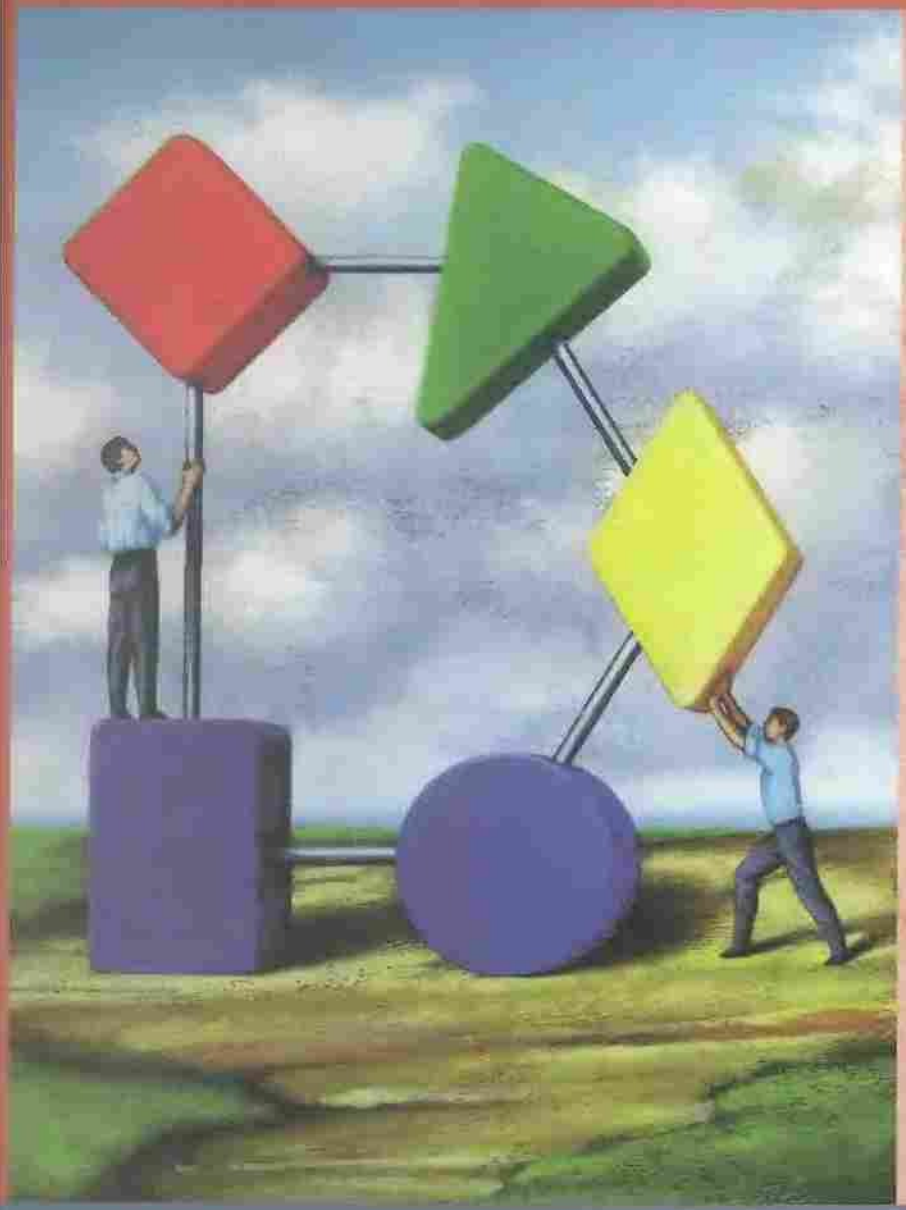
COMPUTER ALGORITHMS



ELLIS HOROWITZ

SARTAJ SAHNI

SANGUTHEVAR RAJASEKARAN

COMPUTER ALGORITHMS

Ellis Horowitz
University of Southern California

Sartaj Sahni
University of Florida

Sanguthevar Rajasekaran
University of Florida

Computer Science Press
An imprint of W. H. Freeman and Company
New York

Acqisitions Editor: Richard Bonacci
Project Editor: Penelope Hull
Text Designer: The Authors
Text Illustrations: The Authors
Cover Designer: A Good Thing
Cover Illustration: Tomek Olbinski
Production Coordinator: Sheila Anderson
Composition: The Authors
Manufacturing: R R Donnelley & Sons Company

Library of Congress Cataloging-in-Publication Data

Horowitz, Ellis.
Computer algorithms / Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran.
p. cm.
Includes bibliographical references and index.
ISBN 0-7167-8316-9
1. Computer algorithms. 2. Pseudocode (Computer program language).
I. Sahni, Sartaj. II. Rajasekaran, Sanguthevar. III. Title.
QA76.9.A43H67 1998
005.1ÑDC21

97-20318
CIP

© 1998 by W. H. Freeman and Company. All rights reserved. No part of this book may be reproduced by any mechanical, photographic, or electronic process, or in the form of a phonographic recording, nor may it be stored in a retrieval system, transmitted, or otherwise copied for public or private use, without written permission from the publisher.

Printed in the United States of America
First printing, 1997

Computer Science Press
An imprint of W. H. Freeman and Company
41 Madison Avenue, New York, New York 10010
Houndmills, Basingstoke RG21 6XS, England

Contents

PREFACE

If we try to identify those contributions of computer science which will be long lasting, surely one of these will be the refinement of the concept called *algorithm*. Ever since man invented the idea of a machine which could perform basic mathematical operations, the study of what can be computed and how it can be done well was launched. This study, inspired by the computer, has led to the discovery of many important algorithms and design methods. The discipline called computer science has embraced the study of algorithms as its own. It is the purpose of this book to organize what is known about them in a coherent fashion so that students and practitioners can learn to devise and analyze new algorithms for themselves.

A book which contains every algorithm ever invented would be exceedingly large. Traditionally, algorithms books proceeded by examining only a small number of problem areas in depth. For each specific problem the most efficient algorithm for its solution is usually presented and analyzed. This approach has one major flaw. Though the student sees many fast algorithms and may master the tools of analysis, she/he remains unconfident about how to devise good algorithms in the first place.

The missing ingredient is a lack of emphasis on *design* techniques. A knowledge of design will certainly help one to create good algorithms, yet without the tools of analysis there is no way to determine the quality of the result. This observation that design should be taught on a par with analysis led us to a more promising line of approach: namely to organize this book around some fundmental strategies of algorithm design. The number of basic design strategies is reasonably small. Moreover all of the algorithms one would typically wish to study can easily be fit into these categories; for example, mergesort and quicksort are perfect examples of the divide-and-conquer strategy while Kruskal's minimum spanning tree algorithm and Dijkstra's single source shortest path algorithm are straight forward examples of the greedy strategy. An understanding of these strategies is an essential first step towards acquiring the skills of design.

Though we strongly feel that the emphasis on design as well as analysis is the appropriate way to organize the study of algorithms, a cautionary remark is in order. First, we have not included every known design principle.

One example is linear programming which is one of the most successful techniques, but is often discussed in a course of its own. Secondly, the student should be inhibited from taking a cookbook approach to algorithm design by assuming that each algorithm must derive from only a single technique. This is not so.

A major portion of this book, Chapters 3 through 9, deal with the different design strategies. First each strategy is described in general terms. Typically a "program abstraction" is given which outlines the form that the computation will take if this strategy can be applied. Following this there are a succession of examples which reveal the intricacies and varieties of the general strategy. The examples are somewhat loosely ordered in terms of increasing complexity. The type of complexity may arise in several ways. Usually we begin with a problem which is very simple to understand and requires no data structures other than a one-dimensional array. For this problem it is usually obvious that the design strategy yields a correct solution. Later examples may require a proof that an algorithm based on this design technique does work. Or, the later algorithms may require more sophisticated data structures (e.g., trees or graphs) and their analyses may be more complex. The major goal of this organization is to emphasize the arts of synthesis and analysis of algorithms. Auxiliary goals are to expose the student to good program structure and to proofs of algorithm correctness.

The algorithms in this book are presented in a pseudocode that resembles C and Pascal. Section 1.2.1 describes the pseudocode conventions. Executable versions (in C++) of many of these algorithms can be found in our home page. Most of the algorithms presented in this book are short and the language constructs used to describe them are simple enough that any one can understand. Chapters 13, 14, and 15 deal with parallel computing.

Another special feature of this book is that we cover the area of randomized algorithms extensively. Many of the algorithms discussed in Chapters 13, 14, and 15 are randomized. Some randomized algorithms are presented in the other chapters as well. An introductory one quarter course on parallel algorithms might cover Chapters 13, 14, and 15 and perhaps some minimal additional material.

We have identified certain sections of the text (indicated with (*)) that are more suitable for advanced courses. We view the material presented in this book as ideal for a one semester or two quarter course given to juniors, seniors, or graduate students. It does require prior experience with programming in a higher level language but everything else is self-contained. Practically speaking, it seems that a course on data structures is helpful, if only for the fact that the students have greater programming maturity. For a school on the quarter system, the first quarter might cover the basic design techniques as given in Chapters 3 through 9: divide-and-conquer, the greedy method, dynamic programming, search and traversal, backtracking, branch-and-bound, and algebraic methods (see TABLE I). The second quarter would cover Chapters 10 through 15: lower bound theory, $\mathcal{NP}$-completeness and

approximation methods, PRAM algorithms, Mesh algorithms and Hypercube algorithms (see TABLE II).

Week	Subject	Reading
1	Introduction	1.1 to 1.3
2	Introduction Data structures	1.4 2.1, 2.2
3	Data structures	2.3 to 2.6
4	Divide-and-conquer	Chapter 3 Assignment I due
5	The greedy method	Chapter 4 Exam I
6	Dynamic programming	Chapter 5
7	Search and traversal techniques	Chapter 6 Assignment II due
8	Backtracking	Chapter 7
9	Branch-and-bound	Chapter 8
10	Algebraic methods	Chapter 9 Assignment III due Exam II

TABLE I: FIRST QUARTER

For a semester schedule where the student has not been exposed to data structures and O-notation, Chapters 1 through 7, 11, and 13 is about the right amount of material (see TABLE III).

A more rigorous pace would cover Chapters 1 to 7, 11, 13, and 14 (see TABLE IV).

An advanced course, for those who have prior knowledge about data structures and O notation, might consist of Chapters 3 to 11, and 13 to 15 (see TABLE V).

Programs for most of the algorithms given in this book are available from the following URL: `http://www.cise.ufl.edu/~raj/BOOK.html`. Please send your comments to `raj@cise.ufl.edu`.

For homework there are numerous exercises at the end of each chapter. The most popular and instructive homework assignment we have found is one which requires the student to execute and time two programs using the same data sets. Since most of the algorithms in this book provide all the implementation details, they can be easily made use of. Translating these algorithms into any programming language should be easy. The problem then reduces to devising suitable data sets and obtaining timing results. The timing results should agree with the asymptotic analysis that was done

Week	Subject	Reading
1	Lower bound theory	10.1 to 10.3
2	Lower bound theory $\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems	10.4 11.1, 11.2
3	$\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems	11.3, 11.4
4	$\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems Approximation algorithms	11.5, 11.6 12.1, 12.2 Assignment I due
5	Approximation algorithms	12.3 to 12.6 Exam I
6	PRAM algorithms	13.1 to 13.4
7	PRAM algorithms	13.5 to 13.9 Assignment II due
8	Mesh algorithms	14.1 to 14.5
9	Mesh algorithms Hypercube algorithms	14.6 to 14.8 15.1 to 15.3
10	Hypercube algorithms	15.4 to 15.8 Assignment III due Exam II

TABLE II: SECOND QUARTER

Week	Subject	Reading
1	Introduction	1.1 to 1.3
2	Introduction Data structures	1.4 2.1, 2.2
3	Data structures	2.3 to 2.6
4	Divide-and-conquer	3.1 to 3.4 Assignment I due
5	Divide-and-conquer	3.5 to 3.7 Exam I
6	The greedy method	4.1 to 4.4
7	The greedy method	4.5 to 4.7 Assignment II due
8	Dynamic programming	5.1 to 5.5
9	Dynamic programming	5.6 to 5.10
10	Search and traversal	6.1 to 6.4 Assignment III due Exam II
11	Backtracking	7.1 to 7.3
12	Backtracking	7.4 to 7.6
13	$\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems	11.1 to 11.3 Assignment IV due
14	$\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems	11.4 to 11.6
15	PRAM algorithms	13.1 to 13.4
16	PRAM algorithms	13.5 to 13.9 Assignment V due Exam III

TABLE III: SEMESTER – Medium pace (no prior exposure)

Week	Subject	Reading
1	Introduction	1.1 to 1.3
2	Introduction Data structures	1.4 2.1, 2.2
3	Data structures	2.3 to 2.6
4	Divide-and-conquer	3.1 to 3.5 Assignment I due
5	Divide-and-conquer The greedy method	3.6 to 3.7 4.1 to 4.3 Exam I
6	The greedy method	4.4 to 4.7
7	Dynamic programming	5.1 to 5.7 Assignment II due
8	Dynamic programming Search and traversal techniques	5.8 to 5.10 6.1 to 6.2
9	Search and traversal techniques Backtracking	6.3, 6.4 7.1, 7.2
10	Backtracking	7.3 to 7.6 Assignment III due Exam II
11	$\mathcal{NP}$-hard and $\mathcal{NP}$-complete problems	11.1 to 11.3
12	$\mathcal{NP}$-hard and $\mathcal{NP}$-complete problems	11.4 to 11.6
13	PRAM algorithms	13.1 to 13.4 Assignment IV due
14	PRAM algorithms	13.5 to 13.9
15	Mesh algorithms	14.1 to 14.3
16	Mesh algorithms	14.4 to 14.8 Assignment V due Exam III

TABLE IV: SEMESTER – Rigorous pace (no prior exposure)

Week	Subject	Reading
1	Divide-and-conquer	3.1 to 3.5
2	Divide-and-conquer The greedy method	3.6, 3.7 4.1 to 4.3
3	The greedy method	4.4 to 4.7
4	Dynamic programming	Chapter 5 Assignment I due
5	Search and traversal techniques	Chapter 6 Exam I
6	Backtracking	Chapter 7
7	Branch-and-bound	Chapter 8 Assignment II due
8	Algebraic methods	Chapter 9
9	Lower bound theory	Chapter 10
10	$\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems	11.1 to 11.3 Exam II Assignment III
11	$\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems	11.4 to 11.6
12	PRAM algorithms	13.1 to 13.4
13	PRAM algorithms	13.5 to 13.9 Assignment IV due
14	Mesh algorithms	14.1 to 14.5
15	Mesh algorithms Hypercube algorithms	14.6 to 14.8 15.1 to 15.3
16	Hypercube algorithms	15.4 to 15.8 Assignment V due Exam III

TABLE V: SEMESTER - Advanced course (rigorous pace)

for the algorithm. This is a nontrivial task which can be both educational and fun. Most importantly it emphasizes an aspect of this field that is often neglected, that there is an experimental side to the practice of algorithms.

Acknowledgements

We are grateful to Martin J. Biernat, Jeff Jenness, Saleem Khan, Ming-Yang Kao, Douglas M. Campbell, and Stephen P. Leach for their critical comments which have immensely enhanced our presentation. We are thankful to the students at UF for pointing out mistakes in earlier versions. We are also thankful to Teo Gonzalez, Danny Krizanc, and David Wei who carefully read portions of this book.

Ellis Horowitz

Sartaj Sahni

Sanguthevar Rajasekaran

June, 1997

Chapter 1

INTRODUCTION

1.1 WHAT IS AN ALGORITHM?

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who wrote a textbook on mathematics. This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique, or method.

Definition 1.1 [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

Criteria 1 and 2 require that an algorithm produce one or more *outputs* and have zero or more *inputs* that are externally supplied. According to criterion 3, each operation must be *definite*, meaning that it must be perfectly clear what should be done. Directions such as "add 6 or 7 to x" or "compute $5/0$" are not permitted because it is not clear which of the two possibilities should be done or what the result is.

The fourth criterion for algorithms we assume in this book is that they *terminate* after a finite number of operations. A related consideration is that the time for termination should be reasonably short. For example, an algorithm could be devised that decides whether any given position in the game of chess is a winning position. The algorithm works by examining all possible moves and countermoves that could be made from the starting position. The difficulty with this algorithm is that even using the most modern computers, it may take billions of years to make the decision. We must be very concerned with analyzing the efficiency of each of our algorithms.

Criterion 5 requires that each operation be *effective*; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithms that are definite and effective are also called *computational procedures*. One important example of computational procedures is the operating system of a digital computer. This procedure is designed to control the execution of jobs, in such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered. Though computational procedures include important examples such as this one, we restrict our study to computational procedures that always terminate.

To help us achieve the criterion of definiteness, algorithms are written in a programming language. Such languages are designed so that each legitimate sentence has a unique meaning. A *program* is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for program. Most readers of this book have probably already programmed and run some algorithms on a computer. This is desirable because before you study a concept in general, it helps if you had some practical experience with it. Perhaps you had some difficulty getting started in formulating an initial solution to a problem, or perhaps you were unable to decide which of two algorithms was better. The goal of this book is to teach you how to make these decisions.

The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify:

1. *How to devise algorithms* — Creating an algorithm is an art which may never be fully automated. A major goal of this book is to study vari-

ous design techniques that have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Many of the chapters of this book are organized around what we believe are the major methods of algorithm design. The reader may now wish to glance back at the table of contents to see what these methods are called. Some of these techniques may already be familiar, and some have been found to be so useful that books have been written about them. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering. In this book we can only hope to give an introduction to these many approaches to algorithm formulation. All of the approaches we consider have applications in a variety of areas including computer science. But some important design techniques such as linear, nonlinear, and integer programming are not covered here as they are traditionally covered in other courses.

2. *How to validate algorithms* — Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as *algorithm validation*. The algorithm need not as yet be expressed as a program. It is sufficient to state it in any precise way. The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as *program proving* or sometimes as *program verification*. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a *specification*, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

3. *How to analyze algorithms* — This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory (both immediate and auxiliary) to hold the program and data. *Analysis of algorithms* or *performance analysis* refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which sometimes requires great mathematical skill. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist.

Questions such as how well does an algorithm perform in the best case, in the worst case, or on the average are typical. For each algorithm in the text, an analysis is also given. Analysis is more fully described in Section 1.3.2.

4. *How to test a program* — Testing a program consists of two phases: debugging and profiling (or performance measurement). *Debugging* is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them. However, as E. Dijkstra has pointed out, "debugging can only point to the presence of errors, but not to their absence." In cases in which we cannot verify the correctness of output on sample data, the following strategy can be employed: let more than one programmer develop programs for the same problem, and compare the outputs produced by these programs. If the outputs match, then there is a good chance that they are correct. A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs. *Profiling* or *performance measurement* is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization. A description of the measurement of timing complexity can be found in Section 1.3.5. For some of the algorithms presented here, we show how to devise a range of data sets that will be useful for debugging and profiling.

These four categories serve to outline the questions we ask about algorithms throughout this book. As we can't hope to cover all these subjects completely, we content ourselves with concentrating on design and analysis, spending less time on program construction and correctness.

