, $v(f_2) = bd$ and $v(f_3) = ac - bd$. Thu the value of f_3 is equal to the first expression. The value of f_6 is $ad + bc$ an is equal to the second expression. Thus the computation computes the produ of two complex numbers.

There is another computation for complex multiplication which uses only three real multiplications:

$$\begin{aligned} f_1 &\leftarrow a+b \\ f_2 &\leftarrow f_1 * c \\ f_3 &\leftarrow d-c \\ f_4 &\leftarrow a * f_3 \\ f_5 &\leftarrow f_4 + f_2 \\ f_6 &\leftarrow d+c \\ f_7 &\leftarrow b * f_6 \\ f_8 &\leftarrow f_2 - f_7 \end{aligned}$$

It is easily shown that $v(f_5) = ad + bc$ and $v(f_8) = ac - bd$. □

It should be clear that a computation computes an expression if and only if substituting arbitrary values from the field F for inputs results in a computation of the specific instance of the expression obtained by substituting the input values for the indeterminates of the expression.

In the remainder of the chapter we are concerned primarily with lower bounds on the number of multiplications needed to compute a set of expressions. The reason for concentrating on multiplications is that in certain domains multiplications are arbitrarily more costly than additions and subtractions.

For example, we saw in Chapter 6 that because we can multiply two 2 × 2 matrices with seven scalar multiplications, we have an $O_A(n^{\log 7})$ matrix-multiplication algorithm. The fact that Strassen's algorithm uses 18 additions is asymptotically negligible. In the first level of recursion for Strassen's algorithm the seven $(n/2) \times (n/2)$ matrix multiplications are arbitrarily more costly than the 18 (or any other number of) additions and subtractions of matrices of the same size, as n approaches infinity.

In fact, if we could multiply two 2 × 2 matrices with six scalar multiplications using only the laws of a noncommutative ring, then we would have an $O_A(n^{\log 6}) = O_A(n^{2.59})$ matrix-multiplication algorithm, no matter how many additions or subtractions a 2 × 2 matrix multiplication took. (It can be shown, however, that if the algorithm is to work for an arbitrary ring, seven multiplications are needed for a 2 × 2 matrix multiplication; see Hopcroft and Kerr [1971].)

For another example, we saw in Section 2.6 that we could multiply two n-bit integers in $O_B(n^{1.59})$ steps because three multiplications were sufficient to compute the expressions ac, bd, and $ad + bc$. In fact if we could evaluate these expressions with a computation that had only two multiplications, then we would have $M(n) \leq 2M(n/2) + cn$ for some constant c. Such a computation, applied recursively, would yield an $O_B(n \log n)$ integer-multiplication algorithm, which is better than any known. Unfortunately, we shall show that in any field, these expressions cannot be computed with fewer than three multiplications.

12.3 A MATRIX FORMULATION OF PROBLEMS

Many problems can be formulated in terms of computing the product of a matrix and a column vector. The elements of the matrix are from $F[a_1, \ldots, a_n]$, where F is a field and $a_1, \ldots, a_n$ are indeterminates. The components of the column vector are indeterminates distinct from $a_1, \ldots, a_n$.

Example 12.4. The problem of multiplying two complex numbers $a + ib$ and $c + id$ can be couched as the matrix-vector product

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac - bd \\ bc + ad \end{bmatrix}.$$

Here F is the field of reals and the elements of the matrix are chosen from $F[a, b]$. The vector is composed of the indeterminates c and d. □

Example 12.5. Evaluation of the polynomial $\Sigma_{i=0}^{n} a_i x^i$ can be expressed as

$$[1, x, x^2, \ldots, x^n] \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{bmatrix}.$$

Here F is the field of reals and the elements of the $1 \times (n + 1)$ matrix are from $F[x]$. □

12.4 A ROW-ORIENTED LOWER BOUND ON MULTIPLICATIONS

Definition. Let F be a field and $a_1, \ldots, a_n$ indeterminates. Denote by $F^m[a_1, \ldots, a_n]$ the m-dimensional space of vectors with components from $F[a_1, \ldots, a_n]$. Let F^m be the m-dimensional space of vectors with components from F. A set of vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ from $F^m[a_1, \ldots, a_n]$ is *linearly independent modulo F^m* if for $u_1, \ldots, u_k$ in F, $\Sigma_{i=1}^{k} u_i \mathbf{v}_i$ in F^m implies the u_i are all zero. If the $\mathbf{v}_i$ are not linearly independent modulo F^m, then they are said to be *dependent modulo F^m*.

An alternative way to view linear independence modulo F^m is to consider the quotient space $F^m[a_1, \ldots, a_n]/F^m$ of equivalence classes of vectors in $F^m[a_1, \ldots, a_n]$. (Vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ in $F^m[a_1, \ldots, a_n]$ are *equivalent* if and only if $\mathbf{v}_1 - \mathbf{v}_2$ is in F^m.) Linear independence modulo F^m means linear independence of the equivalence classes of $F^m[a_1, \ldots, a_n]/F^m$.

Example 12.6. Let F be the field of reals and let a and b be indeterminates. Then the three vectors

$$\mathbf{v}_1 = \begin{bmatrix} 2a \\ 4b \\ 2a - 2b \end{bmatrix}, \qquad \mathbf{v}_2 = \begin{bmatrix} -a \\ -b \\ b + 3 \end{bmatrix}, \qquad \mathbf{v}_3 = \begin{bmatrix} \pi \\ b - 1 \\ a \end{bmatrix}$$

in $F^3[a, b]$ are dependent modulo F^3 since

$$\tfrac{1}{2}\mathbf{v}_1 + \mathbf{v}_2 - \mathbf{v}_3 = \begin{bmatrix} -\pi \\ 1 \\ 3 \end{bmatrix}$$

is in F^3.

On the other hand, the vectors

$$\mathbf{v}_1 = \begin{bmatrix} b \\ a \end{bmatrix}, \qquad \mathbf{v}_2 = \begin{bmatrix} a \\ 0 \end{bmatrix}, \qquad \mathbf{v}_3 = \begin{bmatrix} 0 \\ b \end{bmatrix}$$

are linearly independent modulo F^2. Suppose $\Sigma_{i=1}^{3} u_i \mathbf{v}_i$ is in F^2. Then $u_1 b + u_2 a$ is in F. Since neither a nor b is in F, we must have $u_1 = u_2 = 0$. In addition $u_1 a + u_3 b$ in F implies that $u_1 = u_3 = 0$. We conclude that $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ are linearly independent modulo F^2. □

If M is an $r \times p$ matrix with elements from F, then the number of linearly independent rows of M is equal to the number of linearly independent columns of M. However, if M has elements from $F[a_1, \ldots, a_n]$, then the number of linearly independent rows modulo F^p is not in general the same as the number of linearly independent columns of M modulo F^r. For example, the 1×3 matrix $[a_1, a_2, a_3]$ has one linearly independent row modulo F^3 and three linearly independent columns modulo F. The *row rank of M modulo F^p* is the number of linearly independent rows modulo F^p. The *column rank* of M modulo F^r is defined analogously.

For the remainder of this section, and for the next two sections, the following notational conventions are usually used.

1. F denotes a field,
2. $\{a_1, \ldots, a_n\}$ and $\{x_1, \ldots, x_p\}$ denote disjoint sets of indeterminates,
3. M is an $r \times p$ matrix with elements from $F[a_1, \ldots, a_n]$, and
4. $\mathbf{x}$ is the column vector $[x_1, \ldots, x_p]^T$.†

Theorem 12.1. Let M be a matrix with elements from $F[a_1, \ldots, a_n]$ and let $\mathbf{x}$ be the column vector $[x_1, \ldots, x_p]^T$. Assume that the row rank of M is r. Then any computation of $M\mathbf{x}$ requires at least r multiplication steps.

† Note that it is inconvenient to draw column vectors in column form, so we shall often write them in row form as in Chapter 7 and indicate transposition by the superscript T.

Proof. Assume without loss of generality that M has r rows. (Otherwise we could delete rows of M other than the r independent rows and let M' be the resulting matrix. Every computation of $M\mathbf{x}$ is a computation of $M'\mathbf{x}$. It follows that if $M'\mathbf{x}$ requires r multiplications, then $M\mathbf{x}$ requires r multiplications.)

Suppose there are s multiplications in some computation of $M\mathbf{x}$. Let $e_1, \ldots, e_s$ be the expressions computed at each of the multiplication steps. Then the elements of $M\mathbf{x}$ can be expressed as a linear function of the expressions $e_1, \ldots, e_s$ and the indeterminates. That is, we can write

$$M\mathbf{x} = N\mathbf{e} + \mathbf{f}, \tag{12.1}$$

where $\mathbf{e}$ is the vector whose ith component is e_i and $\mathbf{f}$ is a vector whose components are linear functions of the a_i's and x_i's alone. N is an $r \times s$ matrix whose elements are in F only.

Now suppose $r > s$. It is an elementary result of linear algebra that if $r > s$, the rows of N are linearly dependent.† That is, there exists $\mathbf{y} \neq \mathbf{0}$‡ in F^r such that $\mathbf{y}^T N = \mathbf{0}^T$. Multiplying (12.1) by $\mathbf{y}^T$ yields

$$\mathbf{y}^T M\mathbf{x} = \mathbf{y}^T N\mathbf{e} + \mathbf{y}^T\mathbf{f} = \mathbf{y}^T\mathbf{f}. \tag{12.2}$$

From (12.2) we see that $(\mathbf{y}^T M)\mathbf{x} = \mathbf{y}^T\mathbf{f}$, an expression which is a linear function of the indeterminates. Since $\mathbf{x}$ has a distinct indeterminate in each component, we conclude that the row vector $\mathbf{y}^T M$ must have only elements from F. Otherwise, there would be a term involving a product of indeterminates in $\mathbf{y}^T M\mathbf{x}$, and hence in $\mathbf{y}^T\mathbf{f}$, contrary to the assumption that $\mathbf{f}$ is linear in the indeterminates. Since $\mathbf{y} \neq \mathbf{0}$ and $\mathbf{y}^T M$ is in F^p, the rows of M are dependent modulo F^p, contrary to assumption. Thus $s \geq r$ and the theorem follows. □

Example 12.7. In Section 12.2 we considered the computation of three expressions, ac, bd, and $ad + bc$, for a simple recursive integer multiplication algorithm. The three formulas may be expressed by the matrix-vector product

$$\begin{bmatrix} a & 0 \\ 0 & b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}. \tag{12.3}$$

Let $\mathbf{R}$ be the field of real numbers. The three rows of the matrix in (12.3) were shown linearly independent modulo $\mathbf{R}^2$ in Example 12.6, and thus a computation of these three expressions with respect to the reals requires three real multiplications. □

† If the reader is not familiar with this result, the *basis theorem*, Lemma 12.1 (p. 435), is a harder result whose proof should provide sufficient insight to prove the present result.