EXERCISES

1. Look up the words algorism and algorithm in your dictionary and write down their meanings.

2. The name al-Khowarizmi (algorithm) literally means "from the town of Khowarazm." This city is now known as Khiva, and is located in Uzbekistan. See if you can find this country in an atlas.

3. Use the WEB to find out more about al-Khowarizmi, e.g., his dates, a picture, or a stamp.

1.2 ALGORITHM SPECIFICATION

1.2.1 Pseudocode Conventions

In computational theory, we distinguish between an algorithm and a program. The latter does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called *flowcharts* are another possibility, but they work well only if the algorithm is small and simple. In this text we present most of our algorithms using a pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces: { and }. A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.

3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **record**s. Here is an example:

```
node = record
       {   datatype_1  data_1;
                 .
                 .
                 .
           datatype_n  data_n;
           node        *link;
       }
```

   In this example, *link* is a pointer to the record type *node*. Individual data items of a record can be accessed with $\rightarrow$ and period. For instance if p points to a record of type *node*, $p \rightarrow data\_1$ stands for the value of the first field in the record. On the other hand, if q is a record of type *node*, $q.data\_1$ will denote its first field.

4. Assignment of values to variables is done using the assignment statement

   ⟨*variable*⟩ := ⟨*expression*⟩;

5. There are two boolean values **true** and **false**. In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators $<, \leq, =, \neq, \geq$, and $>$ are provided.

6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j)th element of the array is denoted as $A[i, j]$. Array indices start at zero.

7. The following looping statements are employed: **for**, **while**, and **repeat-until**. The **while** loop takes the following form:

   while ⟨*condition*⟩ **do**
   {
   ⟨*statement* 1⟩
   ⋮
   ⟨*statement n*⟩
   }

   As long as ⟨*condition*⟩ is **true**, the statements get executed. When ⟨*condition*⟩ becomes **false**, the loop is exited. The value of ⟨*condition*⟩ is evaluated at the top of the loop.

   The general form of a **for** loop is

   for *variable* := *value*1 **to** *value*2 **step** *step* **do**
   {
   ⟨*statement* 1⟩
   ⋮
   ⟨*statement n*⟩
   }

   Here *value*1, *value*2, and *step* are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** *step*" is optional and taken as +1 if it does not occur. *step* could either be positive or negative. *variable* is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

variable := *value*1;
fin := *value*2;
incr := *step*;
while ((*variable* − *fin*) ∗ *step* ≤ 0) **do**
{
⟨*statement* 1⟩
⋮
⟨*statement n*⟩
variable := *variable* + *incr*;
}

A **repeat-until** statement is constructed as follows:

repeat
⟨*statement* 1⟩
⋮
⟨*statement n*⟩
until ⟨*condition*⟩

The statements are executed as long as ⟨*condition*⟩ is **false**. The value of ⟨*condition*⟩ is computed after executing the statements.

The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.

8. A conditional statement has the following forms:

if ⟨*condition*⟩ **then** ⟨*statement*⟩
if ⟨*condition*⟩ **then** ⟨*statement* 1⟩ **else** ⟨*statement* 2⟩

Here ⟨*condition*⟩ is a boolean expression and ⟨*statement*⟩, ⟨*statement* 1⟩, and ⟨*statement* 2⟩ are arbitrary statements (simple or compound).

We also employ the following **case** statement:

case
{
:⟨*condition* 1⟩**:** ⟨*statement* 1⟩
⋮
:⟨*condition n*⟩**:** ⟨*statement n*⟩
:else: ⟨*statement n* + 1⟩
}

Here ⟨*statement* 1⟩, ⟨*statement* 2⟩, etc. could be either simple statements or compound statements. A **case** statement is interpreted as follows. If ⟨*condition* 1⟩ is **true**, ⟨*statement* 1⟩ gets executed and the **case** statement is exited. If ⟨*statement* 1⟩ is **false**, ⟨*condition* 2⟩ is evaluated. If ⟨*condition* 2⟩ is **true**, ⟨*statement* 2⟩ gets executed and the **case** statement exited, and so on. If none of the conditions ⟨*condition* 1⟩, ..., ⟨*condition* n⟩ are true, ⟨*statement* $n+1$⟩ is executed and the **case** statement is exited. The **else** clause is optional.

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

    Algorithm *Name* (⟨*parameter list*⟩)

    where *Name* is the name of the procedure and (⟨*parameter list*⟩) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

    As an example, the following algorithm finds and returns the maximum of n given numbers:

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

    In this algorithm (named Max), A and n are procedure parameters. *Result* and i are local variables.

Next we present two examples to illustrate the process of translating a problem into an algorithm.

Example 1.1 [Selection sort] Suppose we must devise an algorithm that sorts a collection of $n \geq 1$ elements of arbitrary type. A simple solution is given by the following

From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm because it leaves several questions unanswered. For example, it does not tell us where and how the elements are initially stored or where we should place the result. We assume that the elements are stored in an array a, such that the ith integer is stored in the ith position $a[i]$, $1 \leq i \leq n$. Algorithm 1.1 is our first attempt at deriving a solution.

```
1   for i := 1 to n do
2   {
3       Examine a[i] to a[n] and suppose
4       the smallest element is at a[j];
5       Interchange a[i] and a[j];
6   }
```

Algorithm 1.1 Selection sort algorithm

To turn Algorithm 1.1 into a pseudocode program, two clearly defined subtasks remain: finding the smallest element (say $a[j]$) and interchanging it with $a[i]$. We can solve the latter problem using the code

$$t := a[i]; \; a[i] := a[j]; \; a[j] := t;$$

The first subtask can be solved by assuming the minimum is $a[i]$, checking $a[i]$ with $a[i+1], a[i+2], \ldots$, and, whenever a smaller element is found, regarding it as the new minimum. Eventually $a[n]$ is compared with the current minimum, and we are done. Putting all these observations together, we get the algorithm SelectionSort (Algorithm 1.2).

The obvious question to ask at this point is, Does SelectionSort work correctly? Throughout this text we use the notation $a[i:j]$ to denote the array elements $a[i]$ through $a[j]$.

Theorem 1.1 Algorithm SelectionSort(a, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1:n]$ such that $a[1] \leq a[2] \leq \cdots \leq a[n]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q, $a[1:q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \cdots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n-1$ without damaging the correctness of the algorithm. □

```
1   Algorithm SelectionSort(a, n)
2   // Sort the array a[1 : n] into nondecreasing order.
3   {
4       for i := 1 to n do
5       {
6           j := i;
7           for k := i + 1 to n do
8               if (a[k] < a[j]) then j := k;
9           t := a[i]; a[i] := a[j]; a[j] := t;
10      }
11  }
```

Algorithm 1.2 Selection sort

1.2.2 Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is *direct recursive*. Algorithm $\mathcal{A}$ is said to be *indirect recursive* if it calls another algorithm which in turn calls $\mathcal{A}$. These recursive mechanisms are extremely powerful, but even more importantly, many times they can express an otherwise complex process very clearly. For these reasons we introduce recursion here.

Typically, beginning programmers view recursion as a somewhat mystical technique that is useful only for some very special class of problems (such as computing factorials or Ackermann's function). This is unfortunate because any algorithm that can be written using assignment, the **if-then-else** statement, and the **while** statement can also be written using assignment, the **if-then-else** statement, and recursion. Of course, this does not say that the resulting algorithm will necessarily be easier to understand. However, there are many instances when this will be the case. When is recursion an appropriate mechanism for algorithm exposition? One instance is when the problem itself is recursively defined. Factorial fits this category, as well as binomial coefficients, where

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$$

The following two examples show how to develop a recursive algorithm. In the first example, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

Example 1.2 [Towers of Hanoi] The Towers of Hanoi puzzle is fashioned after the ancient Tower of Brahma ritual (see Figure 1.1). According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top. Besides this tower there were two other diamond towers (labeled B and C). Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower C for intermediate storage. As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk. According to legend, the world will come to an end when the priests have completed their task.

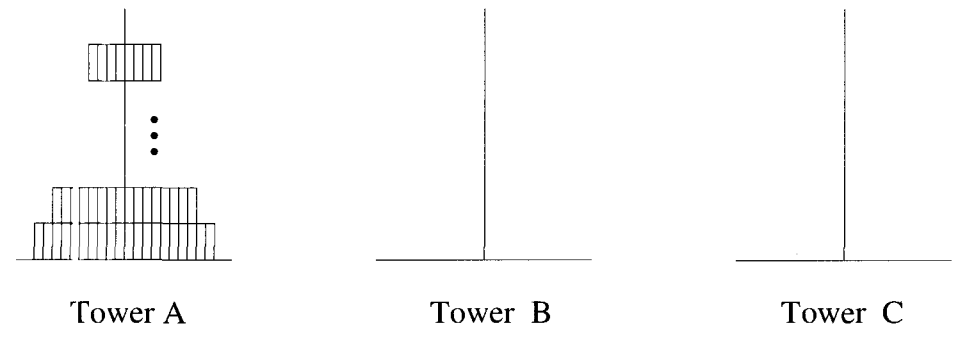


Figure 1.1 Towers of Hanoi

A very elegant solution results from the use of recursion. Assume that the number of disks is n. To get the largest disk to the bottom of tower B, we move the remaining $n-1$ disks to tower C and then move the largest to tower B. Now we are left with the task of moving the disks from tower C to tower B. To do this, we have towers A and B available. The fact that tower B has a disk on it can be ignored as the disk is larger than the disks being moved from tower C and so any disk can be placed on top of it. The recursive nature of the solution is apparent from Algorithm 1.3. This algorithm is invoked by TowersOfHanoi(n,A,B,C). Observe that our solution for an n-disk problem is formulated in terms of solutions to two $(n-1)$-disk problems. □

Example 1.3 [Permutation generator] Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c),$

```
1   Algorithm TowersOfHanoi(n, x, y, z)
2   // Move the top n disks from tower x to tower y.
3   {
4       if (n ≥ 1) then
5       {
6           TowersOfHanoi(n − 1, x, z, y);
7           write ("move top disk from tower", x,
8           "to top of tower", y);
9           TowersOfHanoi(n − 1, z, y, x);
10      }
11  }
```

Algorithm 1.3 Towers of Hanoi

(b, c, a), (c, a, b), $(c, b, a)\}$. It is easy to see that given n elements, there are $n!$ different permutations. A simple algorithm can be obtained by looking at the case of four elements (a, b, c, d). The answer can be constructed by writing

1. a followed by all the permutations of (b, c, d)
2. b followed by all the permutations of (a, c, d)
3. c followed by all the permutations of (a, b, d)
4. d followed by all the permutations of (a, b, c)

The expression "followed by all the permutations" is the clue to recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n - 1$ elements. These considerations lead to Algorithm 1.4, which is invoked by Perm$(a, 1, n)$. Try this algorithm out on sets of length one, two, and three to ensure that you understand how it works. □

EXERCISES

1. Horner's rule is a means for evaluating a polynomial at a point x_0 using a minimum number of multiplications. If the polynomial is $A(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, Horner's rule is

```
1   Algorithm Perm(a, k, n)
2   {
3       if (k = n) then write (a[1 : n]); // Output permutation.
4       else // a[k : n] has more than one permutation.
5           // Generate these recursively.
6           for i := k to n do
7           {
8               t := a[k]; a[k] := a[i]; a[i] := t;
9               Perm(a, k + 1, n);
10              // All permutations of a[k + 1 : n]
11              t := a[k]; a[k] := a[i]; a[i] := t;
12          }
13  }
```

Algorithm 1.4 Recursive permutation generator

$$A(x_0) = (\cdots (a_n x_0 + a_{n-1})x_0 + \cdots + a_1)x_0 + a_0$$

Write an algorithm to evaluate a polynomial using Horner's rule.

2. Given n boolean variables $x_1, x_2, \ldots,$ and x_n, we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: true, true; true, false; false, true; and false, false. Write an algorithm to accomplish this.

3. Devise an algorithm that inputs three integers and outputs them in nondecreasing order.

4. Present an algorithm that searches an unsorted array $a[1 : n]$ for the element x. If x occurs, then return a position in the array; else return zero.

5. The factorial function $n!$ has value 1 when $n \leq 1$ and value $n * (n-1)!$ when $n > 1$. Write both a recursive and an iterative algorithm to compute $n!$.

6. The Fibonacci numbers are defined as $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both a recursive and an iterative algorithm to compute f_i.

7. Give both a recursive and an iterative algorithm to compute the binomial coefficient $\binom{n}{m}$ as defined in Section 1.2.2, where $\binom{n}{0} = \binom{n}{n} = 1$.

8. Ackermann's function $A(m, n)$ is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,\ 1) & \text{if } n = 0 \\ A(m-1,\ A(m,\ n-1)) & \text{otherwise} \end{cases}$$

This function is studied because it grows very fast for small values of m and n. Write a recursive algorithm for computing this function. Then write a nonrecursive algorithm for computing it.

9. The *pigeonhole principle* states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Present an algorithm to find a and b such that $f(a) = f(b)$. Assume that the function inputs are $1, 2, \ldots,$ and n.

10. Give an algorithm to solve the following problem: Given n, a positive integer, determine whether n is the sum of all of its divisors, that is, whether n is the sum of all t such that $1 \leq t < n$, and t divides n.

11. Consider the function $F(x)$ that is defined by "if x is even, then $F(x) = x/2$; else $F(x) = F(F(3x + 1))$." Prove that $F(x)$ terminates for all integers x. (*Hint:* Consider integers of the form $(2i + 1)2^k - 1$ and use induction.)

12. If S is a set of n elements, the *powerset* of S is the set of all possible subsets of S. For example, if $S = (a, b, c)$, then $powerset(S) = \{(\ ), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)\}$. Write a recursive algorithm to compute $powerset(S)$.

1.3 PERFORMANCE ANALYSIS

One goal of this book is to develop skills for making evaluative judgments about algorithms. There are many criteria upon which we can judge an algorithm. For instance:

1. Does it do what we want it to do?

2. Does it work correctly according to the original specifications of the task?

3. Is there documentation that describes how to use it and how it works?

4. Are procedures created in such a way that they perform logical subfunctions?

5. Is the code readable?

These criteria are all vitally important when it comes to writing software, most especially for large systems. Though we do not discuss how to reach these goals, we try to achieve them throughout this book with the pseudocode algorithms we write. Hopefully this more subtle approach will gradually infect your own program-writing habits so that you will automatically strive to achieve these goals.

There are other criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

Definition 1.2 [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. □

Performance evaluation can be loosely divided into two major phases: (1) a priori estimates and (2) a posteriori testing. We refer to these as *performance analysis* and *performance measurement* respectively.

1.3.1 Space Complexity

Algorithm abc (Algorithm 1.5) computes $a+b+b*c+(a+b-c)/(a+b)+4.0$; Algorithm Sum (Algorithm 1.6) computes $\sum_{i=1}^{n} a[i]$ iteratively, where the $a[i]$'s are real numbers; and RSum (Algorithm 1.7) is a recursive algorithm that computes $\sum_{i=1}^{n} a[i]$.

```
1  Algorithm abc(a, b, c)
2  {
3      return a + b + b * c + (a + b − c)/(a + b) + 4.0;
4  }
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

The space needed by each of these algorithms is seen to be the sum of the following components:

```
1  Algorithm Sum(a, n)
2  {
3      s := 0.0;
4      for i := 1 to n do
5          s := s + a[i];
6      return s;
7  }
```

Algorithm 1.6 Iterative function for sum

```
1  Algorithm RSum(a, n)
2  {
3      if (n ≤ 0) then return 0.0;
4      else return RSum(a, n − 1) + a[n];
5  }
```

Algorithm 1.7 Recursive function for sum

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P(\text{instance characteristics})$, where c is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely on estimating $S_P(\text{instance characteristics})$. For any given problem, we need first to determine which instance characteristics to use to measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the algorithm. At times, more complex measures of the interrelationships among the data items are used.

Example 1.4 For Algorithm 1.5, the problem instance is characterized by the specific values of a, b, and c. Making the assumption that one word is adequate to store the values of each of a, b, c, and the result, we see that the space needed by abc is independent of the instance characteristics. Consequently, $S_P(\text{instance characteristics}) = 0$. □

Example 1.5 The problem instances for Algorithm 1.6 are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type *integer*. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{\mathsf{Sum}}(n) \geq (n+3)$ (n for $a[\,]$, one each for n, i, and s). □

Example 1.6 Let us consider the algorithm RSum (Algorithm 1.7). As in the case of Sum, the instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\,]$). Since the depth of recursion is $n+1$, the recursion stack space needed is $\geq 3(n+1)$. □

1.3.2 Time Complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This run time is denoted by t_P(instance characteristics).

Because many of the factors t_P depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_P. If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P. So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

Obtaining such an exact formula is in itself an impossible task, since the time needed for an addition, subtraction, multiplication, and so on, often depends on the numbers being added, subtracted, multiplied, and so on. The value of $t_P(n)$ for any given n can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. The execution time is physically clocked, and $t_P(n)$ obtained. Even with this experimental approach, one could face difficulties. In a multiuser system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these other programs, and so on.

Given the minimal utility of determining the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by n, we might as well lump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

$$\textbf{return } a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

of Algorithm 1.5 could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true, since the time for a multiply and divide generally depends on the numbers involved in the operation).

The number of steps any program statement is assigned depends on the kind of statement. For example, comments count as zero steps; an assignment statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the **for**, **while**, and **repeat-until** statements, we consider the step counts only for the control part of the statement. The control parts for **for** and **while** statements have the following forms:

for $i :=$ ⟨*expr*⟩ **to** ⟨*expr*1⟩ **do**

while (⟨*expr*⟩) **do**

Each execution of the control part of a **while** statement is given a step count equal to the number of step counts assignable to ⟨*expr*⟩. The step count for each execution of the control part of a **for** statement is one, unless the counts attributable to ⟨*expr*⟩ and ⟨*expr*1⟩ are functions of the instance characteristics. In this latter case, the first execution of the control part of the **for** has a step count equal to the sum of the counts for ⟨*expr*⟩ and ⟨*expr*1⟩ (note that these expressions are computed only when the loop is started). Remaining executions of the **for** statement have a step count of one; and so on.

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways. In the first method, we introduce a new variable, *count*, into the program. This is a global variable with initial value 0. Statements to increment *count* by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, *count* is incremented by the step count of that statement.

Example 1.7 When the statements to increment *count* are introduced into Algorithm 1.6, the result is Algorithm 1.8. The change in the value of *count* by the time this program terminates is the number of steps executed by Algorithm 1.6.

Since we are interested in determining only the change in the value of *count*, Algorithm 1.8 may be simplified to Algorithm 1.9. For every initial value of *count*, Algorithms 1.8 and 1.9 compute the same final value for *count*. It is easy to see that in the **for** loop, the value of *count* will increase by a total of $2n$. If *count* is zero to start with, then it will be $2n + 3$ on termination. So each invocation of Sum (Algorithm 1.6) executes a total of $2n + 3$ steps. □

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       count := count + 1; // count is global; it is initially zero.
5       for i := 1 to n do
6       {
7           count := count + 1; // For for
8           s := s + a[i]; count := count + 1; // For assignment
9       }
10      count := count + 1; // For last time of for
11      count := count + 1; // For the return
12      return s;
13  }
```

Algorithm 1.8 Algorithm 1.6 with count statements added

```
1   Algorithm Sum(a, n)
2   {
3       for i := 1 to n do count := count + 2;
4       count := count + 3;
5   }
```

Algorithm 1.9 Simplified version of Algorithm 1.8

Example 1.8 When the statements to increment *count* are introduced into Algorithm 1.7, Algorithm 1.10 is obtained. Let $t_{\mathsf{RSum}}(n)$ be the increase in the value of *count* when Algorithm 1.10 terminates. We see that $t_{\mathsf{RSum}}(0) = 2$. When $n > 0$, *count* increases by 2 plus whatever increase results from the invocation of RSum from within the **else** clause. From the definition of t_{RSum}, it follows that this additional increase is $t_{\mathsf{RSum}}(n-1)$. So, if the value of *count* is zero initially, its value at the time of termination is $2+t_{\mathsf{RSum}}(n-1)$, $n > 0$.

```
1   Algorithm RSum(a, n)
2   {
3       count := count + 1; // For the if conditional
4       if (n ≤ 0) then
5       {
6           count := count + 1; // For the return
7           return 0.0;
8       }
9       else
10      {
11          count := count + 1;  // For the addition, function
12                               // invocation and return
13          return RSum(a, n − 1) + a[n];
14      }
15  }
```

Algorithm 1.10 Algorithm 1.7 with count statements added

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as *recurrence relations*. One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrences disappear:

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad\qquad n \geq 0 \end{aligned}$$

So the step count for RSum (Algorithm 1.7) is $2n + 2$. □

The step count is useful in that it tells us how the run time for a program changes with changes in the instance characteristics. From the step count for Sum, we see that if n is doubled, the run time also doubles (approximately); if n increases by a factor of 10, the run time increases by a factor of 10; and so on. So, the run time grows *linearly* in n. We say that Sum is a linear time algorithm (the time complexity is linear in the instance characteristic n).

Definition 1.3 [Input size] One of the instance characteristics that is frequently used in the literature is the *input size*. The input size of any instance of a problem is defined to be the number of words (or the number of elements) needed to describe that instance. The input size for the problem of summing an array with n elements is $n + 1$, n for listing the n elements and 1 for the value of n (Algorithms 1.6 and 1.7). The problem tackled in Algorithm 1.5 has an input size of 3. If the input to any problem instance is a single element, the input size is normally taken to be the number of bits needed to specify that element. Run times for many of the algorithms presented in this text are expressed as functions of the corresponding input sizes. □

Example 1.9 [Matrix addition] Algorithm 1.11 is to add two $m \times n$ matrices a and b together. Introducing the *count*-incrementing statements leads to Algorithm 1.12. Algorithm 1.13 is a simplified version of Algorithm 1.12 that computes the same value for *count*. Examining Algorithm 1.13, we see that line 7 is executed n times for each value of i, or a total of mn times; line 5 is executed m times; and line 9 is executed once. If *count* is 0 to begin with, it will be $2mn + 2m + 1$ when Algorithm 1.13 terminates.

From this analysis we see that if $m > n$, then it is better to interchange the two **for** statements in Algorithm 1.11. If this is done, the step count becomes $2mn+2n+1$. Note that in this example the instance characteristics are given by m and n and the input size is $2mn + 2$. □

The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of

```
1   Algorithm Add(a, b, c, m, n)
2   {
3       for i := 1 to m do
4           for j := 1 to n do
5               c[i, j] := a[i, j] + b[i, j];
6   }
```

Algorithm 1.11 Matrix addition

```
1   Algorithm Add(a, b, c, m, n)
2   {
3       for i := 1 to m do
4       {
5           count := count + 1; // For 'for i'
6           for j := 1 to n do
7           {
8               count := count + 1; // For 'for j'
9               c[i, j] := a[i, j] + b[i, j];
10              count := count + 1; // For the assignment
11          }
12          count := count + 1;// For loop initialization and
13                             // last time of 'for j'
14      }
15      count := count + 1;    // For loop initialization and
16                             // last time of 'for i'
17  }
```

Algorithm 1.12 Matrix addition with counting statements

```
1   Algorithm Add(a, b, c, m, n)
2   {
3       for i := 1 to m do
4       {
5           count := count + 2;
6           for j := 1 to n do
7               count := count + 2;
8       }
9       count := count + 1;
10  }
```

Algorithm 1.13 Simplified algorithm with counting only

steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed. *The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.* By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

In Table 1.1, the number of steps per execution and the frequency of each of the statements in Sum (Algorithm 1.6) have been listed. The total number of steps required by the algorithm is determined to be $2n + 3$. It is important to note that the frequency of the **for** statement is $n + 1$ and not n. This is so because i has to be incremented to $n + 1$ before the **for** loop can terminate.

Table 1.2 gives the step count for RSum (Algorithm 1.7). Notice that under the s/e (steps per execution) column, the **else** clause has been given a count of $1 + t_{\text{RSum}}(n - 1)$. This is the total cost of this line each time it is executed. It includes all the steps that get executed as a result of the invocation of RSum from the **else** clause. The frequency and total steps columns have been split into two parts: one for the case $n = 0$ and the other for the case $n > 0$. This is necessary because the frequency (and hence total steps) for some statements is different for each of these cases.

Table 1.3 corresponds to algorithm Add (Algorithm 1.11). Once again, note that the frequency of the first **for** loop is $m + 1$ and not m. This is so as i needs to be incremented up to $m + 1$ before the loop can terminate. Similarly, the frequency for the second **for** loop is $m(n + 1)$.

When you have obtained sufficient experience in computing step counts, you can avoid constructing the frequency table and obtain the step count as in the following example.

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

Table 1.1 Step table for Algorithm 1.6

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 **Algorithm** RSum(a, n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n - 1) + a[n]$;	$1 + x$	0	1	0	$1 + x$
7 }	0	–	–	0	0
Total				2	$2 + x$

$x = t_{\mathsf{RSum}}(n - 1)$

Table 1.2 Step table for Algorithm 1.7

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i,j] := a[i,j] + b[i,j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

Table 1.3 Step table for Algorithm 1.11

Example 1.10 [Fibonacci numbers] The Fibonacci sequence of numbers starts as

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence f_0, then $f_0 = 0$, $f_1 = 1$, and in general

$$f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$$

Fibonacci (Algorithm 1.14) takes as input any nonnegative integer n and prints the value f_n.

To analyze the time complexity of this algorithm, we need to consider the two cases (1) $n = 0$ or 1 and (2) $n > 1$. When $n = 0$ or 1, lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2. When $n > 1$, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n - 1$ times each (note that the last time line 9 is executed, i is incremented to $n+1$, and the loop exited). Line 8 has an s/e of 2, line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1. The total steps for the case $n > 1$ is therefore $4n + 1$. □

Summary of Time Complexity

The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for. The number of steps is itself a function of the instance characteristics. Although any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number

```
1   Algorithm Fibonacci(n)
2   // Compute the nth Fibonacci number.
3   {
4       if (n ≤ 1) then
5           write (n);
6       else
7       {
8           fnm2 := 0; fnm1 := 1;
9           for i := 2 to n do
10          {
11              fn := fnm1 + fnm2;
12              fnm2 := fnm1; fnm1 := fn;
13          }
14          write (fn);
15      }
16  }
```

Algorithm 1.14 Fibonacci numbers

of steps is computed as a function of some subset of these. Usually, we choose those characteristics that are of importance to us. For example, we might wish to know how the computing (or run) time (i.e., time complexity) increases as the number of inputs increase. In this case the number of steps will be computed as a function of the number of inputs alone. For a different algorithm, we might be interested in determining how the computing time increases as the magnitude of one of the inputs increases. In this case the number of steps will be computed as a function of the magnitude of this input alone. Thus, before the step count of an algorithm can be determined, we need to know exactly which characteristics of the problem instance are to be used. These define the variables in the expression for the step count. In the case of Sum, we chose to measure the time complexity as a function of the number n of elements being added. For algorithm Add, the choice of characteristics was the number m of rows and the number n of columns in the matrices being added.

Once the relevant characteristics $(n, m, p, q, r, \ldots)$ have been selected, we can define what a step is. A *step* is any computation unit that is independent of the characteristics $(n, m, p, q, r, \ldots)$. Thus, 10 additions can be one step; 100 multiplications can also be one step; but n additions cannot. Nor can $m/2$ additions, $p + q$ subtractions, and so on, be counted as one step.

A systematic way to assign step counts was also discussed. Once this has been done, the time complexity (i.e., the total step count) of an algorithm can be obtained using either of the two methods discussed.

The examples we have looked at so far were sufficiently simple that the time complexities were nice functions of fairly simple characteristics like the number of inputs and the number of rows and columns. For many algorithms, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic. For example, the searching algorithm you wrote for Exercise 4 in Section 1.2, may terminate in one step if x is the first element examined by your algorithm, or it may take two steps (this happens if x is the second element examined), and so on. In other words, knowing n alone is not enough to estimate the run time of your algorithm.

We can extricate ourselves from the difficulties resulting from situations when the chosen parameters are not adequate to determine the step count uniquely by defining three kinds of step counts: best case, worst case, and average. The *best-case step count* is the minimum number of steps that can be executed for the given parameters. The *worst-case step count* is the maximum number of steps that can be executed for the given parameters. The *average step count* is the average number of steps executed on instances with the given parameters.

Our motivation to determine step counts is to be able to compare the time complexities of two algorithms that compute the same function and also to predict the growth in run time as the instance characteristics change.

Determining the exact step count (best case, worst case, or average) of an algorithm can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly is not a very worthwhile endeavor, since the notion of a step is itself inexact. (Both the instructions $x := y$; and $x := y + z + (x/y) + (x * y * z - x/z)$; count as one step.) Because of the inexactness of what a step stands for, the exact step count is not very useful for comparative purposes. An exception to this is when the difference between the step counts of two algorithms is very large, as in $3n + 3$ versus $100n + 10$. We might feel quite safe in predicting that the algorithm with step count $3n+3$ will run in less time than the one with step count $100n+10$. But even in this case, it is not necessary to know that the exact step count is $100n + 10$. Something like, "it's about $80n$ or $85n$ or $75n$," is adequate to arrive at the same conclusion.

For most situations, it is adequate to be able to make a statement like $c_1n^2 \leq t_P(n) \leq c_2n^2$ or $t_Q(n,m) = c_1n + c_2m$, where c_1 and c_2 are nonnegative constants. This is so because if we have two algorithms with a complexity of $c_1n^2 + c_2n$ and c_3n respectively, then we know that the one with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$ for sufficiently large values of n. For small values of n, either algorithm could be faster (depending on c_1, c_2, and c_3). If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$, then

$c_1n^2 + c_2n \leq c_3n$ for $n \leq 98$ and $c_1n^2 + c_2n > c_3n$ for $n > 98$. If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1n^2 + c_2n \leq c_3n$ for $n \leq 998$.

No matter what the values of c_1, c_2, and c_3, there will be an n beyond which the algorithm with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$. This value of n will be called the *break-even point.* If the break-even point is zero, then the algorithm with complexity c_3n is always faster (or at least as fast). The exact break-even point cannot be determined analytically. The algorithms have to be run on a computer in order to determine the break-even point. To know that there is a break-even point, it is sufficient to know that one algorithm has complexity $c_1n^2 + c_2n$ and the other c_3n for some constants c_1, c_2, and c_3. There is little advantage in determining the exact values of c_1, c_2, and c_3.

1.3.3 Asymptotic Notation (O, Ω, Θ)

With the previous discussion as motivation, we introduce some terminology that enables us to make meaningful (but inexact) statements about the time and space complexities of an algorithm. In the remainder of this chapter, the functions f and g are nonnegative functions.

Definition 1.4 [Big "oh"] The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$. □

Example 1.11 The function $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$. $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$. $100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for all $n \geq 6$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$ for $n \geq 100$. $6 * 2^n + n^2 = O(2^n)$ as $6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 4$. $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all $n \geq n_0$. $10n^2 + 4n + 2 \neq O(n)$. □

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called *linear*, $O(n^2)$ is called *quadratic*, $O(n^3)$ is called *cubic*, and $O(2^n)$ is called *exponential.* If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times–$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$–are the ones we see most often in this book.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$. It does not say anything about how good this bound is. Notice

that $n = O(2^n)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, and so on. For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$. So, while we often say that $3n + 3 = O(n)$, we almost never say that $3n + 3 = O(n^2)$, even though this latter statement is correct.

From the definition of O, it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol = is unfortunate because this symbol commonly denotes the equals relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol = as "is" and not as "equals."

Theorem 1.2 obtains a very useful result concerning the order of $f(n)$ (that is, the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n.

Theorem 1.2 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$\begin{aligned} f(n) &\leq \textstyle\sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \textstyle\sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \textstyle\sum_{i=0}^{m} |a_i| \qquad \text{for } n \geq 1 \end{aligned}$$

So, $f(n) = O(n^m)$ (assuming that m is fixed). □

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$. □

Example 1.12 The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$). $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$. $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$, $10n^2 + 4n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(1)$, $6 * 2^n + n^2 = \Omega(n^{100})$, $6 * 2^n + n^2 = \Omega(n^{50.2})$, $6 * 2^n + n^2 = \Omega(n^2)$, $6 * 2^n + n^2 = \Omega(n)$, and $6 * 2^n + n^2 = \Omega(1)$. □

As in the case of the big oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we say that $3n + 3 = \Omega(n)$ and $6 * 2^n + n^2 = \Omega(2^n)$, we almost never say that $3n + 3 = \Omega(1)$ or $6 * 2^n + n^2 = \Omega(1)$, even though both of these statements are correct.

Theorem 1.3 is the analogue of Theorem 1.2 for the omega notation.

Theorem 1.3 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof: Left as an exercise. □

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$. □

Example 1.13 The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

The theta notation is more precise than both the the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples have been 1. This is in accordance with practice. We almost never find ourselves saying that $3n + 3 = O(3n)$, that $10 = O(100)$, that $10n^2 + 4n + 2 = \Omega(4n^2)$, that $6 * 2^n + n^2 = O(6 * 2^n)$, or that $6 * 2^n + n^2 = \Theta(4 * 2^n)$, even though each of these statements is true.

Theorem 1.4 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof: Left as an exercise. □

Definition 1.7 [Little "oh"] The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

□

Example 1.14 The function $3n + 2 = o(n^2)$ since $\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \neq o(n)$. $6 * 2^n + n^2 \neq o(2^n)$. □

Analogous to o is the notation ω defined as follows.

Definition 1.8 [Little omega] The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

□

Example 1.15 Let us reexamine the time complexity analyses of the previous section. For the algorithm Sum (Algorithm 1.6) we determined that $t_{\mathsf{Sum}}(n) = 2n + 3$. So, $t_{\mathsf{Sum}}(n) = \Theta(n)$. For Algorithm 1.7, $t_{\mathsf{RSum}}(n) = 2n + 2 = \Theta(n)$. □

Although we might all see that the O, Ω, and Θ notations have been used correctly in the preceding paragraphs, we are still left with the question, Of what use are these notations if we have to first determine the step count exactly? The answer to this question is that the asymptotic complexity (i.e., the complexity in terms of O, Ω, and Θ) can be determined quite easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement (or group of statements) in the algorithm and then adding these complexities. Tables 1.4 through 1.6 do just this for Sum, RSum, and Add (Algorithms 1.6, 1.7, and 1.11).

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 $s := 0.0$;	1	1	$\Theta(1)$
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$\Theta(n)$
5 $s := s + a[i]$;	1	n	$\Theta(n)$
6 **return** s;	1	1	$\Theta(1)$
7 }	0	–	$\Theta(0)$
Total			$\Theta(n)$

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

Although the analyses of Tables 1.4 through 1.6 are carried out in terms of step counts, it is correct to interpret $t_P(n) = \Theta(g(n))$, $t_P(n) = \Omega(g(n))$, or $t_P(n) = O(g(n))$ as a statement about the computing time of algorithm P. This is so because each step takes only $\Theta(1)$ time to execute.

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a, n)	0	–	–	0	$\Theta(0)$
2 {	0	–	–	0	$\Theta(0)$
3 **if** $(n \leq 0)$ **then**	1	1	1	1	$\Theta(1)$
4 **return** 0.0;	1	1	0	1	$\Theta(0)$
5 **else return**					
6 RSum$(a, n-1) + a[n]$);	$1+x$	0	1	0	$\Theta(1+x)$
7 }	0	–	–	0	$\Theta(0)$
Total				2	$\Theta(1+x)$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table 1.5 Asymptotic complexity of RSum (Algorithm 1.7).

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 **for** $i := 1$ **to** m **do**	1	$\Theta(m)$	$\Theta(m)$
4 **for** $i := 1$ **to** n **do**	1	$\Theta(mn)$	$\Theta(mn)$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	$\Theta(mn)$	$\Theta(mn)$
6 }	0	–	$\Theta(0)$
Total			$\Theta(mn)$

Table 1.6 Asymptotic complexity of Add (Algorithm 1.11)

After you have had some experience using the table method, you will be in a position to arrive at the asymptotic complexity of an algorithm by taking a more global approach. We elaborate on this method in the following examples.

Example 1.16 [Permutation generator] Consider Perm (Algorithm 1.4). When $k = n$, we see that the time taken is $\Theta(n)$. When $k < n$, the **else** clause is entered. At this time, the second **for** loop is entered $n - k + 1$ times. Each iteration of this loop takes $\Theta(n + t_{\mathsf{Perm}}(k+1, n))$ time. So, $t_{\mathsf{Perm}}(k,\ n) = \Theta((n-k+1)(n+t_{\mathsf{Perm}}(k+1,\ n)))$ when $k < n$. Since $t_{\mathsf{Perm}}(k+1,\ n)$ is at least n when $k+1 \leq n$, we get $t_{\mathsf{Perm}}(k,\ n) = \Theta((n-k+1)t_{\mathsf{Perm}}(k+1,\ n))$ for $k < n$. Using the substitution method, we obtain $t_{\mathsf{Perm}}(1, n) = \Theta(n(n!))$, $n \geq 1$. □

Example 1.17 [Magic square] The next example we consider is a problem from recreational mathematics. A magic square is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same. Figure 1.2 gives an example magic square for the case $n = 5$. In this example, the common sum is 65.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.2 Example magic square

H. Coxeter has given the following simple rule for generating a magic square when n is odd:

> Start with 1 in the middle of the top row; then go up and left, assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue.

The magic square of Figure 1.2 was formed using this rule. Algorithm 1.15 is for creating an $n \times n$ magic square for the case in which n is odd. This results from Coxeter's rule.

The magic square is represented using a two-dimensional array having n rows and n columns. For this application it is convenient to number the rows (and columns) from 0 to $n-1$ rather than from 1 to n. Thus, when the algorithm "falls off the square," the **mod** operator sets i and/or j back to 0 or $n-1$.