‡ $\mathbf{0}$ is the all-zero vector of the appropriate dimension.

12.5 A COLUMN-ORIENTED LOWER BOUND ON MULTIPLICATIONS

The result analogous to Theorem 12.1 in terms of the number of linearly independent columns is true but somewhat harder to obtain. We need a preliminary result concerning sets of linearly independent vectors.

Lemma 12.1. Let $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ be a set of $k \geq 1$ m-vectors with components from $F[a_1, \ldots, a_n]$. Assuming $\{\mathbf{v}_i | 1 \leq i \leq k\}$ has a subset of q vectors which are linearly independent modulo F^m, then for any elements $b_2, \ldots, b_k$ in F, the set $\{\mathbf{v}'_i | \mathbf{v}'_i = \mathbf{v}_i + b_i\mathbf{v}_1, 2 \leq i \leq k\}$ contains a subset of $q - 1$ linearly independent vectors modulo F^m.

Proof. By renumbering $\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_k$, if necessary, we can assume that either $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_q\}$ or $\{\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$ is linearly independent.†

CASE 1. Suppose $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_q\}$ is linearly independent. Then we claim $\{\mathbf{v}'_2, \mathbf{v}'_3, \ldots, \mathbf{v}'_q\}$ is linearly independent. To see this, suppose $\Sigma_{i=2}^{q} c_i\mathbf{v}'_i$ is in F^m for some set of c_i's in F. Then $\Sigma_{i=1}^{q} c_i\mathbf{v}_i$ is in F^m where $c_1 = \Sigma_{i=2}^{q} c_ib_i$. But if $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_q\}$ is linearly independent, then $c_i = 0$ for $1 \leq i \leq q$. Thus, $\{\mathbf{v}'_2, \mathbf{v}'_3, \ldots, \mathbf{v}'_q\}$ is linearly independent.

CASE 2. Suppose $\{\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$ is linearly independent. If $\{\mathbf{v}'_2, \ldots, \mathbf{v}'_q\}$ is linearly independent, we have our result, so assume not. Then there exist $c_2, \ldots, c_q$ in F, not all 0, such that $\Sigma_{i=2}^{q} c_i\mathbf{v}'_i$ is in F^m. Let $c_1 = \Sigma_{i=2}^{q} c_ib_i$. Then $\mathbf{w} = \Sigma_{i=1}^{q} c_i\mathbf{v}_i$ is also in F^m.

We may assume $c_1 \neq 0$, for otherwise $\Sigma_{i=2}^{q} c_i\mathbf{v}_i$ would be in F^m, contradicting the assumption that $\{\mathbf{v}_2, \ldots, \mathbf{v}_{q+1}\}$ is linearly independent. We may thus write

$$\mathbf{v}_1 = c_1^{-1}\left(\mathbf{w} - \sum_{i=2}^{q} c_i\mathbf{v}_i\right).$$

Since not all of $c_2, \ldots, c_q$ are 0 we can assume without loss of generality that $c_2 \neq 0$. We now repeat the argument above with the set $\{\mathbf{v}'_3, \mathbf{v}'_4, \ldots, \mathbf{v}'_{q+1}\}$. If this set is linearly independent, we are done, so assume $\Sigma_{i=3}^{q+1} d_i\mathbf{v}'_i$ is in F^m, where the d_i's are in F and not all 0. Let $d_1 = \Sigma_{i=3}^{q+1} d_ib_i$. Then $\mathbf{x} = d_1\mathbf{v}_1 + \Sigma_{i=3}^{q+1} d_i\mathbf{v}_i$ is in F^m. If $d_1 = 0$, we contradict the linear independence of $\{\mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$. If $d_1 \neq 0$, we can write

$$\mathbf{v}_1 = d_1^{-1}\left(\mathbf{x} - \sum_{i=3}^{q+1} d_i\mathbf{v}_i\right).$$

Equating the two expressions for $\mathbf{v}_1$ we obtain

$$d_1^{-1}\left(\mathbf{x} - \sum_{i=3}^{q+1} d_i\mathbf{v}_i\right) = c_1^{-1}\left(\mathbf{w} - \sum_{i=2}^{q} c_i\mathbf{v}_i\right).$$

† We delete "modulo F^m" throughout this proof.

Multiplying by c_1d_1 and rearranging terms we obtain

$$d_1c_2\mathbf{v}_2 + \sum_{i=3}^{q} (d_1c_i - c_1d_i)\mathbf{v}_i - c_1d_{q+1}\mathbf{v}_{q+1} = d_1\mathbf{w} - c_1\mathbf{x}.$$

Since $\mathbf{w}$ and $\mathbf{x}$ are in F^m, $d_1\mathbf{w} - c_1\mathbf{x}$ is also in F^m. Since c_2 and d_1 are nonzero $\{\mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_{q+1}\}$ is linearly dependent, a contradiction. □

Let M be an $r \times p$ matrix with elements from $F[a_1, \ldots, a_n]$ as before. We now show that if q columns of M are linearly independent modulo F^r, $q \geq 1$, then any computation of $M\mathbf{x}$ requires at least q multiplications, where $\mathbf{x} = [x_1, \ldots, x_p]^T$.

Definition. A multiplication is said to be *active* if the value of one of the operands multiplied involves one of the indeterminates x_i and the other operand is not an element of F. For example, $b*c$ is an active multiplication if $v(b) = 3 + a_2$ and $v(c) = x_1 + 2*x_3$, but not if $v(b) = 3$ and $v(c) = x_1 + 2*x_3$ or if $v(b) = 3 + a_2$ and $v(c) = a_1 + 2*a_3$.

Theorem 12.2. Let $\mathbf{y}$ be a vector of dimension r with elements from $F[a_1, \ldots, a_n]$. If M has column rank q, then any computation of $M\mathbf{x} + \mathbf{y}$ has at least q active multiplications, for $q \geq 1$.

Proof. The proof proceeds by induction on q.

BASIS. $q = 1$. There exists a column of M not in F^r and hence an element e of M which is in $F[a_1, \ldots, a_n]$ but not in F. Thus some element of $M\mathbf{x}$ has a term with the product ex_j for some j. Thus so does $M\mathbf{x} + \mathbf{y}$. A computation with no active multiplications can compute only expressions of the form $P(a_1, \ldots, a_n) + L(x_1, \ldots, x_p)$, where P is a polynomial and L is a linear function, each with coefficients in F. Thus a computation of $M\mathbf{x} + \mathbf{y}$ must have at least one active multiplication.

INDUCTIVE STEP. Assume $q > 1$ and that the theorem is true for $q - 1$. Let C be a computation of $M\mathbf{x} + \mathbf{y}$. By the inductive hypothesis, C has at least $q - 1 \geq 1$ active multiplications. Suppose $f \leftarrow g*h$ is the first such multiplication. Then without loss of generality, we may assume that

$$v(g) = P(a_1, \ldots, a_n) + \sum_{i=1}^{p} c_ix_i, \tag{12.4}$$

where P is a polynomial with coefficients in F and the c_i's are in F. Moreover, we may assume $c_1 \neq 0$ without loss of generality.

Our strategy is to construct from C and the set of expressions $M\mathbf{x} + \mathbf{y}$ a new set of expressions $M'\mathbf{x}' + \mathbf{y}'$ with a computation C' having one fewer active multiplication than does C. Furthermore, $q - 1$ columns of M' will be linearly independent modulo F^r. Thus by the inductive hypothesis, C' has $q - 1$ active multiplications, implying that C has q active multiplications.

Specifically, we replace x_1 in C by an expression e which makes g of (12.4) equal to 0. Moreover, the expression e will be computable without active multiplications. The computation C' begins with a computation of e, the value of e being assigned to x_1 (which is no longer an input variable). The remainder of C' consists of C with $f \leftarrow g*h$ replaced by $f \leftarrow 0$. It is then shown the set of expressions computed by C' can be expressed as $M'\mathbf{x}' + \mathbf{y}'$, where $q - 1$ columns of M' are linearly independent modulo F^r.

We now return to the details of the proof. From (12.4) and the assumption that $c_1 \neq 0$, we have $g = 0$ if

$$x_1 = -c_1^{-1}\left[P(a_1, \ldots, a_n) + \sum_{i=2}^{p} c_i x_i\right]. \tag{12.5}$$

The right side of (12.5) is the expression e alluded to above. The computation C' formed from C as described above computes $M\mathbf{x} + \mathbf{y}$ with the expression e substituted for x_1. Thus $M'\mathbf{x}' + \mathbf{y}'$ can be written as

$$M\begin{bmatrix} e \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_p \end{bmatrix} + \mathbf{y},$$

which can be written

$$M\begin{bmatrix} -c_1^{-1}\sum_{i=2}^{p} c_i x_i \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_p \end{bmatrix} + M\begin{bmatrix} -c_1^{-1}P(a_1, \ldots, a_n) \\ 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} + \mathbf{y}. \tag{12.6}$$

Consider the first term of (12.6). Let the ith column of M be $\mathbf{m}_i$. For $2 \leq i \leq p$, define $\mathbf{m}_i' = \mathbf{m}_i - (c_i/c_1)\mathbf{m}_1$. Let M' be the matrix of $p - 1$ columns whose ith column is $\mathbf{m}_{i+1}'$ and let $\mathbf{x}' = [x_2, \ldots, x_p]^T$. Thus the first term of (12.6) is equal to $M'\mathbf{x}'$.

The second term of (12.6) is a vector with elements in $F[a_1, \ldots, a_n]$, since the elements of M are in $F[a_1, \ldots, a_n]$. Thus the second and third terms of (12.6) can be combined to form a new vector $\mathbf{y}'$ with components in $F[a_1, \ldots, a_n]$. Thus C' computes $M'\mathbf{x}' + \mathbf{y}'$. It is immediate from Lemma 12.1 that at least $q - 1$ columns of M' are linearly independent modulo F^r.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{n1} & a_{n2} & \cdots & a_{np} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ \cdot \\ v_p \end{bmatrix}$$

Fig. 12.1. General matrix-vector product.

Hence C' has $q - 1$ active multiplications, which implies that C has q active multiplications. □

We shall give two examples of the use of Theorem 12.2.

Example 12.8. We claim that the multiplication of an $n \times p$ matrix A by a vector **v** of length p requires np scalar multiplications. Formally, for $1 \leq i \leq n$ and $1 \leq j \leq p$, let a_{ij} and v_j be indeterminates. Then A**v** is the matrix-vector product of Fig. 12.1.

The direct application of Theorems 12.1 or 12.2 to A**v** can yield only that MAX(n, p) multiplications are required. However, the matrix-vector product can also be expressed as M**x**, where the ith row of M has $v_1, \ldots, v_p$ in columns $(i-1)p+1$ through ip and 0's elsewhere. The vector **x** is a column vector consisting of the rows of A concatenated together. M and **x** are illustrated in Fig. 12.2.