The time to initialize and output the square is $\Theta(n^2)$. The third **for** loop (in which *key* ranges over 2 through n^2) is iterated n^2-1 times and each iteration takes $\Theta(1)$ time. So, this **for** loop takes $\Theta(n^2)$ time. Hence the overall time complexity of Magic is $\Theta(n^2)$. Since there are n^2 positions in which the algorithm must place a number, we see that $\Theta(n^2)$ is the best bound an algorithm for the magic square problem can have. □

Example 1.18 [Computing x^n] Our final example is to compute x^n for any real number x and integer $n \geq 0$. A naive algorithm for solving this problem is to perform $n-1$ multiplications as follows:

power := x;
for $i := 1$ **to** $n-1$ **do** *power* := *power* $* x$;

This algorithm takes $\Theta(n)$ time. A better approach is to employ the "repeated squaring" trick. Consider the special case in which n is an integral power of 2 (that is, in which n equals 2^k for some integer k). The following algorithm computes x^n.

power := x;
for $i := 1$ **to** k **do** *power* := *power*^{2};

The value of *power* after q iterations of the **for** loop is x^{2^q}. Therefore, this algorithm takes only $\Theta(k) = \Theta(\log n)$ time, which is a significant improvement over the run time of the first algorithm.

Can the same algorithm be used when n is not an integral power of 2? Fortunately, the answer is yes. Let $b_k b_{k-1} \cdots b_1 b_0$ be the binary representation of the integer n. This means that $n = \sum_{q=0}^{k} b_q 2^q$. Now,

$$x^n = x^{\sum_{q=0}^{k} b_q 2^q} = (x)^{b_0} * (x^2)^{b_1} * (x^4)^{b_2} * \cdots * (x^{2^k})^{b_k}$$

Also observe that b_0 is nothing but n **mod** 2 and that $\lfloor n/2 \rfloor$ is $b_k b_{k-1} \cdots b_1$ in binary form. These observations lead us to Exponentiate (Algorithm 1.16) for computing x^n.

```
1   Algorithm Magic(n)
2   // Create a magic square of size n, n being odd.
3   {
4       if ((n mod 2) = 0) then
5       {
6           write ("n is even"); return;
7       }
8       else
9       {
10          for i := 0 to n − 1 do  // Initialize square to zero.
11              for j := 0 to n − 1 do  square[i, j] := 0;
12          square[0, (n − 1)/2] := 1; // Middle of first row
13          // (i, j) is the current position.
14          j := (n − 1)/2;
15          for key := 2 to n^2 do
16          {
17              // Move up and left. The next two if statements
18              // may be replaced by the mod operator if
19              // −1 mod n has the value n − 1.
20              if (i ≥ 1) then k := i − 1; else k := n − 1;
21              if (j ≥ 1) then l := j − 1; else l := n − 1;
22              if (square[k, l] ≥ 1) then i := (i + 1) mod n;
23              else // square[k,l] is empty.
24              {
25                  i := k; j := l;
26              }
27              square[i, j] := key;
28          }
29          // Output the magic square.
30          for i := 0 to n − 1 do
31              for j := 0 to n − 1 do write (square[i, j]);
32      }
33  }
```

Algorithm 1.15 Magic square

```
1   Algorithm Exponentiate(x, n)
2   // Return x^n for an integer n ≥ 0.
3   {
4       m := n; power := 1; z := x;
5       while (m > 0) do
6       {
7           while ((m mod 2) = 0) do
8           {
9               m := ⌊m/2⌋; z := z^2;
10          }
11          m := m − 1; power := power * z;
12      }
13      return power;
14  }
```

Algorithm 1.16 Computation of x^n

Proving the correctness of this algorithm is left as an exercise. The variable m starts with the value of n, and after every iteration of the innermost **while** loop (line 7), its value decreases by a factor of at least 2. Thus there will be only $\Theta(\log n)$ iterations of the **while** loop of line 7. Each such iteration takes $\Theta(1)$ time. Whenever control exits from the innermost **while** loop, the value of m is odd and the instructions $m := m - 1$; $power := power * z$; are executed once. After this execution, since m becomes even, either the innermost **while** loop is entered again or the outermost **while** loop (line 5) is exited (in case $m = 0$). Therefore the instructions $m := m - 1$; $power := power * z$; can only be executed $O(\log n)$ times. In summary, the overall run time of Exponentiate is $\Theta(\log n)$. □

1.3.4 Practical Complexities

We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function can also be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. We can assert that algorithm P is faster than algorithm Q for sufficiently large n. To see the validity of this assertion, observe that the computing time of P is bounded

from above by cn for some constant c and for all n, $n \geq n_1$, whereas that of Q is bounded from below by dn^2 for some constant d and all n, $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm P is faster than algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

You should always be cautiously aware of the presence of the phrase "sufficiently large" in an assertion like that of the preceding discussion. When deciding which of the two algorithms to use, you must know whether the n you are dealing with is, in fact, sufficiently large. If algorithm P runs in $10^6 n$ milliseconds, whereas algorithm Q runs in n^2 milliseconds, and if you always have $n \leq 10^6$, then, other factors being equal, algorithm Q is the one to use.

To get a feel for how the various functions grow with n, you are advised to study Table 1.7 and Figure 1.3 very closely. It is evident from Table 1.7 and Figure 1.3 that the function 2^n grows very rapidly with n. In fact, if an algorithm needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing one billion steps per second, this would require about 18.3 minutes. If $n = 50$, the same algorithm would run for about 13 days on this computer. When $n = 60$, about 310.56 years are required to execute the algorithm and when $n = 100$, about $4 * 10^{13}$ years are needed. So, we may conclude that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Table 1.7 Function values

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility. For example, if an algorithm needs n^{10} steps, then using our 1-billion-steps-per-second computer, we need 10 seconds when $n = 10$, 3171 years when $n = 100$, and $3.17 * 10^{13}$ years when $n = 1000$. If the algorithm's complexity had been n^3 steps instead, then we would need one second when $n = 1000$, 110.67 minutes when $n = 10{,}000$, and 11.57 days when $n = 100{,}000$.

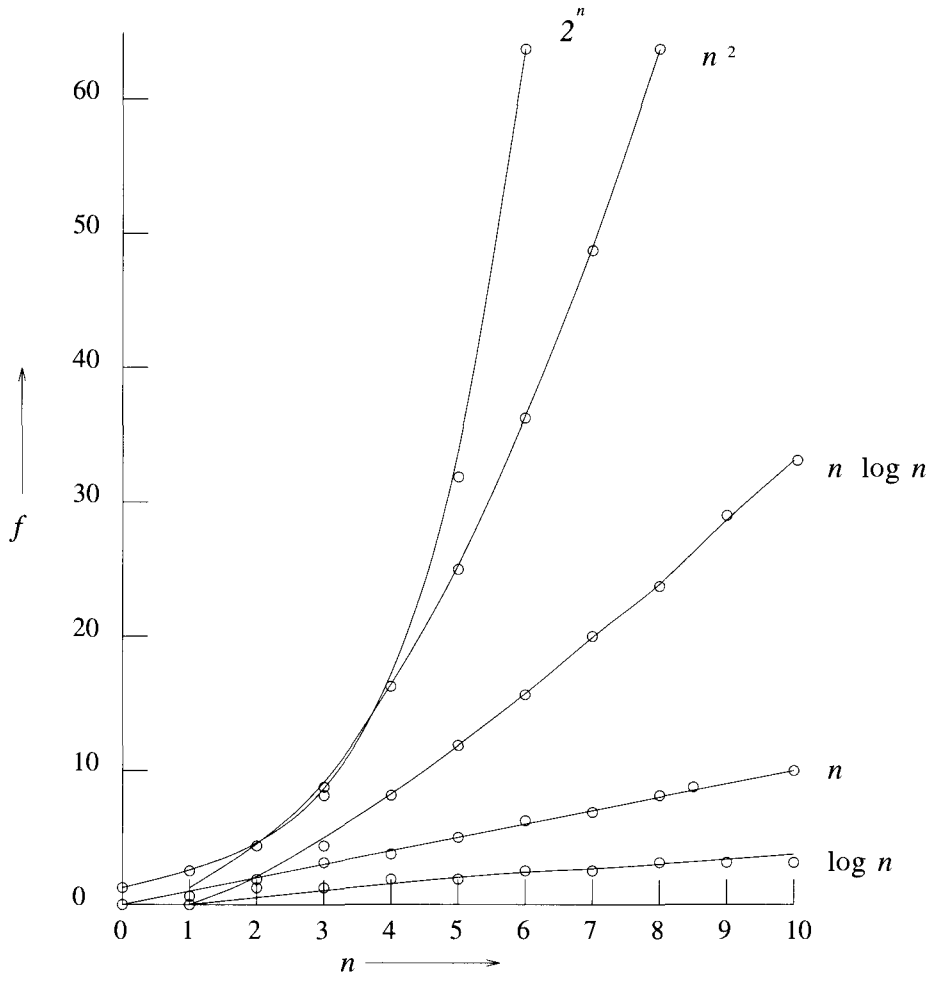


Figure 1.3 Plot of function values

Table 1.8 gives the time needed by a one-billion-steps-per-second computer to execute an algorithm of complexity $f(n)$ instructions. You should note that currently only the fastest computers can execute about 1 billion instructions per second. From a practical standpoint, it is evident that for reasonably large n (say $n > 100$), only algorithms of small complexity (such as n, $n \log n$, n^2, and n^3) are feasible. Further, this is the case even if you could build a computer capable of executing 10^{12} instructions per second. In this case, the computing times of Table 1.8 would decrease by a factor of 1000. Now, when $n = 100$, it would take 3.17 years to execute n^{10} instructions and $4 * 10^{10}$ years to execute 2^n instructions.

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n) = n$	$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = n^4$	$f(n) = n^{10}$	$f(n) = 2^n$
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 hr	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121.36 d	18.3 min
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 yr	13 d
100	.1 μs	.66 μs	10 μs	1 ms	100 ms	3171 yr	$4*10^{13}$ yr
1,000	1 μs	9.96 μs	1 ms	1 s	16.67 min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10 μs	130 μs	100 ms	16.67 min	115.7 d	$3.17*10^{23}$ yr	
100,000	100 μs	1.66 ms	10 s	11.57 d	3171 yr	$3.17*10^{33}$ yr	
1,000,000	1 ms	19.92 ms	16.67 min	31.71 yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

Table 1.8 Times on a 1-billion-steps-per-second computer

1.3.5 Performance Measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on the computer on which the algorithm is run. Unless otherwise stated, all performance values provided in this book are obtained using the Gnu C++ compiler, the default compiler options, and the Sparc 10/30 computer workstation.

In keeping with the discussion of the preceding section, we do not concern ourselves with the space and time needed for compilation. We justify this by the assumption that each program (after it has been fully debugged) is compiled once and then executed several times. Certainly, the space and time needed for compilation are important during program testing, when more time is spent on this task than in running the compiled code.

We do not consider measuring the run-time space requirements of a program. Rather, we focus on measuring the computing time of a program. To obtain the computing (or run) time of a program, we need a clocking procedure. We assume the existence of a program GetTime() that returns the current time in milliseconds.

Suppose we wish to measure the worst-case performance of the sequential search algorithm (Algorithm 1.17). Before we can do this, we need to (1) decide on the values of n for which the times are to be obtained and (2) determine, for each of the above values of n, the data that exhibit the worst-case behavior.

```
1  Algorithm SeqSearch(a, x, n)
2  // Search for x in a[1 : n]. a[0] is used as additional space.
3  {
4      i := n; a[0] := x;
5      while (a[i] ≠ x) do i := i − 1;
6      return i;
7  }
```

Algorithm 1.17 Sequential search

The decision on which values of n to use is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for Algorithm 1.17, our intent is simply to predict how long it will take, in the worst case, to search for x, given the size n of a. An asymptotic analysis reveals that this time is $\Theta(n)$. So, we expect a plot of the times to be a straight line. Theoretically, if we know the times for any two values of n, the straight line is determined, and we can obtain the time for all other values of n from this line. In practice, we need the times for more than two values of n. This is so for the following reasons:

1. Asymptotic analysis tells us the behavior only for sufficiently large values of n. For smaller values of n, the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n.

2. Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve (straight line in the case of Algorithm 1.17) because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, an algorithm with asymptotic complexity $\Theta(n)$ can have time complexity $c_1 n + c_2 \log n + c_3$ or, for that matter, any other function of n in which the highest-order term is $c_1 n$ for some constant c_1, $c_1 > 0$.

It is reasonable to expect that the asymptotic behavior of Algorithm 1.17 begins for some n that is smaller than 100. So, for $n > 100$, we obtain the

run time for just a few values. A reasonable choice is $n = 200, 300, 400, \ldots, 1000$. There is nothing magical about this choice of values. We can just as well use $n = 500, 1,000, 1,500, \ldots, 10,000$ or $n = 512, 1,024, 2,048, \ldots, 2^{15}$. It costs us more in terms of computer time to use the latter choices, and we probably do not get any better information about the run time of Algorithm 1.17 using these choices.

For n in the range [0, 100] we carry out a more-refined measurement, since we are not quite sure where the asymptotic behavior begins. Of course, if our measurements show that the straight-line behavior does not begin in this range, we have to perform a more-detailed measurement in the range [100, 200], and so on, until the onset of this behavior is detected. Times in the range [0, 100] are obtained in steps of 10 beginning at $n = 0$.

Algorithm 1.17 exhibits its worst-case behavior when x is chosen such that it is not one of the $a[i]$'s. For definiteness, we set $a[i] = i$, $1 \leq i \leq n$, and $x = 0$. At this time, we envision using an algorithm such as Algorithm 1.18 to obtain the worst-case times.

```
1   Algorithm TimeSearch()
2   {
3       for j := 1 to 1000 do  a[j] := j;
4       for j := 1 to 10 do
5       {
6           n[j] := 10 * (j - 1); n[j + 10] := 100 * j;
7       }
8       for j := 1 to 20 do
9       {
10          h := GetTime();
11          k := SeqSearch(a, 0, n[j]);
12          h1 := GetTime();
13          t := h1 - h;
14          write (n[j], t);
15      }
16  }
```

Algorithm 1.18 Algorithm to time Algorithm 1.17

The timing results of this algorithm is summarized in Table 1.9. The times obtained are too small to be of any use to us. Most of the times are zero; this indicates that the precision of our clock is inadequate. The nonzero times are just noise and are not representative of the time taken.

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

Table 1.9 Timing results of Algorithm 1.18. Times are in milliseconds.

To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

Since our clock has an accuracy of about one-tenth of a second, we should not attempt to time any single event that takes less than about one second. With an event time of at least ten seconds, we can expect our observed times to be accurate to one percent.

The body of Algorithm 1.18 needs to be changed to that of Algorithm 1.19. In this algorithm, $r[i]$ is the number of times the search is to be repeated when the number of elements in the array is $n[i]$. Notice that rearranging the timing statements as in Algorithm 1.20 or 1.21 does not produce the desired results. For instance, from the data of Table 1.9, we expect that with the structure of Algorithm 1.20, the value output for $n = 0$ will still be 0. This is because there is a chance that in every iteration of the **for** loop, the clock does not change between the two times GetTime() is called. With the structure of Algorithm 1.21, we expect the algorithm never to exit the **while** loop when $n = 0$ (in reality, the loop will be exited because occasionally the measured time will turn out to be a few milliseconds).

Yet another alternative is shown in Algorithm 1.22. This approach can be expected to yield satisfactory times. It cannot be used when the timing procedure available gives us only the time since the last invocation of GetTime. Another difficulty is that the measured time includes the time needed to read the clock. For small n, this time may be larger than the time to run SeqSearch. This difficulty can be overcome by determining the time taken by the timing procedure and subtracting this time later.

```
1   Algorithm TimeSearch()
2   {
3       // Repetition factors
4       r[21] := {0, 200000, 200000, 150000, 100000, 100000, 100000,
5           50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
6           50000, 50000, 25000, 25000, 25000, 25000};
7       for j := 1 to 1000 do a[j] := j;
8       for j := 1 to 10 do
9       {
10          n[j] := 10 * (j - 1); n[j + 10] := 100 * j;
11      }
12      for j := 1 to 20 do
13      {
14          h := GetTime();
15          for i := 1 to r[j] do k := SeqSearch(a, 0, n[j]);
16          h1 := GetTime();
17          t1 := h1 - h;
18          t := t1; t := t/r[j];
19          write (n[j], t1, t);
20      }
21  }
```

Algorithm 1.19 Timing algorithm

```
1   t := 0;
2   for i := 1 to r[j] do
3   {
4       h := GetTime();
5       k := SeqSearch(a, 0, n[j]);
6       h1 := GetTime();
7       t := t + h1 - h;
8   }
9   t := t/r[j];
```

Algorithm 1.20 Improper timing construct

```
1   t := 0;
2   while (t < DESIRED_TIME) do
3   {
4       h := GetTime();
5       k := SeqSearch(a, 0, n[j]);
6       h1 := GetTime();
7       t := t + h1 − h;
8   }
```

Algorithm 1.21 Another improper timing construct

```
1   h := GetTime(); t := 0;
2   while (t < DESIRED_TIME) do
3   {
4       k := SeqSearch(a, 0, n[j]);
5       h1 := GetTime();
6       t := h1 − h;
7   }
```

Algorithm 1.22 An alternate timing construct

Timing results of Algorithm 1.19, is given in Table 1.10. The times for n in the range [0, 1000] are plotted in Figure 1.4. Values in the range [10, 100] have not been plotted. The linear dependence of the worst-case time on n is apparent from this graph.

n	$t1$	t	n	$t1$	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7725	0.309

Times are in milliseconds

Table 1.10 Worst-case run times for Algorithm 1.17

The graph of Figure 1.4 can be used to predict the run time for other values of n. We can go one step further and get the equation of the straight line. The equation of this line is $t = c + mn$, where m is the slope and c the value for $n = 0$. From the graph, we see that $c = 0.002$. Using the point $n = 600$ and $t = 0.186$, we obtain $m = (t - c)/n = 0.184/600 = 0.0003067$. So the line of Figure 1.4 has the equation $t = 0.002 + 0.0003067n$, where t is the time in milliseconds. From this, we expect that when $n = 1000$, the worst-case search time will be 0.3087 millisecond, and when $n = 500$, it will be 0.155 millisecond. Compared to the observed times of Table 1.10, we see that these figures are very accurate!

Summary of Running Time Calculation

To obtain the run time of a program, we need to plan the experiment. The following issues need to be addressed during the planning stage:

1. What is the accuracy of the clock? How accurate do our results have to be? Once the desired accuracy is known, we can determine the length of the shortest event that should be timed.

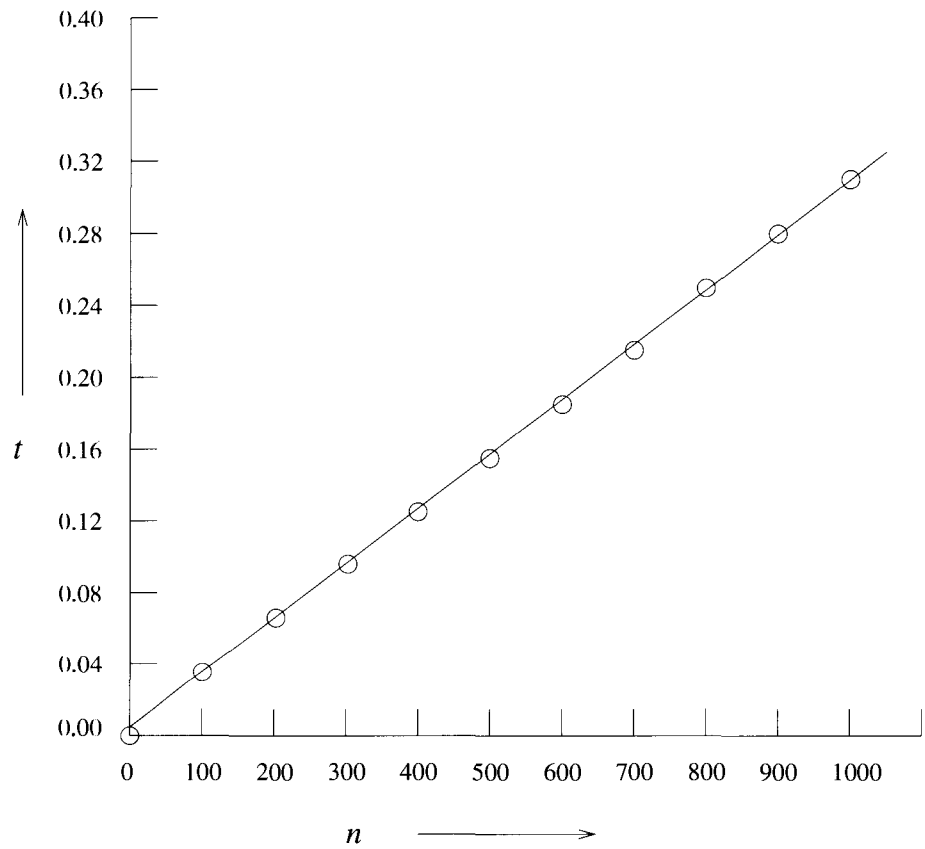


Figure 1.4 Plot of the data in Table 1.10

2. For each instance size, a repetition factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.

3. Are we measuring worst-case or average performance? Suitable test data need to be generated.

4. What is the purpose of the experiment? Are the times being obtained for comparative purposes, or are they to be used to predict run times? If the latter is the case, then contributions to the run time from such sources as the repetition loop and data generation need to be subtracted (in case they are included in the measured time). If the former is the case, then these times need not be subtracted (provided they are the same for all programs being compared).

5. In case the times are to be used to predict run times, then we need to fit a curve through the points. For this, the asymptotic complexity should be known. If the asymptotic complexity is linear, then a least-squares straight line can be fit; if it is quadratic, then a parabola can be used (that is, $t = a_0 + a_1 n + a_2 n^2$). If the complexity is $\Theta(n \log n)$, then a least-squares curve of the form $t = a_0 + a_1 n + a_2 n \log_2 n$ can be fit. When obtaining the least-squares approximation, one should discard data corresponding to small values of n, since the program does not exhibit its asymptotic behavior for these n.

Generating Test Data

Generating a data set that results in the worst-case performance of an algorithm is not always easy. In some cases, it is necessary to use a computer program to generate the worst-case data. In other cases, even this is very difficult. In these cases, another approach to estimating worst-case performance is taken. For each set of values of the instance characteristics of interest, we generate a suitably large number of random test data. The run times for each of these test data are obtained. The maximum of these times is used as an estimate of the worst-case time for this set of values of the instance characteristics.

To measure average-case times, it is usually not possible to average over all possible instances of a given characteristic. Although it is possible to do this for sequential search, it is not possible for a sort algorithm. If we assume that all keys are distinct, then for any given n, $n!$ different permutations need to be used to obtain the average time. Obtaining average-case data is usually much harder than obtaining worst-case data. So, we often adopt the strategy outlined above and simply obtain an estimate of the average time on a suitable set of test data.

Whether we are estimating worst-case or average time using random data, the number of instances that we can try is generally much smaller than the total number of such instances. Hence, it is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment. This is a very algorithm-specific task, and we do not go into it here.

EXERCISES

1. Compare the two functions n^2 and $2^n/4$ for various values of n. Determine when the second becomes larger than the first.

2. Prove by induction:

   (a) $\sum_{i=1}^{n} i = n(n+1)/2,\ n \geq 1$

   (b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6,\ n \geq 1$

   (c) $\sum_{i=0}^{n} x^i = (x^{n+1}-1)/(x-1),\ x \neq 1,\ n \geq 0$

3. Determine the frequency counts for all statements in the following two algorithm segments:

```
1  for i := 1 to n do
2      for j := 1 to i do
3          for k := 1 to j do
4              x := x + 1;
```

(a)

```
1  i := 1;
2  while (i ≤ n) do
3  {
4      x := x + 1;
5      i := i + 1;
6  }
```

(b)

4. (a) Introduce statements to increment *count* at all appropriate points in Algorithm 1.23.

   (b) Simplify the resulting algorithm by eliminating statements. The simplified algorithm should compute the same value for *count* as computed by the algorithm of part (a).

   (c) What is the exact value of *count* when the algorithm terminates? You may assume that the initial value of *count* is 0.

   (d) Obtain the step count for Algorithm 1.23 using the frequency method. Clearly show the step count table.

5. Do Exercise 4 for Transpose (Algorithm 1.24).

6. Do Exercise 4 for Algorithm 1.25. This algorithm multiplies two $n \times n$ matrices a and b.

```
1   Algorithm D(x, n)
2   {
3       i := 1;
4       repeat
5       {
6           x[i] := x[i] + 2; i := i + 2;
7       } until (i > n);
8       i := 1;
9       while (i ≤ ⌊n/2⌋) do
10      {
11          x[i] := x[i] + x[i + 1]; i := i + 1;
12      }
13  }
```

Algorithm 1.23 Example algorithm

```
1   Algorithm Transpose(a, n)
2   {
3       for i := 1 to n − 1 do
4           for j := i + 1 to n do
5           {
6               t := a[i, j]; a[i, j] := a[j, i]; a[j, i] := t;
7           }
8   }
```

Algorithm 1.24 Matrix transpose

```
1   Algorithm Mult(a, b, c, n)
2   {
3       for i := 1 to n do
4           for j := 1 to n do
5           {
6               c[i, j] := 0;
7               for k := 1 to n do
8                   c[i, j] := c[i, j] + a[i, k] * b[k, j];
9           }
10  }
```

Algorithm 1.25 Matrix multiplication

7. (a) Do Exercise 4 for Algorithm 1.26. This algorithm multiplies two matrices a and b, where a is an $m \times n$ matrix and b is an $n \times p$ matrix.

```
1   Algorithm Mult(a, b, c, m, n, p)
2   {
3       for i := 1 to m do
4           for j := 1 to p do
5           {
6               c[i, j] := 0;
7               for k := 1 to n do
8                   c[i, j] := c[i, j] + a[i, k] * b[k, j];
9           }
10  }
```

Algorithm 1.26 Matrix multiplication

(b) Under what conditions is it profitable to interchange the two outermost **for** loops?

8. Show that the following equalities are correct:

(a) $5n^2 - 6n = \Theta(n^2)$

(b) $n! = O(n^n)$

(c) $2n^2 2^n + n \log n = \Theta(n^2 2^n)$

(d) $\sum_{i=0}^{n} i^2 = \Theta(n^3)$

(e) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$.

(f) $n^{2^n} + 6 * 2^n = \Theta(n^{2^n})$

(g) $n^3 + 10^6 n^2 = \Theta(n^3)$

(h) $6n^3/(\log n + 1) = O(n^3)$

(i) $n^{1.001} + n \log n = \Theta(n^{1.001})$

(j) $n^{k+\epsilon} + n^k \log n = \Theta(n^{k+\epsilon})$ for all fixed k and ϵ, $k \geq 0$ and $\epsilon > 0$

(k) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$

(l) $33n^3 + 4n^2 = \Omega(n^2)$

(m) $33n^3 + 4n^2 = \Omega(n^3)$

9. Show that the following equalities are incorrect:

   (a) $10n^2 + 9 = O(n)$

   (b) $n^2 \log n = \Theta(n^2)$

   (c) $n^2/\log n = \Theta(n^2)$

   (d) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

10. Prove Theorems 1.3 and 1.4.

11. Analyze the computing time of SelectionSort (Algorithm 1.2).

12. Obtain worst-case run times for SelectionSort (Algorithm 1.2). Do this for suitable values of n in the range [0, 100]. Your report must include a plan for the experiment as well as the measured times. These times are to be provided both in a table and as a graph.

13. Consider the algorithm Add (Algorithm 1.11).

    (a) Obtain run times for $n = 1, 10, 20, \ldots, 100$.

    (b) Plot the times obtained in part (a).

14. Do the previous exercise for matrix multiplication (Algorithm 1.26).

15. A complex-valued matrix X is represented by a pair of matrices (A, B), where A and B contain real values. Write an algorithm that computes the product of two complex-valued matrices (A, B) and (C, D), where $(A, B) * (C, D) = (A + iB) * (C + iD) = (AC - BD) + i(AD + BC)$. Determine the number of additions and multiplications if the matrices are all $n \times n$.

1.4 RANDOMIZED ALGORITHMS

1.4.1 Basics of Probability Theory

Probability theory has the goal of characterizing the outcomes of natural or conceptual "experiments." Examples of such experiments include tossing a coin ten times, rolling a die three times, playing a lottery, gambling, picking a ball from an urn containing white and red balls, and so on.

Each possible outcome of an experiment is called a *sample point* and the set of all possible outcomes is known as the *sample space* S. In this text we assume that S is finite (such a sample space is called a *discrete sample space*). An *event* E is a subset of the sample space S. If the sample space consists of n sample points, then there are 2^n possible events.

Example 1.19 [Tossing three coins] When a coin is tossed, there are two possible outcomes: heads (H) and tails (T). Consider the experiment of throwing three coins. There are eight possible outcomes: HHH, HHT, HTH, HTT, THH, THT, TTH, and TTT. Each such outcome is a sample point. The sets $\{HHT, HTT, TTT\}$, $\{HHH, TTT\}$, and $\{\ \}$ are three possible events. The third event has no sample points and is the empty set. For this experiment there are 2^8 possible events. □

Definition 1.9 [Probability] The probability of an event E is defined to be $\frac{|E|}{|S|}$, where S is the sample space. □

Example 1.20 [Tossing three coins] The probability of the event $\{HHT, HTT, TTT\}$ is $\frac{3}{8}$. The probability of the event $\{HHH, TTT\}$ is $\frac{2}{8}$ and that of the event $\{\ \}$ is zero. □

Note that the probability of S, the sample space, is 1.

Example 1.21 [Rolling two dice] Let us look at the experiment of rolling two (six-faced) dice. There are 36 possible outcomes some of which are $(1,1), (1,2), (1,3), \ldots$. What is the probability that the sum of the two faces is 10? The event that the sum is 10 consists of the following sample points: $(1,9), (2,8), (3,7), (4,6), (5,5), (6,4), (7,3), (8,2)$, and $(9,1)$. Therefore, the probability of this event is $\frac{9}{36} = \frac{1}{4}$. □

Definition 1.10 [Mutual exclusion] Two events E_1 and E_2 are said to be *mutually exclusive* if they do not have any common sample points, that is, if $E_1 \cap E_2 = \Phi$. □

Example 1.22 [Tossing three coins] When we toss three coins, let E_1 be the event that there are two H's and let E_2 be the event that there are at least two T's. These two events are mutually exclusive since there are no common sample points. On the other hand, if E_2' is defined to be the event that there is at least one T, then E_1 and E_2' will *not* be mutually exclusive since they will have THH, HTH, and HHT as common sample points. □

The probability of event E is denoted as $Prob.[E]$. The *complement* of E, denoted $\bar{E}$, is defined to be $S - E$. If E_1 and E_2 are two events, the probability of E_1 or E_2 or both happening is denoted as $Prob.[E_1 \cup E_2]$. The probability of both E_1 and E_2 occurring at the same time is denoted as $Prob.[E_1 \cap E_2]$. The corresponding event is $E_1 \cap E_2$.

Theorem 1.5

$$\begin{array}{rrcl} 1. & Prob.[\bar{E}] & = & 1 - Prob.[E]. \\ 2. & Prob.[E_1 \cup E_2] & = & Prob.[E_1] + Prob.[E_2] - Prob.[E_1 \cap E_2] \\ & & \leq & Prob.[E_1] + Prob.[E_2] \end{array}$$

Definition 1.11 [Conditional probability] Let E_1 and E_2 be any two events of an experiment. The *conditional probability of* E_1 *given* E_2, denoted by $Prob.\ [E_1|E_2]$, is defined as $\frac{Prob.[E_1 \cap E_2]}{Prob.[E_2]}$. □

Example 1.23 [Tossing four coins] Consider the experiment of tossing four coins. Let E_1 be the event that the number of H's is even and let E_2 be the event that there is at least one H. Then, E_2 is the complement of the event that there are no H's. The probability of no H's is $\frac{1}{16}$. Therefore, $Prob.[E_2] = 1 - \frac{1}{16} = \frac{15}{16}$. $Prob.[E_1 \cap E_2]$ is $\frac{7}{16}$ since the event $E_1 \cap E_2$ has the seven sample points $HHHH$, $HHTT$, $HTHT$, $HTTH$, $THHT$, $THTH$, and $TTHH$. Thus, $Prob.[E_1|E_2]$ is $\frac{7/16}{15/16} = \frac{7}{15}$. □

Definition 1.12 [Independence] Two events E_1 and E_2 are said to be *independent* if $Prob.[E_1 \cap E_2] = Prob.[E_1] * Prob.[E_2]$. □

Example 1.24 [Rolling a die twice] Intuitively, we say two events E_1 and E_2 are independent if the probability of one event happening is in no way affected by the occurrence of the other event. In other words, if $Prob.[E_1|E_2] = Prob.[E_1]$, these two events are independent. Suppose we roll a die twice. What is the probability that the outcome of the second roll is 5 (call this event E_1), given that the outcome of the first roll is 4 (call this event E_2)? The answer is $\frac{1}{6}$ no matter what the outcome of the first roll is. In this case E_1 and E_2 are independent. Therefore, $Prob.[E_1 \cap E_2] = \frac{1}{6} * \frac{1}{6} = \frac{1}{36}$. □

Example 1.25 [Flipping a coin 100 times] If a coin is flipped 100 times what is the probability that all of the outcomes are tails? The probability that the first outcome is T is $\frac{1}{2}$. Since the outcome of the second flip is independent of the outcome of the first flip, the probability that the first two outcomes are T's can be obtained by multiplying the corresponding probabilities to get $\frac{1}{4}$. Extending the argument to all 100 outcomes, we conclude that the probability of obtaining 100 T's is $\left(\frac{1}{2}\right)^{100}$. In this case we say the outcomes of the 100 coin flips are *mutually independent.* □

Definition 1.13 [Random variable] Let S be the sample space of an experiment. A *random variable* on S is a function that maps the elements of S to the set of real numbers. For any sample point $s \in S$, $X(s)$ denotes the image of s under this mapping. If the range of X, that is, the set of values X can take, is finite, we say X is *discrete.*

Let the range of a discrete random variable X be $\{r_1, r_2, \ldots, r_m\}$. Then, $Prob.[X = r_i]$, for any i, is defined to be the the number of sample points whose image is r_i divided by the number of sample points in S. In this text we are concerned mostly with discrete random variables. □

Example 1.26 We flip a coin four times. The sample space consists of 2^4 sample points. We can define a random variable X on S as the number of heads in the coin flips. For this random variable, then, $X(HTHH) = 3$, $X(HHHH) = 4$, and so on. The possible values that X can take are 0, 1, 2, 3, and 4. Thus X is discrete. $Prob.[X = 0]$ is $\frac{1}{16}$, since the only sample point whose image is 0 is $TTTT$. $Prob.[X = 1]$ is $\frac{4}{16}$, since the four sample points $HTTT$, $THTT$, $TTHT$, and $TTTH$ have 1 as their image. □

Definition 1.14 [Expected value] If the sample space of an experiment is $S = \{s_1, s_2, \ldots, s_n\}$, the *expected value* or the *mean* of any random variable X is defined to be $\sum_{i=1}^{n} Prob.[s_i] * X(s_i) = \frac{1}{n} \sum_{i=1}^{n} X(s_i)$. □

Example 1.27 [Coin tosses] The sample space corresponding to the experiment of tossing three coins is $S = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}$. If X is the number of heads in the coin flips, then the expected value of X is $\frac{1}{8}(3 + 2 + 2 + 1 + 2 + 1 + 1 + 0) = 1.5$. □

Definition 1.15 [Probability distribution] Let X be a discrete random variable defined over the sample space S. Let $\{r_1, r_2, \ldots, r_m\}$ be its range. Then, the *probability distribution* of X is the sequence $Prob.[X = r_1]$, $Prob.[X = r_2]$, ... , $Prob.[X = r_m]$. Notice that $\sum_{i=1}^{m} Prob.[X = r_i] = 1$. □

Example 1.28 [Coin tosses] If a coin is flipped three times and X is the number of heads, then X can take on four values, 0, 1, 2, and 3. The probability distribution of X is given by $Prob.[X = 0] = \frac{1}{8}$, $Prob.[X = 1] = \frac{3}{8}$, $Prob.[X = 2] = \frac{3}{8}$, and $Prob.[X = 3] = \frac{1}{8}$. □

Definition 1.16 [Binomial distribution] A *Bernoulli* trial is an experiment that has two possible outcomes, namely, *success* and *failure*. The probability of success is p. Consider the experiment of conducting the Bernoulli trial n times. This experiment has a sample space S with 2^n sample points. Let X be a random variable on S defined to be the numbers of successes in the n trials. The variable X is said to have a *binomial distribution* with parameters $(n,\ p)$. The expected value of X is np. Also,

$$Prob.[X = i] \ = \ \binom{n}{i} p^i (1-p)^{n-i}$$

□

In several applications, it is necessary to estimate the probabilities at the tail ends of probability distributions. One such estimate is provided by the following lemma.

Lemma 1.1 [Markov's inequality] If X is any nonnegative random variable whose mean is μ, then

$$Prob.[X \geq x] \ \leq \ \frac{\mu}{x}$$

□

Example 1.29 Let μ be the mean of a random variable X. We can use Markov's lemma (also called Markov's inequality) to make the following statement: "The probability that the value of X exceeds 2μ is $\leq \frac{1}{2}$." Consider the example: if we toss a coin 1000 times, what is the probability that the number of heads is ≥ 600? If X is the number of heads in 1000 tosses, then, the expected value of X, $E[X]$, is 500. Applying Markov's inequality with $x = 600$ and $\mu = 500$, we infer that $P[X \geq 600] \leq \frac{5}{6}$. □

Though Markov's inequality can be applied to any nonnegative random variable, it is rather weak. We can obtain tighter bounds for a number of important distributions including the binomial distribution. These bounds are due to Chernoff. Chernoff bounds as applied to the binomial distribution are employed in this text to analyze randomized algorithms.