It is easily shown that the columns of M are linearly independent modulo F^n. Thus by Theorem 12.2 any computation for A**v** requires at least np multiplications. □

Example 12.9. Consider the problem of evaluating the nth-degree polynomial $\Sigma_{i=0}^{n} a_i x^i$. This problem can be expressed as the matrix-vector product M**x**:

$$[1, x, x^2, \ldots, x^n] \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{bmatrix}.$$

Here the elements of M are in $F[x]$ and $\mathbf{x} = [a_0, a_1, \ldots, a_n]^T$. It is straightforward to show that the set of columns of M, other than the first column, forms a linearly independent set modulo F. Thus M has n independent columns and the evaluation of an arbitrary nth-degree polynomial requires n multiplications.

Horner's rule, which evaluates a polynomial by the scheme

$$(\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0,$$

$$n \text{ rows} \quad \underbrace{\begin{bmatrix} v_1 & v_2 & \cdots & v_p & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 & & 0 \\ 0 & 0 & \cdots & 0 & v_1 & v_2 & \cdots & v_p & \cdots & 0 & 0 & & 0 \\ \cdot & & & & & & & & & & & & \cdot \\ \cdot & & & & & & & & & & & & \cdot \\ \cdot & & & & & & & & & & & & \cdot \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & v_1 & v_2 & \cdots & v_p \end{bmatrix}}_{np \text{ columns}} \begin{bmatrix} a_{11} \\ a_{12} \\ \cdot \\ \cdot \\ \cdot \\ a_{1p} \\ a_{21} \\ a_{22} \\ \cdot \\ \cdot \\ \cdot \\ a_{2p} \\ \cdot \\ \cdot \\ \cdot \\ a_{n1} \\ a_{n2} \\ \cdot \\ \cdot \\ \cdot \\ a_{np} \end{bmatrix}$$

Fig. 12.2. Equivalent form of matrix-vector product.

requires exactly n multiplications and hence is optimal in the sense of requiring as few multiplications as possible. In an analogous manner it can be shown that n additions or subtractions are needed to evaluate $\Sigma_{i=0}^{n} a_i x^i$. Thus Horner's rule uses the minimum number of arithmetic operations needed to evaluate a polynomial at a single point. □

12.6 A ROW-AND-COLUMN-ORIENTED BOUND ON MULTIPLICATIONS

We now combine Theorems 12.1 and 12.2 to obtain a stronger result than can be obtained by considering row and column independence separately. Let M be the $r \times p$ matrix with elements from $F[a_1, \ldots, a_n]$ as before. Let $\mathbf{x} = [x_1, \ldots, x_p]^T$.

Theorem 12.3. Suppose M has a submatrix S with q rows and c columns such that for any vectors $\mathbf{u}$ and $\mathbf{v}$ in F^q and F^c, respectively, $\mathbf{u}^T S \mathbf{v}$ is an element of F if and only if either $\mathbf{u} = \mathbf{0}$ or $\mathbf{v} = \mathbf{0}$. Then any computation of $M\mathbf{x}$ requires at least $q + c - 1$ multiplications.

Proof. To begin, assume without loss of generality that M itself has only q rows and that S is the first c columns of M. Suppose $M\mathbf{x}$ can be computed with s multiplications. Let $\mathbf{e}$ be the vector whose components are the s expressions computed by the multiplications. Assume in particular that the ith element of $\mathbf{e}$ is computed before the jth for $i < j$. Then as in Theorem 12.1 we may write

$$M\mathbf{x} = N\mathbf{e} + \mathbf{f}, \tag{12.7}$$

where N is a $q \times s$ matrix with elements in F, and $\mathbf{f}$ is a vector whose components are linear combinations of the a_i's and x_i's.

As in Theorem 12.1 we may conclude that $s \geq q$. If not, we could find a vector $\mathbf{y} \neq \mathbf{0}$ in F^q such that $\mathbf{y}^T N = \mathbf{0}^T$, implying that the components of $\mathbf{y}^T M$ are in F. Then the components of $\mathbf{y}^T S$ would be in F, contrary to the hypothesis (take $\mathbf{u} = \mathbf{y}$ and $\mathbf{v}^T = [1, \ldots, 1]$).

Since $s \geq q$ we can partition N into two matrices N_1 and N_2, where N_1 consists of the first $s - q + 1$ columns of N and N_2 consists of the last $q - 1$ columns of N. Also, let $\mathbf{e}_1$ and $\mathbf{e}_2$ be the first $s - q + 1$ and $q - 1$ elements of $\mathbf{e}$, respectively. Then we can write (12.7) as

$$M\mathbf{x} = N_1\mathbf{e}_1 + N_2\mathbf{e}_2 + \mathbf{f}. \tag{12.8}$$

Since N_2 is $q \times (q - 1)$, there exists a vector $\mathbf{y} \neq \mathbf{0}$ in F^q such that $\mathbf{y}^T N_2 = \mathbf{0}^T$. Multiplying (12.8) by $\mathbf{y}^T$ yields

$$\mathbf{y}^T M\mathbf{x} = \mathbf{y}^T N_1\mathbf{e}_1 + \mathbf{y}^T\mathbf{f}. \tag{12.9}$$

Let $M' = \mathbf{y}^T M$. Note that M' is a $1 \times p$ matrix which is a linear combination of the rows of M. Since the products in $\mathbf{e}_1$ can be computed without reference to those of $\mathbf{e}_2$ (the former were assumed to be computed first), it is clear that the problem $M'\mathbf{x}$ can be computed by a computation with $s - q + 1$ multiplications using (12.9). If we can show that c columns of M' are linearly independent modulo F, then $s - q + 1 \geq c$, by Theorem 12.2. Thus $s \geq q + c - 1$, as was to be proved.

We now prove that the first c columns of $M' = \mathbf{y}^T M$ are linearly independent modulo F. Let $\mathbf{y}^T = [y_1, \ldots, y_q]$. The first c entries of M' are $\Sigma_{i=1}^q y_i M_{ij}$, for $1 \leq j \leq c$, where M_{ij} is the ijth element of M. Suppose there exists a vector $\mathbf{z} \neq \mathbf{0}$ whose components are $z_1, \ldots, z_c$, each z_i in F, such that $\Sigma_{j=1}^c z_j \Sigma_{i=1}^q y_i M_{ij}$ is in F. That is, the first c columns of M' are dependent modulo F. Then $\mathbf{y}^T S\mathbf{z}$ would be in F, contrary to the hypothesis about S. Thus M' has c linearly independent columns modulo F and the theorem is proved. □

We now apply Theorem 12.3 to the multiplication of two complex numbers, $a + ib$ and $c + id$, to show that three real multiplications are required. Observe that neither Theorem 12.1 nor Theorem 12.2 is alone strong enough to do this.

Example 12.10. Consider the problem of multiplying the complex numbers $a + ib$ and $c + id$, namely, evaluating

$$M\mathbf{x} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}.$$

Let S be M itself. Let F be the reals and let

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \neq \mathbf{0} \quad \text{and} \quad \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \neq \mathbf{0}$$

be in F^2. Assume that

$$[y_1 \quad y_2] \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

is an element of F. The above product is $y_1z_1a + y_2z_1b + y_2z_2a - y_1z_2b$. If this expression is an element of F, then the coefficients of a and b must be zero. Thus

$$y_1z_1 + y_2z_2 = 0 \tag{12.10}$$

and

$$y_2z_1 - y_1z_2 = 0. \tag{12.11}$$

Suppose $y_1 \neq 0$. Then from (12.11) we have $z_2 = y_2z_1/y_1$. Substituting into (12.10) and multiplying by y_1, we have $(y_1^2 + y_2^2)z_1 = 0$. Since $y_1 \neq 0$, we have $y_1^2 + y_2^2 \neq 0$ and thus $z_1 = 0$.† Then by (12.11), $z_2 = 0$. But $z_1 = z_2 = 0$ contradicts the hypothesis that

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \neq \mathbf{0}.$$

Now assume that $y_1 = 0$. If $y_2 = 0$, we have a contradiction immediately. If $y_2 \neq 0$, then interchanging the roles of y_1 and y_2 in the argument above results in $z_1 = z_2 = 0$, again contradicting the hypothesis.

Thus Theorem 12.3 applies to $M\mathbf{x}$ with $q = c = 2$, so three real multiplications are necessary. The program in Example 12.3 shows that three multiplications are sufficient. □

Theorems 12.1–12.3 can be extended to allow divisions in a computation. When divisions are permitted as computational steps, we must allow the computation to include steps that cause division by zero for certain values of the inputs. Consequently, the theory regarding multiplicative operations (i.e., both multiplication and division) assumes that the field F is infinite. If F were finite, we could be discussing a computation which worked for no possible inputs because each possible input causes a division by zero.

† Note that the assumption F is the field of reals is essential here.

With these modifications, Theorems 12.1–12.3 can each be made to hold for multiplicative operations instead of multiplications only. In Theorem 12.2 the definition of an *active multiplicative operation*, $f \leftarrow g*h$ or $f \leftarrow g/h$, must be that either the value of at least one of g and h involves one of the x_i's and the other operand is not in F, or g is in F, h involves one of the x's and the operation is division.

12.7 PRECONDITIONING

In Example 12.9 we showed that the evaluation of an nth-degree polynomial at a single point requires n multiplications. However, there was an underlying assumption that the polynomial was represented by its coefficients. Here we consider the problem of minimizing the number of multiplications needed to compute a single polynomial if one is allowed to represent a polynomial by any set of parameters computed from the coefficients. If one were to evaluate a polynomial several times, it would make sense to spend some time computing a different representation for the polynomial, provided the new representation permitted faster evaluation. This change of representation is called preconditioning.† In Example 12.9, on polynomial evaluation, a multiplication was active if one factor involved the coefficients $a_0, a_1, \ldots, a_n$ and the other factor was not an element of F. Thus a multiplication in which both factors involved the coefficients but neither factor involved the variable x was counted. With preconditioning we get, "for free," all multiplications where neither term involves the variable.

Definition. The *degree* of a multivariate polynomial is the maximum power of any variable in the polynomial. For example, $p(x, y) = x^3 + x^2y^2$ is of degree 3.

The next lemma states that for any $v + 1$ polynomials in v variables, there exists a nontrivial polynomial function of the given polynomials which is identically zero. For example, let $p_1 = x^2$ and $p_2 = x + x^3$. Then $p_1 + 2p_1^2 + p_1^3 - p_2^2 = 0$ for all x.

Lemma 12.2. Let $p_i(x_1, \ldots, x_r)$, $1 \le i \le n$, represent n polynomials. If $n > r$, then there exists a polynomial $g(p_1, \ldots, p_n)$ whose coefficients are not identically zero, but which is identically zero as a function of the x_i's.