Lemma 1.2 [Chernoff bounds] If X is a binomial with parameters $(n,\ p)$, and $m > np$ is an integer, then

$$Prob.(X \geq m) \ \leq \ \left(\frac{np}{m}\right)^m e^{(m-np)}. \tag{1.1}$$

$$\text{Also, } \ Prob.(X \leq \lfloor (1-\epsilon)pn \rfloor) \ \leq \ e^{(-\epsilon^2 np/2)} \tag{1.2}$$

$$\text{and } \ Prob.(X \geq \lceil (1+\epsilon)np \rceil) \ \leq \ e^{(-\epsilon^2 np/3)} \tag{1.3}$$

for all $0 < \epsilon < 1$. □

Example 1.30 Consider the experiment of tossing a coin 1000 times. We want to determine the probability that the number X of heads is ≥ 600. We can use Equation 1.3 to estimate this probability. The value for ϵ here is 0.2. Also, $n = 1000$ and $p = \frac{1}{2}$. Equation 1.3 now becomes

$$P[X \geq 600] \leq e^{[-(0.2)^2(500/3)]} = e^{-20/3} \leq 0.001273$$

This estimate is more precise than that given by Markov's inequality. □

1.4.2 Randomized Algorithms: An Informal Description

A randomized algorithm is one that makes use of a randomizer (such as a random number generator). Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. The execution time of a randomized algorithm could also vary from run to run for the same input.

Randomized algorithms can be categorized into two classes: The first is algorithms that always produce the same (correct) output for the same input. These are called *Las Vegas* algorithms. The execution time of a Las Vegas algorithm depends on the output of the randomizer. If we are lucky, the algorithm might terminate fast, and if not, it might run for a longer period of time. In general the execution time of a Las Vegas algorithm is characterized as a random variable (see Section 1.4.1 for a definition). The second is algorithms whose outputs might differ from run to run (for the same input). These are called *Monte Carlo* algorithms. Consider any problem for which there are only two possible answers, say, yes and no. If a Monte Carlo algorithm is employed to solve such a problem, then the algorithm might give incorrect answers depending on the output of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low. Typically, for a fixed input, a Monte Carlo algorithm does not display

much variation in execution time between runs, whereas in the case of a Las Vegas algorithm this variation is significant.

We can think of a randomized algorithm with one possible randomizer output to be different from the same algorithm with a different possible randomizer output. Therefore, a randomized algorithm can be viewed as a family of algorithms. For a given input, some of the algorithms in this family may run for indefinitely long periods of time (or may give incorrect answers). The objective in the design of a randomized algorithm is to ensure that the number of such bad algorithms in the family is only a small fraction of the total number of algorithms. If for *any* input we can show that at least $1-\epsilon$ (ϵ being very close to 0) fraction of algorithms in the family will run quickly (respectively give the correct answer) on that input, then clearly, a random algorithm in the family will run quickly (or output the correct answer) on any input with probability $\geq 1-\epsilon$. In this case we say that this family of algorithms (or this randomized algorithm) runs quickly (respectively gives the correct answer) with probability at least $1-\epsilon$, where ϵ is called the *error probability.*

Definition 1.17 [The $\widetilde{O}()$] Just like the $O()$ notation is used to characterize the run times of non randomized algorithms, $\widetilde{O}()$ is used for characterizing the run times of Las Vegas algorithms. We say a Las Vegas algorithm has a resource (time, space, and so on.) bound of $\widetilde{O}(g(n))$ if there exists a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geq 1-\frac{1}{n^{\alpha}}$. We shall refer to these bounds as *high probability* bounds.

Similar definitions apply also to such functions as $\widetilde{\Theta}()$, $\widetilde{\Omega}()$, $\widetilde{o}()$, etc. □

Definition 1.18 [High probability] By *high probability* we mean a probability of $\geq 1-n^{-\alpha}$ for any fixed α. We call α the probability parameter. □

As mentioned above, the run time T of any Las Vegas algorithm is typically characterized as a random variable over a sample space S. The sample points of S are all possible outcomes for the randomizer used in the algorithm. Though it is desirable to obtain the distribution of T, often this is a challenging and unnecessary task. The expected value of T often suffices as a good indicator of the run time. We can do better than obtaining the mean of T but short of computing the exact distribution by obtaining the high probability bounds. The high probability bounds of our interest are of the form "With high probability the value of T will not exceed T_0," for some appropriate T_0.

Several results from probability theory can be employed to obtain high probability bounds on any random variable. Two of the more useful such results are Markov's inequality and Chernoff bounds.

Next we give two examples of randomized algorithms. The first is of the Las Vegas type and the second is of the Monte Carlo type. Other examples are presented throughout the text. We say a Monte Carlo (Las Vegas) algorithm has *failed* if it does not give a correct answer (terminate within a specified amount of time).

1.4.3 Identifying the Repeated Element

Consider an array $a[\ ]$ of n numbers that has $\frac{n}{2}$ distinct elements and $\frac{n}{2}$ copies of another element. The problem is to identify the repeated element.

Any deterministic algorithm for solving this problem will need at least $\frac{n}{2}+2$ time steps in the worst case. This fact can be argued as follows: Consider an adversary who has perfect knowledge about the algorithm used and who is in charge of selecting the input for the algorithm. Such an adversary can make sure that the first $\frac{n}{2}+1$ elements examined by the algorithm are all distinct. Even after having looked at $\frac{n}{2}+1$ elements, the algorithm will not be in a position to infer the repeated element. It will have to examine at least $\frac{n}{2}+2$ elements and hence take at least $\frac{n}{2}+2$ time steps.

In contrast there is a simple and elegant randomized Las Vegas algorithm that takes only $\tilde{O}(\log n)$ time. It randomly picks two array elements and checks whether they come from two different cells and have the same value. If they do, the repeated element has been found. If not, this basic step of sampling is repeated as many times as it takes to identify the repeated element.

In this algorithm, the sampling performed is with repetitions; that is, the first and second elements are randomly picked from out of the n elements (each element being equally likely to be picked). Thus there is a probability (equal to $\frac{1}{n}$) that the same array element is picked each time. If we just check for the equality of the two elements picked, our answer might be incorrect (in case the algorithm picked the same array index each time). Therefore, it is essential to make sure that the two array indices picked are different and the two array cells contain the same value.

This algorithm is given in Algorithm 1.27. The algorithm returns the array index of one of the copies of the repeated element. Now we prove that the run time of the above algorithm is $\tilde{O}(\log n)$. Any iteration of the **while** loop will be successful in identifying the repeated number if i is any one the $\frac{n}{2}$ array indices corresponding to the repeated element and j is any one of the same $\frac{n}{2}$ indices other than i. In other words, the probability that the algorithm quits in any given iteration of the **while** loop is $P = \frac{n/2(n/2-1)}{n^2}$, which is $\geq \frac{1}{5}$ for all $n \geq 10$. This implies that the probability that the algorithm does not quit in a given iteration is $< \frac{4}{5}$.

```
1   RepeatedElement(a, n)
2   // Finds the repeated element from a[1 : n].
3   {
4       while (true) do
5       {
6           i := Random() mod n + 1; j := Random() mod n + 1;
7           // i and j are random numbers in the range [1, n].
8           if ((i ≠ j) and (a[i] = a[j])) then return i;
9       }
10  }
```

Algorithm 1.27 Identifying the repeated array number

Therefore, the probability that the algorithm does not quit in 10 iterations is $< \left(\frac{4}{5}\right)^{10} < .1074$. So, Algorithm 1.27 will terminate in 10 iterations or less with probability $\geq .8926$. The probability that the algorithm does not terminate in 100 iterations is $< \left(\frac{4}{5}\right)^{100} < 2.04 * 10^{-10}$. That is, almost certainly the algorithm will quit in 100 iterations or less. If n equals $2 * 10^6$, for example, any deterministic algorithm will have to spend at least one million time steps, as opposed to the 100 iterations of Algorithm 1.27!

In general, the probability that the algorithm does not quit in the first $c\alpha \log n$ (c is a constant to be fixed) iterations is

$$< (4/5)^{c\alpha \log n} = n^{-c\alpha \log (5/4)}$$

which will be $< n^{-\alpha}$ if we pick $c \geq \frac{1}{\log (5/4)}$.

Thus the algorithm terminates in $\frac{1}{\log (5/4)} \alpha \log n$ iterations or less with probability $\geq 1 - n^{-\alpha}$. Since each iteration of the **while** loop takes $O(1)$ time, the run time of the algorithm is $\tilde{O}(\log n)$.

Note that this algorithm, if it terminates, will always output the correct answer and hence is of the Las Vegas type. The above analysis shows that the algorithm will terminate quickly with high probability.

The same problem of inferring the repeated element can be solved using many deterministic algorithms. For example, sorting the array is one way. But sorting takes $\Omega(n \log n)$ time (proved in Chapter 10). An alternative is to partition the array into $\lceil \frac{n}{3} \rceil$ parts, where each part (possibly except for one part) has three array elements, and to search the individual parts for

the repeated element. At least one of the parts will have two copies of the repeated element. (Prove this!) The run time of this algorithm is $\Theta(n)$.

1.4.4 Primality Testing

Any integer greater than one is said to be a *prime* if its only divisors are 1 and the integer itself. By convention, we take 1 to be a nonprime. Then $2, 3, 5, 7, 11$, and 13 are the first six primes. Given an integer n, the problem of deciding whether n is a prime is known as *primality testing*. It has a number of applications including cryptology.

If a number n is composite (i.e., nonprime), it must have a divisor $\leq \lfloor \sqrt{n} \rfloor$. This observation leads to the following simple algorithm for primality testing: Consider each number ℓ in the interval $[2, \lfloor \sqrt{n} \rfloor]$ and check whether ℓ divides n. If none of these numbers divides n, then n is prime; otherwise it is composite.

Assuming that it takes $\Theta(1)$ time to determine whether one integer divides another, the naive primality testing algorithm has a run time of $O(\sqrt{n})$. The input size for this problem is $\lceil (\log n + 1) \rceil$, since n can be represented in binary form with these many bits. Thus the run time of this simple algorithm is exponential in the input size (notice that $\sqrt{n} = 2^{\frac{1}{2} \log n}$).

We can devise a Monte Carlo randomized algorithm for primality testing that runs in time $O((\log n)^2)$. The output of this algorithm is correct with high probability. If the input is prime, the algorithm never gives an incorrect answer. However, if the input number is composite (i.e., nonprime), then there is a small probability that the answer may be incorrect. Algorithms of this kind are said to have *one-sided error*.

Before presenting further details, we list two theorems from number theory that will serve as the backbone of the algorithm. The proofs of these theorems can be found in the references supplied at the end of this chapter.

Theorem 1.6 [Fermat] If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for any integer $a < n$. □

Theorem 1.7 The equation $x^2 \equiv 1 \pmod{n}$ has exactly two solutions, namely 1 and $n - 1$, if n is prime. □

Corollary 1.1 If the equation $x^2 \equiv 1 \pmod{n}$ has roots other than 1 and $n - 1$, then n is composite. □

Note: Any integer x which is neither 1 nor $n - 1$ but which satisfies $x^2 \equiv 1 \pmod{n}$ is said to be a *nontrivial square root* of 1 modulo n.

Fermat's theorem suggests the following algorithm for primality testing: Randomly choose an $a < n$ and check whether $a^{n-1} \equiv 1 \pmod{n}$ (call this

Fermat's equation). If Fermat's equation is not satisfied, n is composite. If the equation is satisfied, we try some more random a's. If on each a tried, Fermat's equation is satisfied, we output "n is prime"; otherwise we output "n is composite." In order to compute a^{n-1} **mod** n, we could employ Exponentiate (Algorithm 1.16) with some minor modifications. The resultant primality testing algorithm is given as Algorithm 1.28. Here *large* is a number sufficiently *large* that ensures a probability of correctness of $\geq 1 - n^{-\alpha}$.

```
1   Prime0(n, α)
2   // Returns true if n is a prime and false otherwise.
3   // α is the probability parameter.
4   {
5       q := n − 1;
6       for i := 1 to large do  // Specify large.
7       {
8           m := q; y := 1;
9           a := Random() mod q + 1;
10          // Choose a random number in the range [1, n − 1].
11          z := a;
12          // Compute a^{n−1} mod n.
13          while (m > 0) do
14          {
15              while (m mod 2 = 0) do
16              {
17                  z := z^2 mod n; m := ⌊m/2⌋;
18              }
19              m := m − 1; y := (y ∗ z) mod n;
20          }
21          if (y ≠ 1) then return false;
22          // If a^{n−1} mod n is not 1, n is not a prime.
23      }
24      return true;
25  }
```

Algorithm 1.28 Primality testing: first attempt

If the input is prime, Algorithm 1.28 will never output an incorrect answer. If n is composite, will Fermat's equation never be satisfied for any a less than n and greater than one? If so, the above algorithm has to examine just one a before coming up with the correct answer. Unfortunately, the

answer to this question is no. Even if n is composite, Fermat's equation may be satisfied depending on the a chosen.

Is it the case that for every n (that is composite) there will be some nonzero constant fraction of a's less than n that will not satisfy Fermat's equation? If the answer is yes and if the above algorithm tries a sufficiently large number of a's, there is a high probability that at least one a violating Fermat's equation will be found and hence the correct answer be output. Here again, the answer is no. There are composite numbers (known as Carmichael numbers) for which every a that is less than and relatively prime to n will satisfy Fermat's equation. (The number of a's that do not satisfy Fermat's equation need not be a constant fraction.) The numbers 561 and 1105 are examples of Carmichael numbers.

Fortunately, a slight modification of the above algorithm takes care of these problems. The modified primality testing algorithm (also known as Miller-Rabin's algorithm) is the same as Prime0 (Algorithm 1.28) except that within the body of Prime0, we also look for nontrivial square roots of n. The modified version is given in Algorithm 1.29. We assume that n is odd.

Miller-Rabin's algorithm will never give an incorrect answer if the input is prime, since Fermat's equation will always be satisfied and no nontrivial square root of 1 modulo n can be found. If n is composite, the above algorithm will detect the compositeness of n if the randomly chosen a either leads to the discovery of a nontrivial square root of 1 or violates Fermat's equation. Call any such a a *witness* to the compositeness of n. What is the probability that a randomly chosen a will be a witness to the compositeness of n? This question is answered by the following theorem (the proof can be found in the references at the end of this chapter).

Theorem 1.8 There are at least $\frac{n-1}{2}$ witnesses to the compositeness of n if n is composite and odd. □

Assume that n is composite (since if n is prime, the algorithm will always be correct). The probability that a randomly chosen a will be a witness is $\geq \frac{n-1}{2n}$, which is very nearly equal to $\frac{1}{2}$. This means that a randomly chosen a will fail to be a witness with probability $\leq \frac{1}{2}$.

Therefore, the probability that none of the first $\alpha \log n$ a's chosen is a witness is $\leq \left(\frac{1}{2}\right)^{\alpha \log n} = n^{-\alpha}$. In other words, the algorithm Prime will give an incorrect answer with only probability $\leq n^{-\alpha}$.

The run time of the outermost **while** loop is nearly the same as that of Exponentiate (Algorithm 1.16) and equal to $O(\log n)$. Since this **while** loop is executed $O(\log n)$ times, the run time of the whole algorithm is $O(\log^2 n)$.

5. Given a 2-sided coin. Using this coin, how will you simulate an n-sided coin
   (a) when n is a power of 2?.
   (b) when n is not a power of 2?.
6. Compute the run time analysis of the Las Vegas algorithm given in Algorithm 1.30 and express it using the $\tilde{O}()$ notation.

```
1  LasVegas()
2  {
3      while (true) do
4      {
5          i := Random() mod 2;
6          if (i ≥ 1) then return;
7      }
8  }
```

Algorithm 1.30 A Las Vegas algorithm

7. There are $\sqrt{n}$ copies of an element in the array c. Every other element of c occurs exactly once. If the algorithm RepeatedElement is used to identify the repeated element of c, will the run time still be $\tilde{O}(\log n)$? If so, why? If not, what is the new run time?
8. What is the minimum number of times that an element should be repeated in an array (the other elements of the array occurring exactly once) so that it can be found using RepeatedElement in $\tilde{O}(\log n)$ time?
9. An array a has $\frac{n}{4}$ copies of a particular unknown element x. Every other element in a has at most $\frac{n}{8}$ copies. Present an $O(\log n)$ time Monte Carlo algorithm to identify x. The answer should be correct with high probability. Can you develop an $\tilde{O}(\log n)$ time Las Vegas algorithm for the same problem?
10. Consider the naive Monte Carlo algorithm for primality testing presented in Algorithm 1.31. Here Power(x, y) computes x^y. What should be the value of t for the algorithm's output to be correct with high probability?
11. Let $\mathcal{A}$ be a Monte Carlo algorithm that solves a decision problem π in time T. The output of $\mathcal{A}$ is correct with probability $\geq \frac{1}{2}$. Show how

```
1   Prime1(n)
2   {
3       // Specify t.
4       for i := 1 to t do
5       {
6           m := Power(n, 0.5);
7           j := Random() mod m + 2;
8           if ((n mod j) = 0) then return false;
9           // If j divides n, n is not prime.
10      }
11      return true;
12  }
```

Algorithm 1.31 Another primality testing algorithm

you can modify $\mathcal{A}$ so that its answer is correct with high probability. The modified version can take $O(T \log n)$ time.

12. In general a Las Vegas algorithm is preferable to a Monte Carlo algorithm, since the answer given by the former is guaranteed to be correct. There may be critical situations in which even a very small probability of an incorrect answer is unacceptable. Say there is a Monte Carlo algorithm for solving a problem π in T_1 time units whose output is correct with probability $\geq \frac{1}{2}$. Also assume that there is another algorithm that can check whether a given answer is valid for π in T_2 time units. Show how you use these two algorithms to arrive at a Las Vegas algorithm for solving π in time $\widetilde{O}((T_1 + T_2) \log n)$.

13. The problem considered here is that of searching for an element x in an array $a[1 : n]$. Algorithm 1.17 gives a deterministic $\Theta(n)$ time algorithm for this problem. Show that any deterministic algorithm will have to take $\Omega(n)$ time in the worst case for this problem. In contrast a randomized Las Vegas algorithm that searches for x is given in Algorithm 1.32. This algorithm assumes that x is in $a[\,]$. What is the $\widetilde{O}()$ run time of this algorithm?

```
1   Algorithm RSearch(a, x, n)
2   // Searches for x in a[1 : n]. Assume that x is in a[ ].
3   {
4       while (true) do
5       {
6           i := Random() mod n + 1;
7           // i is random in the range [1, n].
8           if (a[i] = x) then return i;
9       }
10  }
```

Algorithm 1.32 Randomized search

1.5 REFERENCES AND READINGS

For a more detailed discussion of performance analysis and measurement, see *Software Development in Pascal*, Third Edition, by S. Sahni, NSPAN Printing and Publishing, 1993.

For a discussion on mathematical tools for analysis see *Concrete Mathematics: A Foundation for Computer Science*, by R. L. Graham, D. E. Knuth, and O. Patashnik, Addison-Wesley, 1989.

More details about the primality testing algorithm can be found in *Introduction to Algorithms*, by T. H. Cormen, C. E. Leiserson, and R. L. Rivest, MIT Press, 1990.

An excellent introductory text on probability theory is *Probability and Random Processes*, by G. R. Grimmet and D. R. Stirzaker, Oxford University Press, 1988. A proof of Lemma 1.1 can be found in this book. For a proof of Lemma 1.2 see *Queueing Systems*, Vol. I, by L. Kleinrock, John Wiley & Sons, 1975.

A formal treatment of randomized algorithms and several examples can be found in "Derivation of randomized algorithms for sorting and selection," by S. Rajasekaran and J. H. Reif, in *Parallel Algorithm Derivation and Program Transformation*, edited by R. Paige, J. H. Reif, and R. Wachter, Kluwer Academic Publishers, 1993, pp. 187–205. For more on randomized algorithms see *Randomized Algorithms* by R. Motwani and P. Raghavan, Cambridge University Press, 1995.

Chapter 3

DIVIDE-AND-CONQUER

3.1 GENERAL METHOD

Given a function to compute on n inputs the *divide-and-conquer* strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$, yielding k subproblems. These subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the subproblems resulting from a divide-and-conquer design are of the *same* type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

To be more precise, suppose we consider the divide-and-conquer strategy when it splits the input into two subproblems of the same kind as the original problem. This splitting is typical of many of the problems we examine here. We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look. By a *control abstraction* we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. DAndC (Algorithm 3.1) is initially invoked as DAndC(P), where P is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These subproblems $P_1, P_2, \ldots, P_k$ are solved by recursive applications of DAndC. Combine is a function that determines the solution to P using the solutions to the k subproblems. If the size of P is n and the sizes of the k subproblems are $n_1, n_2, \ldots, n_k$, respectively, then the

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
9       }
10  }
```

Algorithm 3.1 Control abstraction for divide-and-conquer

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases} \tag{3.1}$$

where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems. For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \tag{3.2}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

One of the methods for solving any such recurrence relation is called the *substitution method.* This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Beginning with the recurrence (3.2) and using the substitution method, it can be shown that

$$T(n) = n^{\log_b a}[T(1) + u(n)]$$

where $u(n) = \sum_{j=1}^{k} h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$. Table 3.1 tabulates the asymptotic value of $u(n)$ for various values of $h(n)$. This table allows one to easily obtain the asymptotic value of $T(n)$ for many of the recurrences one encounters when analyzing divide-and-conquer algorithms. Let us consider some examples using this table.

$h(n)$	$u(n)$
$O(n^r)$, $r < 0$	$O(1)$
$\Theta((\log n)^i)$, $i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r)$, $r > 0$	$\Theta(h(n))$

Table 3.1 $u(n)$ values for various $h(n)$ values

Example 3.2 Look at the following recurrence when n is a power of 2:

$$T(n) = \begin{cases} T(1) & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Comparing with (3.2), we see that $a = 1$, $b = 2$, and $f(n) = c$. So, $\log_b(a) = 0$ and $h(n) = f(n)/n^{\log_b a} = c = c(\log n)^0 = \Theta((\log n)^0)$. From Table 3.1, we obtain $u(n) = \Theta(\log n)$. So, $T(n) = n^{\log_b a}[c + \Theta(\log n)] = \Theta(\log n)$. □

Example 3.3 Next consider the case in which $a = 2$, $b = 2$, and $f(n) = cn$. For this recurrence, $\log_b a = 1$ and $h(n) = f(n)/n = c = \Theta((\log n)^0)$. Hence, $u(n) = \Theta(\log n)$ and $T(n) = n[T(1) + \Theta(\log n)] = \Theta(n \log n)$. □

Example 3.4 As another example, consider the recurrence $T(n) = 7T(n/2) + 18n^2$, $n \geq 2$ and a power of 2. We obtain $a = 7$, $b = 2$, and $f(n) = 18n^2$. So, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$, where $r = 2 - \log_2 7 < 0$. So, $u(n) = O(1)$. The expression for $T(n)$ is

$$\begin{aligned} T(n) &= n^{\log_2 7}[T(1) + O(1)] \\ &= \Theta(n^{\log_2 7}) \end{aligned}$$

as $T(1)$ is assumed to be a constant. □

Example 3.5 As a final example, consider the recurrence $T(n) = 9T(n/3) + 4n^6$, $n \geq 3$ and a power of 3. Comparing with (3.2), we obtain $a = 9$, $b = 3$, and $f(n) = 4n^6$. So, $\log_b a = 2$ and $h(n) = 4n^6/n^2 = 4n^4 = \Omega(n^4)$. From Table 3.1, we see that $u(n) = \Theta(h(n)) = \Theta(n^4)$. So,

$$\begin{aligned} T(n) &= n^2[T(1) + \Theta(n^4)] \\ &= \Theta(n^6) \end{aligned}$$

as $T(1)$ can be assumed constant. □

EXERCISES

1. Solve the recurrence relation (3.2) for the following choices of a, b, and $f(n)$ (c being a constant):

   (a) $a = 1$, $b = 2$, and $f(n) = cn$

   (b) $a = 5$, $b = 4$, and $f(n) = cn^2$

   (c) $a = 28$, $b = 3$, and $f(n) = cn^3$

2. Solve the following recurrence relations using the substitution method:

   (a) All three recurrences of Exercise 1.

   (b)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ T(\sqrt{n}) + c & n > 4 \end{cases}$$

(c)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

(d)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \frac{\log n}{\log \log n} & n > 4 \end{cases}$$

3.2 BINARY SEARCH

Let a_i, $1 \leq i \leq n$, be a list of elements that are sorted in nondecreasing order. Consider the problem of determining whether a given element x is present in the list. If x is present, we are to determine a value j such that $a_j = x$. If x is not in the list, then j is to be set to zero. Let $P = (n, a_i, \ldots, a_\ell, x)$ denote an arbitrary instance of this search problem (n is the number of elements in the list, $a_i, \ldots, a_\ell$ is the list of elements, and x is the element searched for).

Divide-and-conquer can be used to solve this problem. Let Small(P) be true if $n = 1$. In this case, S(P) will take the value i if $x = a_i$; otherwise it will take the value 0. Then $g(1) = \Theta(1)$. If P has more than one element, it can be divided (or reduced) into a new subproblem as follows. Pick an index q (in the range $[i, \ell]$) and compare x with a_q. There are three possibilities: (1) $x = a_q$: In this case the problem P is immediately solved. (2) $x < a_q$: In this case x has to be searched for only in the sublist $a_i, a_{i+1}, \ldots, a_{q-1}$. Therefore, P reduces to $(q - i, a_i, \ldots, a_{q-1}, x)$. (3) $x > a_q$: In this case the sublist to be searched is $a_{q+1}, \ldots, a_\ell$. P reduces to $(\ell - q, a_{q+1}, \ldots, a_\ell, x)$.

In this example, any given problem P gets divided (reduced) into one new subproblem. This division takes only $\Theta(1)$ time. After a comparison with a_q, the instance remaining to be solved (if any) can be solved by using this divide-and-conquer scheme again. If q is always chosen such that a_q is the middle element (that is, $q = \lfloor (n+1)/2 \rfloor$), then the resulting search algorithm is known as binary search. Note that the answer to the new subproblem is also the answer to the original problem P; there is no need for any combining. Algorithm 3.2 describes this binary search method, where BinSrch has four inputs $a[\,], i, l$, and x. It is initially invoked as BinSrch$(a, 1, n, x)$.

A nonrecursive version of BinSrch is given in Algorithm 3.3. BinSearch has three inputs a, n, and x. The **while** loop continues processing as long as there are more elements left to check. At the conclusion of the procedure 0 is returned if x is not present, or j is returned, such that $a[j] = x$.

Is BinSearch an algorithm? We must be sure that all of the operations such as comparisons between x and $a[mid]$ are well defined. The relational operators carry out the comparisons among elements of a correctly if these operators are appropriately defined. Does BinSearch terminate? We observe

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Algorithm 3.2 Recursive binary search

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12              else return mid;
13      }
14      return 0;
15  }
```

Algorithm 3.3 Iterative binary search

that *low* and *high* are integer variables such that each time through the loop either x is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* becomes greater than *high* and causes termination in a finite number of steps if x is not present.

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

place them in $a[1:14]$, and simulate the steps that BinSearch goes through as it searches for different values of x. Only the variables *low*, *high*, and *mid* need to be traced as we simulate the algorithm. We try the following values for x: 151, −14, and 9 for two successful searches and one unsuccessful search. Table 3.2 shows the traces of BinSearch on these three inputs. □

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Table 3.2 Three examples of binary search on 14 elements

These examples may give us a little more confidence about Algorithm 3.3, but they by no means prove that it is correct. Proofs of algorithms are very useful because they establish the correctness of the algorithm for *all* possible inputs, whereas testing gives much less in the way of guarantees. Unfortunately, algorithm proving is a very difficult process and the complete proof of an algorithm can be many times longer than the algorithm itself. We content ourselves with an informal "proof" of BinSearch.

Theorem 3.1 Algorithm BinSearch(a, n, x) works correctly.

Proof: We assume that all statements work as expected and that comparisons such as $x > a[mid]$ are appropriately carried out. Initially $low = 1$, $high = n$, $n \geq 0$, and $a[1] \leq a[2] \leq \cdots \leq a[n]$. If $n = 0$, the **while** loop is

not entered and 0 is returned. Otherwise we observe that each time through the loop the possible elements to be checked for equality with x are $a[low]$, $a[low+1]$, ..., $a[mid]$, ..., $a[high]$. If $x = a[mid]$, then the algorithm terminates successfully. Otherwise the range is narrowed by either increasing low to $mid+1$ or decreasing $high$ to $mid-1$. Clearly this narrowing of the range does not affect the outcome of the search. If low becomes greater than $high$, then x is not present and hence the loop is exited. □

Notice that to fully test binary search, we need not concern ourselves with the values of $a[1:n]$. By varying x sufficiently, we can observe all possible computation sequences of BinSearch without devising different values for a. To test all successful searches, x must take on the n values in a. To test all unsuccessful searches, x need only take on $n+1$ different values. Thus the complexity of testing BinSearch is $2n+1$ for each n.

Now let's analyze the execution profile of BinSearch. The two relevant characteristics of this profile are the frequency counts and space required for the algorithm. For BinSearch, storage is required for the n elements of the array plus the variables $low, high, mid$, and x, or $n+4$ locations. As for the time, there are three possibilities to consider: the best, average, and worst cases.

Suppose we begin by determining the time for BinSearch on the previous data set. We observe that the only operations in the algorithm are comparisons and some arithmetic and data movements. We concentrate on comparisons between x and the elements in $a[\ ]$, recognizing that the frequency count of all other operations is of the same order as that for these comparisons. Comparisons between x and elements of $a[\ ]$ are referred to as *element comparisons*. We assume that only one comparison is needed to determine which of the three possibilities of the **if** statement holds. The number of element comparisons needed to find each of the 14 elements is

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

No element requires more than 4 comparisons to be found. The average is obtained by summing the comparisons needed to find all 14 items and dividing by 14; this yields 45/14, or approximately 3.21, comparisons per successful search on the average. There are 15 possible ways that an unsuccessful search may terminate depending on the value of x. If $x < a[1]$, the algorithm requires 3 element comparisons to determine that x is not present. For all the remaining possibilities, BinSearch requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is $(3 + 14 * 4)/15 = 59/15 \approx 3.93$.

The analysis just done applies to any sorted sequence containing 14 elements. But the result we would prefer is a formula for n elements. A good

way to derive such a formula plus a better way to understand the algorithm is to consider the sequence of values for *mid* that are produced by BinSearch for all possible values of x. These values are nicely described using a binary decision tree in which the value in each node is the value of *mid*. For example, if $n = 14$, then Figure 3.1 contains a binary decision tree that traces the way in which these values are produced by BinSearch.

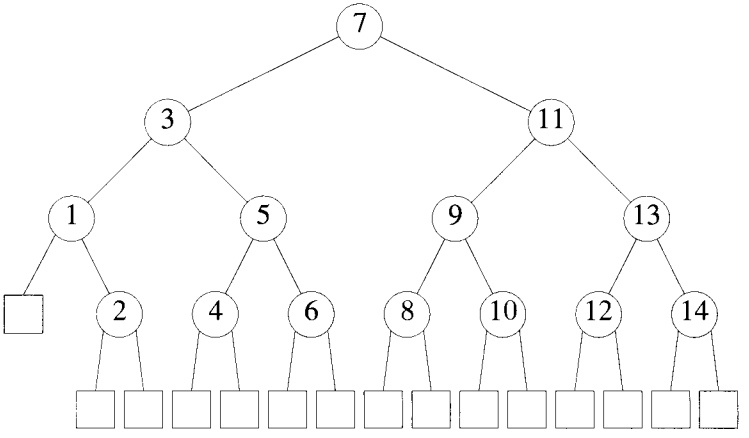


Figure 3.1 Binary decision tree for binary search, $n = 14$

The first comparison is x with $a[7]$. If $x < a[7]$, then the next comparison is with $a[3]$; similarly, if $x > a[7]$, then the next comparison is with $a[11]$. Each path through the tree represents a sequence of comparisons in the binary search method. If x is present, then the algorithm will end at one of the circular nodes that lists the index into the array where x was found. If x is not present, the algorithm will terminate at one of the square nodes. Circular nodes are called *internal* nodes, and square nodes are referred to as *external nodes*.

Theorem 3.2 If n is in the range $[2^{k-1}, 2^k)$, then BinSearch makes at most k element comparisons for a successful search and either $k-1$ or k comparisons for an unsuccessful search. (In other words the time for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$).

Proof: Consider the binary decision tree describing the action of BinSearch on n elements. All successful searches end at a circular node whereas all unsuccessful searches end at a square node. If $2^{k-1} \leq n < 2^k$, then all circular nodes are at levels $1, 2, \ldots, k$ whereas all square nodes are at levels

k and $k+1$ (note that the root is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is only $i-1$. The theorem follows. □

Theorem 3.2 states the worst-case time for binary search. To determine the average behavior, we need to look more closely at the binary decision tree and equate its size to the number of element comparisons in the algorithm. The *distance* of a node from the root is one less than its level. The *internal path length* I is the sum of the distances of all internal nodes from the root. Analogously, the *external path length* E is the sum of the distances of all external nodes from the root. It is easy to show by induction that for any binary tree with n internal nodes, E and I are related by the formula

$$E = I + 2n$$

It turns out that there is a simple relationship between E, I, and the average number of comparisons in binary search. Let $A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ the average number of comparisons in an unsuccessful search. The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root. Hence,

$$A_s(n) = 1 + I/n$$

The number of comparisons on any path from the root to an external node is equal to the distance between the root and the external node. Since every binary tree with n internal nodes has $n+1$ external nodes, it follows that

$$A_u(n) = E/(n+1)$$

Using these three formulas for $E, A_s(n)$, and $A_u(n)$, we find that

$$A_s(n) = (1 + 1/n)A_u(n) - 1$$

From this formula we see that $A_s(n)$ and $A_u(n)$ are directly related. The minimum value of $A_s(n)$ (and hence $A_u(n)$) is achieved by an algorithm whose binary decision tree has minimum external and internal path length. This minimum is achieved by the binary tree all of whose external nodes are on adjacent levels, and this is precisely the tree that is produced by binary search. From Theorem 3.2 it follows that E is proportional to $n \log n$. Using this in the preceding formulas, we conclude that $A_s(n)$ and $A_u(n)$ are both proportional to $\log n$. Thus we conclude that the average- and worst-case numbers of comparisons for binary search are the same to within a constant

factor. The best-case analysis is easy. For a successful search only one element comparison is needed. For an unsuccessful search, Theorem 3.2 states that $\lfloor \log n \rfloor$ element comparisons are needed in the best case.

In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe the best, average, and worst cases:

successful searches			**unsuccessful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

Can we expect another searching algorithm to be significantly better than binary search in the worst case? This question is pursued rigorously in Chapter 10. But we can anticipate the answer here, which is no. The method for proving such an assertion is to view the binary decision tree as a general model for any searching algorithm that depends on comparisons of entire elements. Viewed in this way, we observe that the *longest* path to discover any element is minimized by binary search, and so any alternative algorithm is no better from this point of view.

Before we end this section, there is an interesting variation of binary search that makes only one comparison per iteration of the **while** loop. This variation appears as Algorithm 3.4. The correctness proof of this variation is left as an exercise.

BinSearch will sometimes make twice as many element comparisons as BinSearch1 (for example, when $x > a[n]$). However, for successful searches BinSearch1 may make $(\log n)/2$ more element comparisons than BinSearch (for example, when $x = a[mid]$). The analysis of BinSearch1 is left as an exercise. It should be easy to see that the best-, average-, and worst-case times for BinSearch1 are $\Theta(\log n)$ for both successful and unsuccessful searches.

These two algorithms were run on a Sparc 10/30. The first two rows in Table 3.3 represent the average time for a successful search. The second set of two rows give the average times for all possible unsuccessful searches. For both successful and unsuccessful searches BinSearch1 did marginally better than BinSearch.

EXERCISES

1. Run the recursive and iterative versions of binary search and compare the times. For appropriate sizes of n, have each algorithm find every element in the set. Then try all $n + 1$ possible unsuccessful searches.

2. Prove by induction the relationship $E = I + 2n$ for a binary tree with n internal nodes. The variables E and I are the external and internal path length, respectively.

```
1   Algorithm BinSearch1(a, n, x)
2   // Same specifications as BinSearch except n > 0
3   {
4       low := 1; high := n + 1;
5       // high is one more than possible.
6       while (low < (high − 1)) do
7       {
8           mid := ⌊(low + high)/2⌋;
9           if (x < a[mid]) then high := mid;
10              // Only one comparison in the loop.
11          else low := mid; // x ≥ a[mid]
12      }
13      if (x = a[low]) then return low; // x is present.
14      else return 0; // x is not present.
15  }
```

Algorithm 3.4 Binary search using one comparison per cycle

Array sizes	5,000	10,000	15,000	20,000	25,000	30,000
successful searches						
BinSearch	51.30	67.95	67.72	73.85	76.77	73.40
BinSearch1	47.68	53.92	61.98	67.46	68.95	71.11
unsuccessful searches						
BinSearch	50.40	66.36	76.78	79.54	78.20	81.15
BinSearch1	41.93	52.65	63.33	66.86	69.22	72.26

Table 3.3 Computing times for two binary search algorithms; times are in microseconds

3. In an infinite array, the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. Present an algorithm that takes x as input and finds the position of x in the array in $\Theta(\log n)$ time. *You are not given the value of* n.

4. Devise a "binary" search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. How does this algorithm compare with binary search?

5. Devise a ternary search algorithm that first tests the element at position $n/3$ for equality with some value x, and then checks the element at $2n/3$ and either discovers x or reduces the set size to one-third the size of the original. Compare this with binary search.

6. (a) Prove that BinSearch1 works correctly.

   (b) Verify that the following algorithm segment functions correctly according to the specifications of binary search. Discuss its computing time.

```
low := 1; high := n;
repeat {
    mid := ⌊(low + high)/2⌋;
    if (x ≥ a[mid]) then low := mid;
    else high := mid;
} until ((low + 1) = high)
```

3.3 FINDING THE MAXIMUM AND MINIMUM

Let us consider another simple problem that can be solved by the divide-and-conquer technique. The problem is to find the maximum and minimum items in a set of n elements. Algorithm 3.5 is a straightforward algorithm to accomplish this.