Proof. Let S_d be the set of terms of the form $p_1^{i_1}p_2^{i_2} \ldots p_n^{i_n}$, where $0 \le i_j \le d$ for all j. Note that $\|S_d\| = (d + 1)^n$. Let m be the maximum degree of any of

† Note the difference between this situation and the one in Section 8.5, where all points at which a polynomial was to be evaluated were known. Here, we may process only the coefficients before evaluation, and the evaluations are to be done one at a time, i.e., on-line rather than off-line.

the p_i's. Then each $q \in S_d$ is a polynomial in $x_1, x_2, \ldots, x_r$ of degree at most dmn.

Now any polynomial of degree dmn in r variables can be represented by a vector of length $(dmn + 1)^r$. The components of the vector are the coefficients of the terms in a fixed order. It thus makes sense to talk about linear independence of polynomials, and in particular to say that, by the basis theorem of linear algebra, any set of $(dmn + 1)^r + 1$ polynomials from S_d must be linearly dependent, i.e., have a nontrivial sum that totals zero.

In particular, if m, n, and r are fixed, and $n > r$, it is not hard to show that $(d + 1)^n \geq (dmn + 1)^r$ for sufficiently large d, since $(d + 1)^n$ is a higher-degree polynomial in d than $(dmn + 1)^r$. Thus there exists a d such that a nontrivial sum of members of S_d is identically zero as a function of the x's. This sum is the desired polynomial g. □

Using Lemma 12.2 we now show that any set of parameters for a polynomial in which the coefficients are polynomial functions of parameters requires $n + 1$ parameters to represent an arbitrary nth-degree polynomial.

Lemma 12.3. Let M be a one-to-one mapping from an n-dimensional vector space C^n of coefficients onto an r-dimensional vector space D^r of parameters. If M^{-1} is such that each component of a vector in C^n is a polynomial function of the components of the corresponding vector in D^r, then $r \geq n$.

Proof. Assume $r < n$ and let M^{-1} be given by $c_i = p_i(d_1, \ldots, d_r)$, where the p_i's are polynomials, the d_i's parameters, and the c_i's coefficients. By Lemma 12.2 there exists a nontrivial polynomial g such that $g(c_1, \ldots, c_n)$ is identically zero for all values of the d_i's.† But since g itself is not identically zero, there exist polynomials with coefficients $c_1, \ldots, c_n$ such that $g(c_1, \ldots, c_n)$ is not zero. These polynomials cannot be represented in terms of the d_i's, a contradiction. □

We now show that even with preconditioning, the evaluation of an nth-degree polynomial requires at least $n/2$ multiplications.

Theorem 12.4. A computation for polynomial evaluation requires at least $n/2$ multiplications to evaluate an arbitrary nth-degree polynomial at a single point, even if multiplications in computations involving only coefficients are not counted.

Proof. Assume there exists a computation with m multiplication steps involving the unknown x. Let the results of the multiplications be the expressions $f_1, f_2, \ldots, f_m$. Then each f_i can be expressed as

$$f_i = [L_{2i-1}(f_1, \ldots, f_{i-1}, x) + \beta_{2i-1}] * [L_{2i}(f_1, \ldots, f_{i-1}, x) + \beta_{2i}],$$

† Observe that g is a polynomial in the c_i's and hence in the d_i's, since the c_i's are polynomials in the d_i's.

$1 \leq i \leq m$, where each L_i is a linear function of the f's and x, and each β_i is a polynomial function of the coefficients. The polynomial $p(x)$ which is computed can be expressed as

$$p\,(x) = L_{2m+1}(f_1, \ldots, f_m, x) + \beta_{2m+1}.$$

Thus $p(x)$ can be represented by a new set of $2m+1$ parameters, the β_i's. Furthermore, the coefficients of $p(x)$ are polynomial functions of the β_i's. By Lemma 12.3, $2m+1 \geq n+1$. Thus $m \geq n/2$. □

Theorem 12.4 gives a lower bound on the number of multiplications for preconditioned evaluation of a polynomial at a single point. The problem of representing a polynomial so that it can be computed in $n/2$ multiplications is of considerable interest. Not only must we find a set of parameters to represent the polynomial so that it is easy to evaluate, but we also must be able to readily compute the parameters from the coefficients of the polynomial. Thus it is desirable that the parameters be rational functions of the coefficients. Techniques for finding such parameters are found in the exercises.

EXERCISES

12.1 Let F be a field and x an indeterminate. Show that the set of polynomials with variable x and coefficients from F forms a commutative ring $F[x]$ and that the multiplicative identity in F is a multiplicative identity in $F[x]$.

12.2 Show that any computation for the expressions

$$\begin{aligned} &ac + bd,\\ &ac - bd,\\ &bc + ad,\\ &bc - ad, \end{aligned}$$

requires at least four multiplications, where a, b, c, d are elements of a field F.

12.3 Show that any computation of the product of a column vector by a row vector

$$\begin{bmatrix} a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_m \end{bmatrix} [b_1, b_2, \ldots, b_n]$$

requires at least mn multiplications.

12.4 The general nth-degree polynomial in two variables x and y is $\Sigma_{i=0}^{n}\Sigma_{j=0}^{n} a_{ij}x^iy^j$. Show that $(n+1)(n+2)/2 - 1$ multiplications are necessary and sufficient to evaluate such a polynomial with preconditioning.

12.5 Show that multiplication of a 2×2 Toeplitz matrix (see the exercises to Chapter 6, p. 249) by a vector, i.e.,

$$\begin{bmatrix} b & c \\ a & b \end{bmatrix}\begin{bmatrix} d \\ e \end{bmatrix},$$

requires three multiplications.

12.6 Generalize Exercise 12.5 to show that $2n - 1$ multiplications are needed to multiply an $n \times n$ Toeplitz matrix by a vector. How close to the bound can you come for $n = 3$?

***12.7** We could generalize the multiplication algorithm of Section 2.6 to break the multiplier and multiplicand into three pieces each, and then evaluate the matrix-vector product

$$\begin{bmatrix} a & 0 & 0 \\ b & a & 0 \\ c & b & a \\ 0 & c & b \\ 0 & 0 & c \end{bmatrix}\begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

in as few multiplications as possible. How many multiplications are needed to solve this problem? Does the approach yield an improvement on Section 2.6?

***12.8** Let F be a field and let $a_1, \ldots, a_n$ and $x_1, \ldots, x_p$ be disjoint sets of indeterminates. Let $\mathbf{x} = [x_1, \ldots, x_p]^T$ and let M be an $r \times p$ matrix with elements from $F[a_1, \ldots, a_n]$ having q independent columns modulo F^r. Show that with preconditioning of $\mathbf{x}$, i.e., not counting products which involve only x_i's, a computation of the product $M\mathbf{x}$ still requires $q/2$ multiplications.

***12.9** Suppose exactly k expressions of a set P, no linear combination of which is a constant, are single multiplications (e.g., they are $a*b$ for indeterminates a and b) and assume that q multiplications suffice to compute all expressions in P. Show that there is a computation with q multiplications of which k are those expressions of P which can be computed by single multiplications.

***12.10** Show that at least three multiplications are required to compute the expressions ac and $bc + ad$. [*Hint:* Use Exercise 12.9.]

***12.11** Show, that at least $3k$ multiplications are required to compute the set of $2k$ expressions a_ic_i and $b_ic_i + a_id_i$, for $1 \le i \le k$.

Exercises 12.12 through 12.14 make reference to the problem of multiplying two 2×2 matrices A and B. Specifically, we wish to compute the four expressions of the product AB using noncommutative multiplications:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

***12.12** Show that if there is any computation which works for an arbitrary ring and uses q multiplications, then there is another computation, using q multiplications, which works for the ring of integers modulo 2 and in which all multiplications are of the form

$$(a_{i_1j_1} + a_{i_2j_2} + \cdots + a_{i_kj_k}) * (b_{m_1n_1} + b_{m_2n_2} + \cdots + b_{m_rn_r}).$$

.13 Show that if the left side of any multiplication as described in the previous exercise is a single a_{ij} (i.e., $k = 1$), then the computation has at least seven multiplications. [*Hint:* Set $a_{ij} = 0$ and use Exercise 12.11 with $k = 2$.]

2.14 Show that if none of the left sides are single a_{ij}'s, then one still requires seven multiplications. [*Hint:* Show that in each case there is some set of conditions, e.g., $a_{ij} = 0$ or $a_{kl} = a_{mn}$, such that one multiplication is identically zero and the resulting computation solves the problem of Exercise 12.11 with $k = 2$.]

2.15 Show that multiplication of a 2×2 matrix by a $2 \times n$ matrix requires exactly $\lceil 7n/2 \rceil$ multiplications.

2.16 Exhibit an algorithm to multiply an $n \times 2$ matrix by a $2 \times m$ matrix using $mn + m + n$ multiplications assuming the scalars form a commutative ring. Combine this result with Exercise 12.15 to show that matrix multiplication requires more multiplications in the noncommutative case than in the commutative case.

Definition. Let R be a ring and $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ disjoint sets of indeterminates. A *bilinear form* is an expression of the form

$$\sum_{i,j} r_{ij} a_i b_j,$$

where each r_{ij} is an element of R.

12.17 a) Show that for any set of bilinear forms there exists a computation which is minimal with respect to the number of multiplications and in which all multiplications are between linear functions of the a's and b's.

b) Show that there exists a minimal computation with respect to multiplications where each multiplication is between a linear function of the a's and a linear function of the b's. (Note that this part does not hold if we restrict the underlying ring to be commutative.)

c) Given that computations are generalized to allow division, then show there exists a computation which is minimal with respect to multiplicative operations and which does not make use of division.

*12.18 Let R be a noncommutative ring and let **a**, **b**, and **x** be column vectors of indeterminates $[a_1, \ldots, a_m]^T$, $[b_1, \ldots, b_n]^T$, and $[x_1, \ldots, x_p]^T$, respectively. Let X be a matrix of linear sums of x_i's. Exercise 12.17 implies that the computations of the set of bilinear forms $(\mathbf{a}^T X)^T$ can be expressed as

$$M(P\mathbf{a} \cdot Q\mathbf{x}),$$

where M, P, and Q are matrices with elements from F. Define the *left dual* of the set of expressions $(\mathbf{a}^T X)^T$ to be the set of expressions $(\mathbf{b}^T X^T)^T$ and define the *P-dual* of the computation $M(P\mathbf{a} \cdot Q\mathbf{x})$ to be the computation $P^T(M^T\mathbf{b} \cdot Q\mathbf{x})$. Prove that the P-dual of any computation of $(\mathbf{a}^T X)^T$ computes the left dual of $(\mathbf{a}^T X)^T$.

*12.19 Prove that the minimum number of multiplications needed to multiply an $m \times n$ matrix by an $n \times p$ matrix over a noncommutative ring is the same as to multiply an $n \times m$ matrix by an $m \times p$ matrix. [*Hint:* Use Exercise 12.18.]

So far the exercises contain material concerning lower bounds on the number of arithmetic operations. The following exercises contain material of a more general nature. Problems 12.20 to 12.23 are related to transposing a matrix. For $1 \leq i, j \leq n$, let a_{ij} be an indeterminate. Consider a model of computation in which each variable is an n-tuple of indeterminates. A step $a \leftarrow b \; \theta \; c$ of a computation assigns to an n-tuple a indeterminates selected from the indeterminates present in b or c. The n-tuples b and c are not changed.

***12.20** Prove that any computation of the set of n-tuples

$$\{(a_{1i}, \ldots, a_{ni}) | 1 \leq i \leq n\}$$

from the set of inputs $\{(a_{i1}, \ldots, a_{in}) | 1 \leq i \leq n\}$ requires at least $n \log n$ steps. [*Hint:* For each i and j let s_{ij} be the maximum number of indeterminates with subscript j that occur in a single n-tuple containing a_{ij}. Let

$$f = \sum_{i=1}^{n} \sum_{j=1}^{n} \log s_{ij}.$$

Consider the change in f as steps of the computation are executed.]

****12.21** Restrict the steps of a computation to the form $a \leftarrow b \; \theta \; c$, where θ consists of selecting $n/2$ indeterminates from a cyclic shift of b and selecting $n/2$ indeterminates from the complementary positions in c. Find a computation for the set of n-tuples in Exercise 12.20 with $O(n \log n)$ steps.

****12.22** Let G be a directed acyclic graph with n designated *source vertices* (vertices having no incoming edges) and n designated *output vertices* (vertices having no outgoing edges). Let X and Y be subsets of source and output vertices, respectively. Let $G(X, Y)$ be the subgraph consisting of directed edges on paths from vertices in X to vertices in Y. The *capacity* of $G(X, Y)$ is the minimum number of vertices whose removal (along with the incoming and outgoing edges) separates each vertex in X from each vertex in Y. Assume for each X and Y the graph $G(X, Y)$ has capacity $\text{MIN}(\|X\|, \|Y\|)$. Show for each n there exists such a G with $cn \log n$ edges for some fixed constant c.

****12.23** A *shifting network* is a directed graph with n source vertices numbered 0 to $n-1$ and n output vertices numbered 0 to $n-1$ such that for each s, $0 \leq s \leq n-1$, there exists a set of vertex disjoint paths which contains a path from source vertex i to output vertex $(i+s)$ modulo n, for each i.

a) Show that there exists a shifting network with $2n \log n$ edges for each n.
b) Show that asymptotically $n \log n$ edges are required for a shifting network.
c) Show that a shifting network can be used to compute the transpose of a matrix.

Definition. Let F be a field and $x_1, \ldots, x_n$ indeterminates. A *linear* computation is a sequence of steps of the form $a \leftarrow \lambda_1 b + \lambda_2 c$ where a is a variable, b and c are variables or indeterminates, and λ_1 and λ_2, called *constants*, are elements of F.

****12.24** Let F be the field of complex numbers. Let A be a matrix with elements from F and let $\mathbf{x} = [x_1, \ldots, x_n]^T$. Show that any linear computation of $A\mathbf{x}$ requires $\log[\det(A)]/\log(2c)$ steps, where c is the maximum modulus† of any constant appearing in a step of the computation.

*12.25 Prove that a linear computation of the Fourier transform of $[x_1, \ldots, x_n]$, with constants whose moduli are less than or equal to one, requires $\frac{1}{2}n \log n$ steps.

The following six problems are concerned with the minimum number of two-input gates needed to realize a Boolean function.

*12.26 Show that each Boolean function of n variables can be realized with at most $n2^n$ two-input gates.

*12.27 Show that for each n there exists a Boolean function of n variables whose minimal realization requires approximately $2^n/n$ two-input gates.

*12.28 Little is known about the complexity of realizing specific Boolean functions. However, if we restrict realizations to networks which are trees, then we can show that certain functions require $n^2/\log n$ gates. Prove that the following condition is sufficient for a Boolean function of n variables to require $n^2/\log n$ gates.

Condition: Let $f(x_1, \ldots, x_n)$ be a Boolean function of n variables. Partition the x_i into $b = n/\log n$ blocks of $\log n$ variables each. Assign values (0 or 1) to all variables in $b - 1$ blocks. There are $2^n/n$ ways to do this. The result is a function of $\log n$ variables, which is one of 2^n functions of $\log n$ variables. The actual function obtained depends on the assignment of values to the variables in the $b - 1$ blocks. Let B_i be the block of variables not set to 0. If on the order of $2^n/n$ distinct functions of the variables in B_i can be obtained by appropriate assignments of 0's and 1's to the other variables, and if this statement is true for each i, $1 \leq i \leq \log n$, then any tree network for f must have $n^2/\log n$ gates.

*12.29 Let $m = 2 + \log n$. Let $\{\mathbf{t}_{ij} | 1 \leq i \leq n/m,\ 1 \leq j \leq m\}$ be a set of 0–1 valued m-vectors such that each $\mathbf{t}_{ij}$ has at least two components with value 1. Let

$$f(x_{11}, x_{12}, \ldots, x_{n/m,m}) = \bigoplus_{\substack{1\leq i\leq n/m \\ 1\leq j\leq m}} x_{ij} \cdot \left[\bigoplus_{\substack{1\leq k\leq n/2 \\ k\neq i}} \prod_{\substack{1\leq l\leq m \\ \text{such that } t_{ij}^l=1}} x_{kl} \right],$$

where t_{ij}^l is the lth component of $\mathbf{t}_{ij}$. Prove that any tree network realization for f has at least $n^2/\log n$ gates.

*12.30 Construct other Boolean functions whose realizations as tree networks require (a) $n^{3/2}$ gates, and (b) $n^2/\log n$ gates.

12.31 Let $S_i(x_1, \ldots, x_n)$ be the symmetric Boolean function which has value 1 if and only if exactly i x_j's have value 1.

a) Find a realization of $S_3(x_1, \ldots, x_n)$ with as few two-input gates as possible.

b) Find a tree realization of $S_3(x_1, \ldots, x_n)$ with as few two-input gates as possible.

† The *modulus* of $a + bi$ is $\sqrt{a^2 + b^2}$.

**12.32 In Example 12.9 we proved that the evaluation of an arbitrary polynomial of degree n required n multiplications. Exhibit a specific polynomial of n variables with real coefficients that requires n multiplications.

**12.33 Prove that matrix multiplication for the semiring consisting of positive integers with operations MIN and + requires cn^3 operations for some constant $0 < c < 1$.

**12.34 Prove that Boolean matrix multiplication with operations AND and OR requires n^3 operations.

Exercises 12.35 and 12.36 are concerned with representing a polynomial so that it can be evaluated in approximately $n/2$ multiplications.

**12.35 Let $p(x)$ be an nth-degree polynomial where n is odd and greater than 1. Write

$$p(x) = (x^2 - \alpha)q(x) + bx + c.$$

a) Show that for suitable α, we can make $b = 0$.

b) Prove that there exist suitable α_i's and β_i's such that $p(x)$ can be expressed as

$$p(x) = (x^2 - \alpha_1)[(x^2 - \alpha_2)[\cdots] + \beta_2] + \beta_1$$

and hence $p(x)$ represented by the α_i's and β_i's can be evaluated in $\lfloor n/2 \rfloor + 2$ multiplications.

c) The α_i's in (b) may be complex numbers. How can this difficulty be overcome? [*Hint:* Substitute $y + c$ for x in $p(x)$ for suitable c and instead of evaluating $p(x)$ at $x = x_0$, evaluate some polynomial $p'(y)$ at $y = x_0 - c$.]

Exercise 12.35 provides no method to compute the α_i's. Furthermore, in general the α_i's are not rational numbers. The following exercise is an attempt to overcome these problems.

**12.36 Let m be an integer. For $1 \leq l \leq m$ let

$$r_{l0} = \frac{(m-l)^2 + (l-1)^2 + 3m + 1}{2}$$

and $r_{lj} = m - l + j + 1$, $1 \leq j \leq l$. Define a chain with α's and β's as parameters to be a computation of the form

$$\begin{aligned}
c_{i1}(x) &\leftarrow (x^{r_{i0}} + \alpha_{i1})(x^{r_{i1}} + \beta_{i1}),\\
c_{i2}(x) &\leftarrow (c_{i1} + \alpha_{i2})(x^{r_{i2}} + \beta_{i2}),\\
&\ \ \vdots\\
c_{ii}(x) &\leftarrow (c_{i,i-1} + \alpha_{ii})(x^{r_{ii}} + \beta_{ii}).
\end{aligned}$$

Let $p(x) = \sum_{i=1}^{l} c_{ii}(x)$.

a) Prove that the coefficient of the highest power of x depending on α_{ij} is given by $\sum_{k=j}^{l} r_{ik}$ and that the coefficient of the highest power of x depending on β_{ij} is given by $\sum_{k=0}^{l} r_{ik} - r_{ij}$.

b) Prove that every polynomial of degree less than or equal to $m(m + 1) + 1$

with leading coefficient unity can be represented by $m(m+1)+1$ parameters such that (i) the polynomial can be evaluated from the parameters in $m(m+1)/2+O(m)$ multiplications, and (ii) the parameters are rational functions of the coefficients.

Research Problems

With a straight-line program we can associate a directed acyclic graph as follows. The vertices of the graph represent the inputs and the variables. If $f \leftarrow g*h$ is a step, then there is a directed edge from g to f and from h to f. Observe that the number of steps in the program is exactly twice the number of edges. Thus obtaining a lower bound on the number of edges for any directed acyclic graph arising from a computation for a given problem gives a lower bound on the number of steps in any straight-line program for the problem.

12.37 For large n, prove or disprove that every directed acyclic graph with n source and n output vertices satisfying the capacity condition in Exercise 12.22 has at least $n \log n$ edges.

12.38 Does every graph arising from a straight-line program for multiplying two n-bit integers (bitwise operations are assumed) satisfy the capacity condition in Exercise 12.22?

BIBLIOGRAPHIC NOTES

Theorem 12.2 and the general formulation of the problem treated in this section are from Winograd [1970a]. Theorems 12.1 and 12.3 are from Fiduccia [1971]. The fact that, assuming no divisions, n multiplications are needed to evaluate an nth-degree polynomial is attributed (Knuth [1969]) to A. Garsia. Pan [1966] extended the result to multiplicative operations. Motzkin [1955] showed that $[n/2]+1$ multiplications are necessary for preconditioned polynomial evaluation. Ostrowski [1954] did some of the initial work in this direction. Winograd [1970b] showed three multiplications are necessary for complex products.

Exercise 12.7 is discussed in Winograd [1973]. The material on lower bounds for matrix multiplication (Exercises 12.9 through 12.16) is based on Hopcroft and Kerr [1971]. The results of Exercise 12.17 were independently observed by several people. Part (c) is attributed by S. Winograd (private communication) to P. Ungar. Exercises 12.18 and 12.19 are from Hopcroft and Muszinski [1973]. Exercises 12.20 through 12.22 as well as Exercises 12.37 and 12.38 are based on discussions with R. Floyd. Exercises 12.24 and 12.25 are from Morgenstern [1973], Exercises 12.28 and 12.29 are from Nečiporuk [1966], and Exercise 12.30(a) is from Harper and Savage [1972]. Exercise 12.32 is from Strassen [1974]. Exercise 12.33 is from Kerr [1970] and Exercise 12.34 from Pratt [1974]. Exercise 12.35 is from Eve [1964] and Exercise 12.36 is from Rabin and Winograd [1971].

Some additional attempts to obtain lower bounds on numbers of arithmetic operations are found in Borodin and Cook [1974] and Kedem [1974].

BIBLIOGRAPHY

ADEL'SON-VEL'SKII, G. M., AND Y. M. LANDIS [1962]. "An algorithm for th organization of information," *Dokl. Akad. Nauk SSSR* **146,** 263–266 (in Ru ian). English translation in *Soviet Math. Dokl.* 3 (1962), 1259–1262.

AHO, A. V. (ed.) [1973]. *Currents in the Theory of Computing,* Prentice-Hall, Engle wood Cliffs, N.J.

AHO, A. V., M. R. GAREY, AND J. D. ULLMAN [1972]. "The transitive reduction of directed graph," *SIAM J. Computing* **1:**2, 131–137.

AHO, A. V., D. S. HIRSCHBERG, AND J. D. ULLMAN [1974]. "Bounds on th complexity of the maximal common subsequence problem." Conference Record IEEE 15th Annual Symposium on Switching and Automata Theory.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1968]. "Time and tape complexit of pushdown automaton languages," *Information and Control* **13:**3, 186–206.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1974]. "On finding lowes common ancestors in trees," *SIAM J. Computing,* to appear.

AHO, A. V., K. STEIGLITZ, AND J. D. ULLMAN [1974]. "Evaluating polynomials a fixed sets of points," Bell Laboratories, Murray Hill, N. J.

AHO, A. V., AND J. D. ULLMAN [1972]. *The Theory of Parsing, Translation, and Compiling,* Volume 1: *Parsing,* Prentice-Hall, Englewood Cliffs, N.J.

AHO, A. V., AND J. D. ULLMAN [1973]. *The Theory of Parsing, Translation, and Compiling,* Volume 2: *Compiling,* Prentice-Hall, Englewood Cliffs, N.J.

ARLAZAROV, V. L., E. A. DINIC, M. A. KRONROD, AND I. A. FARADZEV [1970]. "On economical construction of the transitive closure of a directed graph," *Dokl. Akad. Nauk SSSR* **194,** 487–488 (in Russian). English translation in *Soviet Math. Dokl.* **11:**5, 1209–1210.

BELAGA, E. C. [1961]. "On computing polynomials in one variable with initial preconditioning of the coefficients," *Problemi Kibernetiki* **5,** 7–15.

BELLMAN, R. E. [1957]. *Dynamic Programming,* Princeton University Press, Princeton, N.J.

BERGE, C. [1958]. *The Theory of Graphs and its Applications,* Wiley, New York.

BIRKHOFF, G., AND T. BARTEE [1970]. *Modern Applied Algebra,* McGraw-Hill, New York.

BLUESTEIN, L. I. [1970]. "A linear filtering approach to the computation of the discrete Fourier transform," *IEEE Trans. on Electroacoustics AU*-**18:**4, 451–455. Also in Rabiner and Rader [1972], pp. 317–321.

BLUM, M. [1967]. "A machine independent theory of the complexity of recursive functions," *J. ACM* **14:**2, 322–336.

BLUM, M., R. W. FLOYD, V. R. PRATT, R. L. RIVEST, AND R. E. TARJAN [1972]. "Time bounds for selection," *J. Computer and System Sciences* 7:4, 448–461.

BOOK, R. V. [1972]. "On languages accepted in polynomial time," *SIAM J. Computing* **1:**4, 281–287.

BOOK, R. V. [1974]. "Comparing complexity classes," *J. Computer and System Sciences,* to appear.

BORODIN, A. B. [1973a]. "Computational complexity—theory and practice," in Aho [1973], pp. 35–89.

BORODIN, A. B. [1973b]. "On the number of arithmetics required to compute certain functions—circa May 1973," in Traub [1973], pp. 149–180.

BORODIN, A. B., AND S. A. COOK [1974]. "On the number of additions to compute specific polynomials," *Proc. 6th Annual ACM Symposium on Theory of Computing,* 342–347.

BORODIN, A. B., AND I. MUNRO [1971]. "Evaluating polynominals at many points," *Information Processing Letters* **1**:2, 66–68.

BORODIN, A. B., AND I. MUNRO [1975]. The Computational Complexity of Algebraic and Numeric Problems, to appear, American Elsevier, N.Y.

BROWN, W. S. [1971]. "On Euclid's algorithm and greatest common divisors," *J. ACM* **18**:4, 478–504.

BROWN, W. S. [1973]. *ALTRAN User's Manual* (3rd edition), Bell Laboratories, Murray Hill, N.J.

BRUNO, J. L., AND R. SETHI [1974]. "Register allocation for a one-register machine," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems.*

BUNCH, J., AND J. E. HOPCROFT [1974]. "Triangular factorization and inversion by fast matrix multiplication," *Math. Comp.* **28**:125, 231–236.

BURKHARD, W. A. [1973]. "Nonrecursive tree traversal algorithms," *Proc. 7th Annual Princeton Conference on Information Sciences and Systems,* 403–405.

COLLINS, G. E. [1973]. "Computer algebra of polynomials and rational functions," *American Math. Monthly* **80**:7, 725–754.

CONSTABLE, R. L., H. B. HUNT III, AND S. K. SAHNI [1974]. "On the computational complexity of scheme equivalence," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems.*

COOK, S. A. [1971a]. "Linear time simulation of deterministic two-way pushdown automata," *Proc. IFIP Congress* **71**, TA-2. North-Holland, Amsterdam, 172–179.

COOK, S. A. [1971b]. "The complexity of theorem proving procedures," *Proc. 3rd Annual ACM Symposium on Theory of Computing,* 151–158.

COOK, S. A. [1973]. "A hierarchy for nondeterministic time complexity," *J. Computer and System Sciences* **7**:4, 343–353.

COOK, S. A., AND S. O. AANDERAA [1969]. "On the minimum complexity of functions," *Trans. Amer. Math. Soc.* **142**, 291–314.

COCK, S. A., AND R. A. RECKHOW [1973]. "Time-bounded random access machines," *J. Computer and System Sciences* **7**:4, 354–375.

COOK, S. A., AND R. A. RECKHOW [1974]. "On the lengths of proofs in the propositional calculus," *Proc. 6th Annual ACM Symposium on Theory of Computing,* 135–148.

COOLEY, J. M., P. A. LEWIS, AND P. D. WELCH [1967]. "History of the fast Fourier transform," *Proc. IEEE* **55**, 1675–1677.

COOLEY, J. M., AND J. W. TUKEY [1965]. "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.* **19**, 297–301.

CRANE, C. A. [1972]. "Linear lists and priority queues as balanced binary trees," Ph.D. Thesis, Stanford University.

DANIELSON, G. C., AND C. LANCZOS [1942]. "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids," *J. Franklin Institute* **233**, 365–380 and 435–452.

DANTZIG, G. B., W. O. BLATTNER, AND M. R. RAO [1966]. "All shortest routes from a fixed origin in a graph," *Theory of Graphs*, International Symposium, Rome, July 1966. Proceedings published by Gordon and Breach, New York, 1967, pp. 85–90.

DAVIS, M. [1958]. *Computability and Unsolvability*, McGraw-Hill, New York.

DIJKSTRA, E. W. [1959]. "A note on two problems in connexion with graphs," *Numerische Mathematik* **1**, 269–271.

DIVETTI, L., AND A. GRASSELLI [1968]. "On the determination of minimum feedback arc and vertex sets," *IEEE Trans. Circuit Theory* **CT-15:1**, 86–89.

EHRENFEUCHT, A., AND H. P. ZEIGER [1974]. "Complexity measures for regular expressions," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 75–79.

ELGOT, C. C., AND A. ROBINSON [1964]. "Random access stored program machines," *J. ACM* **11:4**, 365–399.

EVE, J. [1964]. "The evaluation of polynomials," *Numerische Mathematik* **6**, 17–21.

EVEN, S. [1973]. "An algorithm for determining whether the connectivity of a graph is at least *k*," TR-73-184, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

FIDUCCIA, C. M. [1971]. "Fast matrix multiplication," *Proc. 3rd Annual ACM Symposium on Theory of Computing*, 45–49.

FIDUCCIA, C. M. [1972]. "Polynomial evaluation via the division algorithm—the fast Fourier transform revisited," *Proc. 4th Annual ACM Symposium on Theory of Computing*, 88–93.

FISCHER, M. J. [1972]. "Efficiency of equivalence algorithms," in Miller and Thatcher [1972], pp. 153–168.

FISCHER, M. J., AND A. R. MEYER [1971]. "Boolean matrix multiplication and transitive closure," *Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory*, 129–131.

FISCHER, M. J., AND M. S. PATERSON [1974]. "String-matching and other products." Project MAC Technical Memorandum 41, MIT, Cambridge, Mass.

FISCHER, M. J., AND M. O. RABIN [1974]. "The complexity of theorem proving procedures," Project MAC Report, MIT, Cambridge, Mass.

FLOYD, R. W. [1962]. "Algorithm 97: shortest path," *Comm. ACM* **5:6**, 345.

FLOYD, R. W. [1964]. "Algorithm 245: treesort 3," *Comm. ACM* **7:12**, 701.

FLOYD, R. W., AND R. L. RIVEST [1973]. "Expected time bounds for selection," Computer Science Dept., Stanford University.

Ford, L. R., and S. M. Johnson [1959]. "A tournament problem," *Amer. Math. Monthly* **66**, 387–389.

Frazer, W. D., and A. C. McKellar [1970]. "Samplesort: a sampling approach to minimal storage tree sorting," *J. ACM* **17**:3, 496–507.

Furman, M. E. [1970]. "Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph," *Dokl. Akad. Nauk SSSR* **194**, 524 (in Russian). English translation in *Soviet Math. Dokl.* **11**:5, 1252.

Gale, D., and R. M. Karp [1970]. "A phenomenon in the theory of sorting," *Conference Record, IEEE 11th Annual Symposium on Switching and Automata Theory,* 51–59.

Galler, B. A., and M. J. Fischer [1964]. "An improved equivalence algorithm," *Comm. ACM* **7**:5, 301–303.

Garey, M. R., D. S. Johnson, and L. Stockmeyer [1974]. "Some simplified polynomial complete problems," *Proc. 6th Annual ACM Symposium on Theory of Computing,* 47–63.

Gentleman, W. M. [1972]. "Optimal multiplication chains for computing powers of polynomials," *Math. Comp.* **26**:120, 935–940.

Gentleman, W. M., and S. C. Johnson [1973]. "Analysis of algorithms, a case study: determinants of polynomials." *Proc. 5th Annual ACM Symposium on Theory of Computing,* 135–141.

Gentleman, W. M., and G. Sande [1966]. "Fast Fourier transforms for fun and profit," *Proc. AFIPS 1966 Fall Joint Computer Conference,* v. 29, Spartan, Washington, D.C., pp. 563–578.

Gilbert, E. N., and E. F. Moore [1959]. "Variable length encodings," *Bell System Technical J.* **38**:4, 933–968.

Godbole, S. S. [1973]. "On efficient computation of matrix chain products," *IEEE Transactions on Computers* **C-22**:9, 864–866.

Good, I. J. [1958]. "The interaction algorithm and practical Fourier series," *J. Royal Statistics Soc.,* Ser B, **20**, 361–372. Addendum 22 (1960), 372–375.

Graham, R. L. [1972]. "An efficient algorithm for determining the convex hull of a planar set," *Information Processing Letters* **1**:4, 132–133.

Gray, J. N., M. A. Harrison, and O. H. Ibarra [1967]. "Two way pushdown automata," *Information and Control* **11**:3, 30–70.

Hadian, A., and M. Sobel [1969]. "Selecting the tth largest using binary errorless comparisons," *Tech. Rept. 121,* Department of Statistics, University of Minnesota, Minneapolis.

Hall, A. D. [1971]. "The ALTRAN system for rational manipulation—a survey," *Comm. ACM* **14**:8, 517–521.

Harary, F. [1969]. *Graph Theory,* Addison-Wesley, Reading, Mass.

Harper, L. H., and J. E. Savage [1972]. "On the complexity of the marriage problem," in *Advances in Mathematics* **9**:3, 299–312.

Hartmanis, J. [1971]. "Computational complexity of random access stored program machines," *Mathematical Systems Theory* **5**:3, 232–245.

HARTMANIS, J., AND J. E. HOPCROFT [1971]. "An overview of the theory of computational complexity," *J. ACM* **18**:3, 444–475.

HARTMANIS, J., P. M. LEWIS II, AND R. E. STEARNS [1965]. "Classification of computations by time and memory requirements," *Proc. IFIP Congress* 65, Spartan, N.Y., 31–35.

HARTMANIS, J., AND R. E. STEARNS [1965]. "On the computational complexity of algorithms," *Trans. Amer. Math. Soc.* **117**, 285–306.

HECHT, M. S. [1973]. "Global data flow analysis of computer programs," Ph.D. Thesis, Dept. of Electrical Engineering, Princeton University.

HEINDEL, L. E., AND E. HOROWITZ [1971]. "On decreasing the computing time for modular arithmetic," *Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory,* 126–128.

HENNIE, F. C., AND R. E. STEARNS [1966]. "Two tape simulation of multitape machines," *J. ACM* **13**:4, 533–546.

HIRSCHBERG, D. S. [1973]. "A linear space algorithm for computing maximal common subsequences," TR-138, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton University, Princeton, N.J.

HOARE, C. A. R. [1962]. "Quicksort," *Computer J.* **5**:1 10–15.

HOHN, F. E. [1958]. *Elementary Matrix Algebra,* Macmillan, New York.

HOPCROFT, J. E. [1971]. "An *n* log *n* algorithm for minimizing states in a finite automaton," in Kohavi and Paz [1971], pp. 189–196.

HOPCROFT, J. E., AND R. M. KARP [1971]. "An algorithm for testing the equivalence of finite automata," TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

HOPCROFT, J. E., AND L. R. KERR [1971]. "On minimizing the number of multiplications necessary for matrix multiplication," *SIAM J. Applied Math.* **20**:1, 30–36.

HOPCROFT, J. E., AND J. MUSINSKI [1973]. "Duality in determining the complexity of noncommutative matrix multiplication," *Proc. 5th Annual ACM Symposium on Theory of Computing,* 73–87.

HOPCROFT, J. E., AND R. E. TARJAN [1973a]. "Efficient planarity testing," TR 73-165, Dept. of Computer Science, Cornell University, Ithaca, N.Y., also *J. ACM,* to appear.

HOPCROFT, J. E., AND R. E. TARJAN [1973b]. "Dividing a graph into triconnected components," *SIAM J. Computing* **2**:3.

HOPCROFT, J. E., AND R. E. TARJAN [1973c]. "Efficient algorithms for graph manipulation," *Comm. ACM* **16**:6, 372–378.

HOPCROFT, J. E., AND J. D. ULLMAN [1969]. *Formal Languages and Their Relation to Automata,* Addison-Wesley, Reading, Mass.

HOPCROFT, J. E., AND J. D. ULLMAN [1973]. "Set merging algorithms," *SIAM J. Computing,* **2**:4, 294–303.

HOPCROFT, J. E., AND J. K. WONG [1974]. "A linear time algorithm for isomorphism of planar graphs," *Proc. 6th Annual ACM Symposium on Theory of Computing,* 172–184.

HOROWITZ, E. [1972]. "A fast method for interpolation using preconditioning." *Information Processing Letters* **1**:4, 157–163.

HORVATH, E. C. [1974]. "Some efficient stable sorting algorithms," *Proc. 6th Annual ACM Symposium on Theory of Computing,* 194–215.

HU, T. C. [1968]. "A decomposition algorithm for shortest paths in a network." *Operations Res.* **16,** 91–102.

HU, T. C., AND A. C. TUCKER [1971]. "Optimum binary search trees." *SIAM J. Applied Math.* **21**:4, 514–532.

HUNT, H. B., III [1973a]. "On the time and tape complexity of languages," *Proc. 5th Annual ACM Symposium on Theory of Computing,* 10–19. See also TR-73-182, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

HUNT, H. B., III [1973b]. "The equivalence problem for regular expressions with intersection is not polynomial in tape," TR 73-156, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

HUNT, H. B., III [1974]. "Stack languages, Rice's theorem and a natural exponential complexity gap," TR-142, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton University, Princeton, N.J.

HUNT, H. B., III, AND D. J. ROSENKRANTZ [1974]. "Computational parallels between the regular and context-free languages," *Proc. 6th Annual ACM Symposium on Theory of Computing,* 64–74.

IBARRA, O. H. [1972]. "A note concerning nondeterministic tape complexities," *J. ACM* **19**:4, 608–612.

IBARRA, O. H., AND S. K. SAHNI [1973]. "Polynomial complete fault detection problems," Technical Report 74-3, Computer, Information and Control Sciences, University of Minnesota, Minneapolis.

JONES, N. D. [1973]. "Reducibility among combinatorial problems in log *n* space," *Proc. 7th Annual Princeton Conference on Information Sciences and Systems,* 547–551.

JOHNSON, D. B. [1973]. "Algorithms for shortest paths," Ph.D. Thesis, Dept. of Computer Science, Cornell University, Ithaca, New York.

JOHNSON, S. C. [1974]. "Sparse polynomial arithmetic," Bell Laboratories, Murray Hill, N.J.

KARATSUBA, A., AND Y. OFMAN [1962]. "Multiplication of multidigit numbers on automata," *Dokl. Akad. Nauk SSSR* **145,** 293–294 (in Russian).

KASAMI, T. [1965]. "An efficient recognition and syntax algorithm for context-free languages," *Scientific Report AFCRL-65-758,* Air Force Cambridge Research Laboratory, Bedford, Mass.

KARP, R. M. [1972]. "Reducibility among combinatorial problems," in Miller and Thatcher [1972], pp. 85–104.

KARP, R. M., R. E. MILLER, AND A. L. ROSENBERG [1972]. "Rapid identification of repeated patterns in strings, trees, and arrays," *Proc. 4th Annual ACM Symposium on Theory of Computing,* 125–136.

EDEM, Z. [1974]. "On the number of multiplications and divisions required to compute certain rational functions," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 334–341.

ERR, L. R. [1970]. "The effect of algebraic structure on the computational complexity of matrix multiplications," Ph.D. Thesis, Cornell University, Ithaca, N.Y.

IRKPATRICK, D. [1972]. "On the additions necessary to compute certain functions," *Proc. 4th Annual ACM Symposium on Theory of Computing*, 94–101.

IRKPATRICK, D. [1974]. "Determining graph properties from matrix representations," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 84–90.

ISLITSYN, S. S. [1964]. "On the selection of the *k*th element of an ordered set by pairwise comparison," *Sibirsk. Mat. Zh.* **5**, 557–564.

LEENE, S. C. [1956]. "Representation of events in nerve nets and finite automata," in *Automata Studies* (Shannon and McCarthy, eds.), Princeton University Press, pp. 3–40.

NUTH, D. E. [1968]. *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

NUTH, D. E. [1969]. *Seminumerical Algorithms*, Addison-Wesley, Reading, Mass.

NUTH, D. E. [1971]. "Optimum binary search trees," *Acta Informatica* **1**, 14–25.

NUTH, D. E. [1973a]. *Sorting and Searching*, Addison-Wesley, Reading, Mass.

NUTH, D. E. [1973b]. "Notes on pattern matching," University of Trondheim, Norway.

NUTH, D. E., AND V. R. PRATT [1971]. "Automata theory can be useful," Stanford University, Stanford, California.

OHAVI, Z., AND A. PAZ (eds.) [1971]. *Theory of Machines and Computations*, Academic Press, New York.

RUSKAL, J. B., JR. [1956]. "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.* **7**:1, 48–50.

UNG, H. T. [1973]. "Fast evaluation and interpolation," Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh.

EWIS, P. M., II, R. E. STEARNS, AND J. HARTMANIS [1965]. "Memory bounds for recognition of context-free and context-sensitive languages," *Conference Record, IEEE 6th Annual Symposium on Switching Circuit Theory and Logic Design*, 191–202.

IPSON, J. [1971]. "Chinese remainder and interpolation algorithms," *Proc. 2nd Symposium on Symbolic and Algebraic Manipulation*, 372–391.

IU, C. L. [1968]. *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York.

IU, C. L. [1972]. "Analysis and synthesis of sorting algorithms," *SIAM J. Computing* **1**:4, 290–304.

MACLANE, S., AND G. BIRKHOFF [1967]. *Algebra*, Macmillan, New York.

MCNAUGHTON, R., AND H. YAMADA [1960]. "Regular expressions and state graphs for automata," *IRE Trans. on Electronic Computers* **9**:1, 39–47. Reprinted in Moore [1964], pp. 157–174.

MEYER, A. R. [1972]. "Weak monadic second order theory of successor is not elementary recursive," Project MAC Report, MIT, Cambridge, Mass.

MEYER, A. R., AND L. STOCKMEYER [1972]. "The equivalence problem for regular expressions with squaring requires exponential space," *Conference Record, IEEE 13th Annual Symposium on Switching and Automata Theory*, 125–129.

MEYER, A. R., AND L. STOCKMEYER [1973]. "Nonelementary word problems in automata and logic," *Proc. AMS Symposium on Complexity of Computation*, April 1973.

MILLER, R. E., AND J. W. THATCHER (eds.) [1972]. *Complexity of Computer Computations*, Plenum Press, New York.

MINSKY, M. [1967]. *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N.J.

MOENCK, R. [1973]. "Fast Computation of GCD's," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 142–151.

MOENCK, R., AND A. B. BORODIN [1972]. "Fast modular transforms via division," *Conference Record, IEEE 13th Annual Symposium on Switching and Automata Theory*, 90–96.

MOORE, E. F. (ed.) [1964]. *Sequential Machines: Selected Papers*, Addison-Wesley, Reading, Mass.

MORGENSTERN, J. [1973]. "Note on a lower bound of the linear complexity of the fast Fourier transform," *J. ACM* **20**:2, 305–306.

MORRIS, R. [1968]. "Scatter storage techniques," *Comm. ACM* **11**:1, 35–44.

MORRIS, J. H., JR., AND V. R. PRATT [1970]. "A linear pattern matching algorithm," Technical Report No. 40, Computing Center, University of California, Berkeley.

MOTZKIN, T. S. [1955]. "Evaluation of polynomials and evaluation of rational functions," *Bull. Amer. Math. Soc.* **61**, 163.

MUNRO, I. [1971]. "Efficient determination of the transitive closure of a directed graph," *Information Processing Letters* **1**:2, 56–58.

MURAOKA, Y., AND D. J. KUCK [1973]. "On the time required for a sequence of matrix products," *Comm. ACM* **16**:1, 22–26.

NECIPORUK, E. I. [1966]. "A Boolean function," *Dokl. Akad. Nauk SSSR* **169**:4 (in Russian). English translation in *Soviet Math. Dokl.* 7 (1966), 999–1000.

NICHOLSON, P. J. [1971]. "Algebraic theory of finite Fourier transforms," *J. Computer and System Sciences* **5**:5, 524–547.

NIEVERGELT, J., AND E. M. REINGOLD [1973]. "Binary search trees of bounded balance," *SIAM J. Computing* **2**:1, 33–43.

OFMAN, Y. [1962]. "On the algorithmic complexity of discrete functions," *Dokl. Akad. Nauk SSSR* **145**, 48–51 (in Russian). English translation in *Soviet Physics Dokl.* 7 (1963), 589–591.

OSTROWSKI, A. M. [1954]. "On two problems in abstract algebra connected with Horner's rule," *Studies Presented to R. von Mises*, Academic Press, N.Y.

PAN, V. Y. [1966]. "Methods of computing values of polynomials," *Russian Mathematical Surveys* **21**:1, 105–136.

Paterson, M. S., M. J. Fischer, and A. R. Meyer [1974]. "An improved overlap argument for on-line multiplication," Project MAC Technical Memorandum 40 MIT, Cambridge, Mass.

Pohl, I. [1972]. "A sorting problem and its complexity," *Comm. ACM* **15**:6. 462–464.

Pratt, T. W. [1975]. *Programming Language Design and Implementation*, to appear. Prentice-Hall, Englewood Cliffs, N.J.

Pratt, V. R. [1974]. "The power of negative thinking in multiplying Boolean matrices," *Proc. 6th Annual ACM Symposium on Theory of Computing*, 80–83.

Pratt, V. R., and F. F. Yao [1973]. "On lower bounds for computing the *i*th largest element," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 70–81.

Prim, R. C. [1957]. "Shortest connection networks and some generalizations," *Bell System Technical J.*, 1389–1401.

Rabin, M. O. [1963]. "Real-time computation," *Israel J. Math.* **1**, 203–211.

Rabin, M. O. [1972]. "Proving simultaneous positivity of linear forms," *J. Computer and System Sciences* **6**:6, 639–650.

Rabin, M. O., and D. Scott [1959]. "Finite automata and their decision problems," *IBM J. Research and Development* **3**, 114–125. Reprinted in Moore [1964], pp. 63–91.

Rabin, M. O., and S. Winograd [1971]. "Fast evaluation of polynomials by rational preparation," *IBM Technical Report RC 3645*, Yorktown Heights, N.Y.

Rabiner, L. R., and C. M. Rader (eds.) [1972]. *Digital Signal Processing*, IEEE Press, New York.

Rabiner, L. R., R. W. Schafer, and C. M. Rader [1969]. "The chirp *z*-transform and its applications," *Bell System Technical J.* **48**:3, 1249–1292. Reprinted in Rabiner and Rader [1972], pp. 322–328.

Rader, C. M. [1968]. "Discrete Fourier transform when the number of data points is prime," *Proc. IEEE* **56**, 1107–1108.

Reingold, E. M. [1972]. "On the optimality of some set algorithms," *J. ACM* **19**:4, 649–659.

Rogers, H., Jr. [1967]. *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

Rounds, W. C. [1973]. "Complexity of recognition in intermediate level languages," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 145–158.

Runge, C., and H. König [1924]. *Die Grundlehren der mathematischen Wissenschaften*, v. 11, Springer, Berlin.

Sahni, S. K. [1972]. "Some related problems from network flows, game theory and integer programming," *Conference Record, IEEE 13th Annual Symposium on Switching and Automata Theory*, 130–138.

Savitch, W. J. [1970]. "Relationship between nondeterministic and deterministic tape complexities," *J. Computer and System Sciences* **4**:2, 177–192.

SAVITCH, W. J. [1971]. "Maze recognizing automata." *Proc. 4th Annual ACM Symposium on Theory of Computing*, 151–156.

SCHÖNHAGE, A. [1971]. "Schnelle Berechnung von Kettenbruchentwicklungen," *Acta Informatica* **1**, 139–144.

SCHÖNHAGE, A., AND V. STRASSEN [1971]. "Schnelle Multiplikation grosser Zahlen," *Computing* **7**, 281–292.

SEIFERAS, J. J., M. J. FISCHER, AND A. R. MEYER [1973]. "Refinements of nondeterministic time and space hierarchies," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 130–137.

SETHI, R. [1973]. "Complete register allocation problems," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 182–195.

SHEPHERDSON, J. C., AND H. E. STURGIS [1963]. "Computability of recursive functions," *J. ACM* **10**:2, 217–255.

SIEVEKING, M. [1972]. "An algorithm for division of power series," *Computing* **10**, 153–156.

SINGLETON, R. C. [1969]. "Algorithm 347: An algorithm for sorting with minimal storage." *Comm. ACM* **12**:3, 185–187.

SLOANE, N. J. A. [1973]. *A Handbook of Integer Sequences*, Academic Press, N.Y.

SPIRA, P. M. [1973]. "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$," *SIAM J. Computing* **2**:1, 28–32.

SPIRA, P. M., AND A. PAN [1973]. "On finding and updating shortest paths and spanning trees," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 82–84.

STEARNS, R. E., AND D. J. ROSENKRANTZ [1969]. "Table machine simulation," *Conference Record, IEEE 10th Annual Symposium on Switching and Automata Theory*, 118–128.

STOCKMEYER, L. [1973]. "Planar 3-colorability is polynomial complete," *SIGACT News* **5**:3, 19–25.

STOCKMEYER, L., AND A. R. MEYER [1973]. "Word problems requiring exponential time," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 1–9.

STONE, H. S. [1972]. *Introduction to Computer Organization and Data Structures*, McGraw-Hill, New York.

STRASSEN, V. [1969]. "Gaussian elimination is not optimal," *Numerische Mathematik* **13**, 354–356.

STRASSEN, V. [1974]. "Schwer berechenbare Polynome mit rationalen Koeffizienten," *SIAM J. Computing*.

TARJAN, R. E. [1972]. "Depth first search and linear graph algorithms," *SIAM J. Computing* **1**:2, 146–160.

TARJAN, R. E. [1973a]. "Finding dominators in directed graphs," *Proc. 7th Annual Princeton Conference on Information Sciences and Systems*, 414–418.

TARJAN, R. E. [1973b]. "Testing flow graph reducibility," *Proc. 5th Annual ACM Symposium on Theory of Computing*, 96–107.

TARJAN, R. E. [1974]. "On the efficiency of a good but not linear set merging algorithm," *J. ACM*, to appear.

THOMPSON, K. [1968]. "Regular expression search algorithm," *Comm. ACM* **11**:6, 419–422.

TRAUB, J. F. (ed.) [1973]. *Complexity of Sequential and Parallel Numerical Algorithms*, Academic Press, New York.

TURING, A. M. [1936]. "On computable numbers, with an application to the *Entscheidungsproblem*," *Proc. London Mathematical Soc. Ser. 2*, **42**, 230–265. Corrections, *ibid.*, **43** (1937), 544–546.

ULLMAN, J. D. [1973]. "Polynomial complete scheduling problems," *Proc. 4th Symposium on Operating System Principles*, 96–101.

ULLMAN, J. D. [1974]. "Fast algorithms for the elimination of common subexpressions," *Acta Informatica*, **2**:3, 191–213.

VALIANT, L. G. [1974]. "General context-free recognition in less than cubic time." Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, Pa.

VARI, T. M. "Some complexity results for a class of Toeplitz matrices," unpublished memorandum, York Univ., Toronto, Ontario, Canada.

WAGNER, R. A. [1974]. "A shortest path algorithm for edge-sparse graphs." Dept. of Systems and Information Science, Vanderbilt University, Nashville, Tennessee.

WAGNER, R. A., AND M. J. FISCHER [1974]. "The string-to-string correction problem," *J. ACM* **21**:1, 168–173.

WARSHALL, S. [1962]. "A theorem on Boolean matrices," *J. ACM* **9**:1, 11–12.

WEINER, P. [1973]. "Linear pattern matching algorithms," *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, 1–11.

WILLIAMS, J. W. J. [1964]. "Algorithm 232: Heapsort," *Comm. ACM* **7**:6, 347–348.

WINOGRAD, S. [1965]. "On the time required to perform addition," *J. ACM* **12**:2, 277–285.

WINOGRAD, S. [1967]. "On the time required to perform multiplication," *J. ACM* **14**:4, 793–802.

WINOGRAD, S. [1970a]. "On the number of multiplications necessary to compute certain functions," *Comm. Pure and Applied Math.* **23**, 165–179.

WINOGRAD, S. [1970b]. "On the multiplication of 2 × 2 matrices," *IBM Research Report RC267*, January 1970.

WINOGRAD, S. [1970c]. "The algebraic complexity of functions," *Proc. International Congress of Mathematicians* (1970), vol. 3, 283–288.

WINOGRAD, S. [1973]. "Some remarks on fast multiplication of polynomials," in Traub [1973], pp. 181–196.

YOUNGER, D. H. [1967]. "Recognition of context-free languages in time n^3," *Information and Control* **10**:2, 189–208.

INDEX