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. The justification for this is that the frequency count for other operations in this algorithm is of the same order as that for element comparisons. More importantly, when the elements in $a[1:n]$ are polynomials, vectors, very large numbers, or strings of characters, the cost of an element comparison is much higher than the cost of the other operations. Hence the time is determined mainly by the total cost of the element comparisons.

StraightMaxMin requires $2(n-1)$ element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

Algorithm 3.5 Straightforward maximum and minimum

that the comparison $a[i] < min$ is necessary only when $a[i] > max$ is false. Hence we can replace the contents of the **for** loop by

```
if (a[i] > max) then max := a[i];
else if (a[i] < min) then min := a[i];
```

Now the best case occurs when the elements are in increasing order. The number of element comparisons is $n - 1$. The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is $2(n - 1)$. The average number of element comparisons is less than $2(n - 1)$. On the average, $a[i]$ is greater than max half the time, and so the average number of comparisons is $3n/2 - 1$.

A divide-and-conquer algorithm for this problem would proceed as follows: Let $P = (n, a[i], \ldots, a[j])$ denote an arbitrary instance of the problem. Here n is the number of elements in the list $a[i], \ldots, a[j]$ and we are interested in finding the maximum and minimum of this list. Let Small(P)/ be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n = 1$. If $n = 2$, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances $P_1 = (\lfloor n/2 \rfloor, a[1], \ldots, a[\lfloor n/2 \rfloor])$ and $P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$. After having divided P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm. How can we combine the solutions for P_1 and P_2 to obtain a solution for P? If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is the larger of MAX(P_1) and MAX(P_2). Also, MIN(P) is the smaller of MIN(P_1) and MIN(P_2).

Algorithm 3.6 results from applying the strategy just described. MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements $\{a(i), a(i+1), \ldots, a(j)\}$. The situation of set sizes one ($i = j$) and two ($i = j - 1$) are handled separately. For sets containing more than two elements, the midpoint is determined (just as in binary search) and two new subproblems are generated. When the maxima and minima of these subproblems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j − 1) then  // Another case of Small(P)
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

Algorithm 3.6 Recursively finding the maximum and minimum

The procedure is initially invoked by the statement

$$\mathsf{MaxMin}(1, n, x, y)$$

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i, j, max, and min. On the array $a[\,]$ above, the tree of Figure 3.2 is produced.

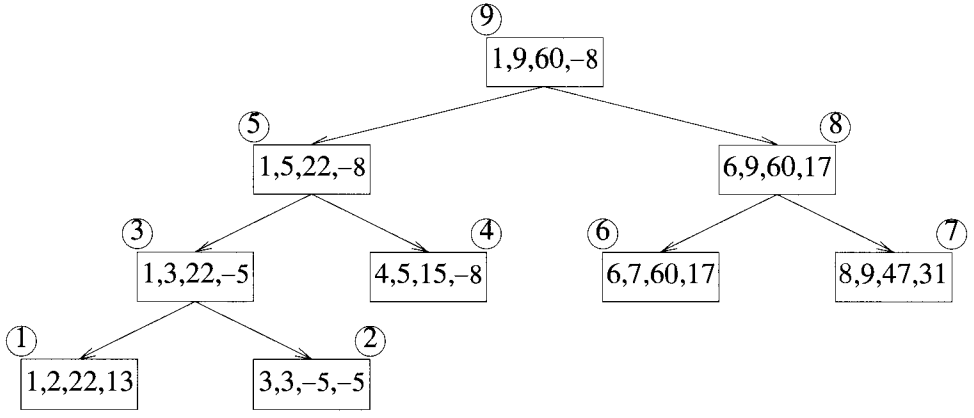


Figure 3.2 Trees of recursive calls of MaxMin

Examining Figure 3.2, we see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9, respectively, and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which max and min are assigned values.

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the $2n - 2$ comparisons for the straightforward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than $3n/2 - 2$ comparisons. So in this sense algorithm MaxMin is optimal (see Chapter 10 for more details). But does this imply that MaxMin is better in practice? Not necessarily. In terms of storage, MaxMin is worse than the straightforward algorithm because it requires stack space for i, j, $max, min, max1$, and $min1$. Given n elements, there will be $\lfloor \log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call (don't forget the return address is also needed).

Let us see what the count is when element comparisons have the same cost as comparisons between i and j. Let $C(n)$ be this number. First, we observe that lines 6 and 7 in Algorithm 3.6 can be replaced with

if $(i \ge j - 1)$ { // Small(P)

to achieve the same effect. Hence, a single comparison between i and $j - 1$ is adequate to implement the modified **if** statement. Assuming $n = 2^k$ for some positive integer k, we get

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$

Solving this equation, we obtain

$$\begin{aligned} C(n) &= 2C(n/2) + 3 \\ &= 4C(n/4) + 6 + 3 \\ &\vdots \\ &= 2^{k-1}C(2) + 3\ \textstyle\sum_0^{k-2} 2^i \\ &= 2^k + 3 * 2^{k-1} - 3 \\ &= 5n/2 - 3 \end{aligned} \tag{3.4}$$

The comparative figure for StraightMaxMin is $3(n-1)$ (including the comparison needed to implement the **for** loop). This is larger than $5n/2 - 3$. Despite this, MaxMin will be slower than StraightMaxMin because of the overhead of stacking i, j, max, and min for the recursion.

Algorithm 3.6 makes several points. If comparisons among the elements of $a[\ ]$ are much more costly than comparisons of integer variables, then the divide-and-conquer technique has yielded a more efficient (actually an optimal) algorithm. On the other hand, if this assumption is not true, the technique yields a less-efficient algorithm. Thus the divide-and-conquer strategy is seen to be only a guide to better algorithm design which may not always succeed. Also we see that it is sometimes necessary to work out the constants associated with the computing time bound for an algorithm. Both MaxMin and StraightMaxMin are $\Theta(n)$, so the use of asymptotic notation is not enough of a discriminator in this situation. Finally, see the exercises for another way to find the maximum and minimum using only $3n/2 - 2$ comparisons.

Note: In the design of any divide-and-conquer algorithm, typically, it is a straightforward task to define Small(P) and S(P). So, from now on, we only discuss how to divide any given problem P and how to combine the solutions to subproblems.

EXERCISES

1. Translate algorithm MaxMin into a computationally equivalent procedure that uses no recursion.

2. Test your iterative version of MaxMin derived above against StraightMaxMin. Count all comparisons.

3. There is an iterative algorithm for finding the maximum and minimum which, though not a divide-and-conquer-based algorithm, is probably more efficient than MaxMin. It works by comparing consecutive pairs of elements and then comparing the larger one with the current maximum and the smaller one with the current minimum. Write out

the algorithm completely and analyze the number of comparisons it requires.

4. In Algorithm 3.6, what happens if lines 7 to 17 are dropped? Does the resultant function still compute the maximum and minimum elements correctly?

3.4 MERGE SORT

As another example of divide-and-conquer, we investigate a sorting algorithm that has the nice property that in the worst case its complexity is $O(n \log n)$. This algorithm is called *merge sort*. We assume throughout that the elements are to be sorted in nondecreasing order. Given a sequence of n elements (also called keys) $a[1], \ldots, a[n]$, the general idea is to imagine them split into two sets $a[1], \ldots, a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

MergeSort (Algorithm 3.7) describes this process very succinctly using recursion and a function Merge (Algorithm 3.8) which merges two sorted sets. Before executing MergeSort, the n elements should be placed in $a[1:n]$. Then MergeSort(1,n) causes the keys to be rearranged into nondecreasing order in a.

Example 3.7 Consider the array of ten elements $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$. Algorithm MergeSort begins by splitting $a[\ ]$ into two subarrays each of size five ($a[1:5]$ and $a[6:10]$). The elements in $a[1:5]$ are then split into two subarrays of size three ($a[1:3]$) and two ($a[4:5]$). Then the items in $a[1:3]$ are split into subarrays of size two ($a[1:2]$) and one ($a[3:3]$). The two values in $a[1:2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

$$(310 \mid 285 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

$$(285, 310 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Algorithm 3.7 Merge sort

Then $a[3]$ is merged with $a[1:2]$ and

(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)

and then $a[1:3]$ and $a[4:5]$:

(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

(179, 285, 310, 351, 652 | 423 | 861 | 254 | 450, 520)

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6:7]$:

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

(179, 285, 310, 351, 652 | 254, 423, 861 | 450, 520)

Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

(179, 285, 310, 351, 652 | 254, 423, 450, 520, 861)

At this point there are two sorted subarrays and the final merge produces the fully sorted result

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

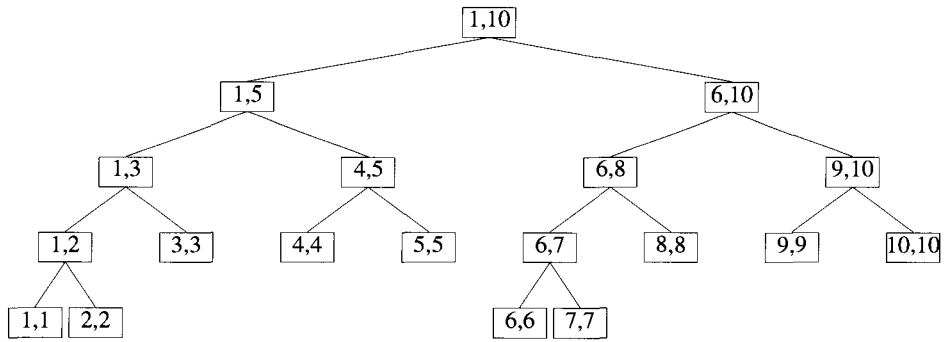


Figure 3.3 Tree of calls of MergeSort(1, 10)

Figure 3.3 is a tree that represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements. The pair of values in each node are the values of the parameters *low* and *high*. Notice how the splitting continues until sets containing a single element are produced. Figure 3.4 is a tree representing the calls to procedure Merge by MergeSort. For example, the node containing 1, 2, and 3 represents the merging of $a[1:2]$ with $a[3]$. □

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

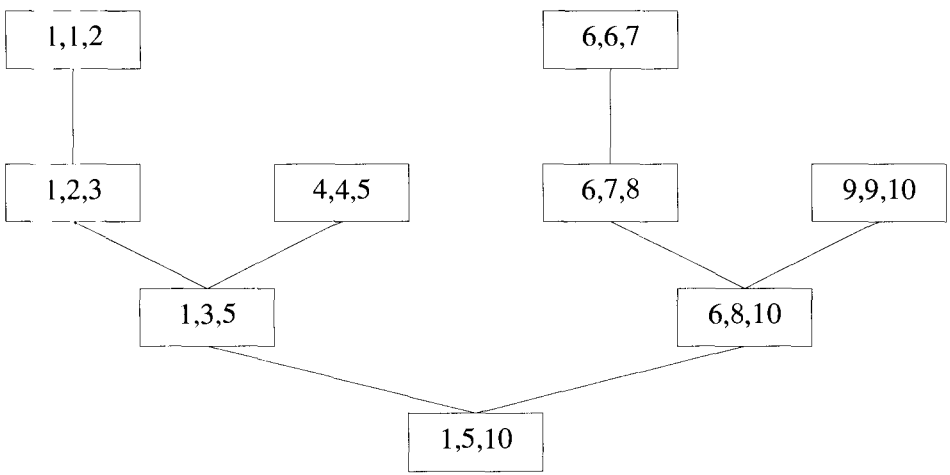


Figure 3.4 Tree of calls of Merge

Though Algorithm 3.7 nicely captures the divide-and-conquer nature of merge sort, there remain several inefficiencies that can and should be eliminated. We present these refinements in an attempt to produce a version of merge sort that is good enough to execute. Despite these improvements the algorithm's complexity remains $O(n \log n)$. We see in Chapter 10 that no sorting algorithm based on comparisons of entire keys can do better.

One complaint we might raise concerning merge sort is its use of $2n$ locations. The additional n locations were needed because we couldn't reasonably merge two sorted sets in place. But despite the use of this space the

algorithm must still work hard and copy the result placed into $b[low : high]$ back into $a[low : high]$ on each call of Merge. An alternative to this copying is to associate a new field of information with each key. (The elements in $a[\ ]$ are called *keys*.) This field is used to link the keys and any associated information together in a sorted list (keys and related information are called *records*). Then the merging of the sorted lists proceeds by changing the link values, and no records need be moved at all. A field that contains only a link will generally be smaller than an entire record, so less space will be used.

Along with the original array $a[\ ]$, we define an auxiliary array $link[1 : n]$ that contains integers in the range $[0, n]$. These integers are interpreted as pointers to elements of $a[\ ]$. A list is a sequence of pointers ending with a zero. Below is one set of values for *link* that contains two lists: Q and R. The integer $Q = 2$ denotes the start of one list and $R = 5$ the start of the other.

link:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	6	4	7	1	3	0	8	0

The two lists are $Q = (2, 4, 1, 6)$ and $R = (5, 3, 7, 8)$. Interpreting these lists as describing sorted subsets of $a[1 : 8]$, we conclude that $a[2] \leq a[4] \leq a[1] \leq a[6]$ and $a[5] \leq a[3] \leq a[7] \leq a[8]$.

Another complaint we could raise about MergeSort is the stack space that is necessitated by the use of recursion. Since merge sort splits each set into two approximately equal-sized subsets, the maximum depth of the stack is proportional to $\log n$. The need for stack space seems indicated by the top-down manner in which this algorithm was devised. The need for stack space can be eliminated if we build an algorithm that works bottom-up; see the exercises for details.

As can be seen from function MergeSort and the previous example, even sets of size two will cause two recursive calls to be made. For small set sizes most of the time will be spent processing the recursion instead of sorting. This situation can be improved by not allowing the recursion to go to the lowest level. In terms of the divide-and-conquer control abstraction, we are suggesting that when Small is true for merge sort, more work should be done than simply returning with no action. We use a second sorting algorithm that works well on small-sized sets.

Insertion sort works exceedingly fast on arrays of less than, say, 16 elements, though for large n its computing time is $O(n^2)$. Its basic idea for sorting the items in $a[1 : n]$ is as follows:

```
for j := 2 to n do {
    place a[j] in its correct position in the sorted set a[1 : j − 1];
}
```

Though all the elements in $a[1 : j-1]$ may have to be moved to accommodate $a[j]$, for small values of n the algorithm works well. Algorithm 3.9 has the details.

```
1   Algorithm InsertionSort(a, n)
2   // Sort the array a[1 : n] into nondecreasing order, n ≥ 1.
3   {
4       for j := 2 to n do
5       {
6           // a[1 : j − 1] is already sorted.
7           item := a[j]; i := j − 1;
8           while ((i ≥ 1) and (item < a[i])) do
9           {
10              a[i + 1] := a[i]; i := i − 1;
11          }
12          a[i + 1] := item;
13      }
14  }
```

Algorithm 3.9 Insertion sort

The statements within the **while** loop can be executed zero up to a maximum of j times. Since j goes from 2 to n, the worst-case time of this procedure is bounded by

$$\sum_{2 \le j \le n} j = n(n+1)/2 - 1 = \Theta(n^2)$$

Its best-case computing time is $\Theta(n)$ under the assumption that the body of the **while** loop is never entered. This will be true when the data is already in sorted order.

We are now ready to present the revised version of merge sort with the inclusion of insertion sort and the links. Function MergeSort1 (Algorithm 3.10) is initially invoked by placing the keys of the records to be sorted in $a[1 : n]$ and setting $link[1 : n]$ to zero. Then one says MergeSort1$(1, n)$. A pointer to a list of indices that give the elements of $a[\ ]$ in sorted order is returned. Insertion sort is used whenever the number of items to be sorted is less than 16. The version of insertion sort as given by Algorithm 3.9 needs to be altered so that it sorts $a[low : high]$ into a linked list. Call the altered version InsertionSort1. The revised merging function, Merge1, is given in Algorithm 3.11.

```
1   Algorithm MergeSort1(low, high)
2   // The global array a[low : high] is sorted in nondecreasing order
3   // using the auxiliary array link[low : high]. The values in link
4   // represent a list of the indices low through high giving a[ ] in
5   // sorted order. A pointer to the beginning of the list is returned.
6   {
7       if ((high − low) < 15) then
8           return InsertionSort1(a, link, low, high);
9       else
10      {
11          mid := ⌊(low + high)/2⌋;
12          q := MergeSort1(low, mid);
13          r := MergeSort1(mid + 1, high);
14          return Merge1(q, r);
15      }
16  }
```

Algorithm 3.10 Merge sort using links

Example 3.8 As an aid to understanding this new version of merge sort, suppose we simulate the algorithm as it sorts the eight-element sequence (50, 10, 25, 30, 15, 70, 35, 55). We ignore the fact that less than 16 elements would normally be sorted using InsertionSort. The *link* array is initialized to zero. Table 3.4 shows how the *link* array changes after each call of MergeSort1 completes. On each row the value of p points to the list in *link* that was created by the last completion of Merge1. To the right are the subsets of sorted elements that are represented by these lists. For example, in the last row $p = 2$ which begins the list of links 2, 5, 3, 4, 7, 1, 8, and 6; this implies $a[2] \leq a[5] \leq a[3] \leq a[4] \leq a[7] \leq a[1] \leq a[8] \leq a[6]$. □

EXERCISES

1. Why is it necessary to have the auxiliary array $b[low : high]$ in function Merge? Give an example that shows why in-place merging is inefficient.

2. The worst-case time of procedure MergeSort is $O(n \log n)$. What is its best-case time? Can we say that the time for MergeSort is $\Theta(n \log n)$?

3. A sorting method is said to be *stable* if at the end of the method, identical elements occur in the same order as in the original unsorted

```
1   Algorithm Merge1(q, r)
2   // q and r are pointers to lists contained in the global array
3   // link[0 : n]. link[0] is introduced only for convenience and need
4   // not be initialized. The lists pointed at by q and r are merged
5   // and a pointer to the beginning of the merged list is returned.
6   {
7       i := q; j := r; k := 0;
8       // The new list starts at link[0].
9       while ((i ≠ 0) and (j ≠ 0)) do
10      { // While both lists are nonempty do
11          if (a[i] ≤ a[j]) then
12          { // Find the smaller key.
13              link[k] := i; k := i; i := link[i];
14              // Add a new key to the list.
15          }
16          else
17          {
18              link[k] := j; k := j; j := link[j];
19          }
20      }
21      if (i = 0) then link[k] := j;
22      else link[k] := i;
23      return link[0];
24  }
```

Algorithm 3.11 Merging linked lists of sorted elements

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	
a:	-	50	10	25	30	15	70	35	55	
link:	0	0	0	0	0	0	0	0	0	
q r p										
1 2 2	2	0	1	0	0	0	0	0	0	(10, 50)
3 4 3	3	0	1	4	0	0	0	0	0	(10, 50), (25, 30)
2 3 2	2	0	3	4	1	0	0	0	0	(10, 25, 30, 50)
5 6 5	5	0	3	4	1	6	0	0	0	(10, 25, 30, 50), (15, 70)
7 8 7	7	0	3	4	1	6	0	8	0	(10, 25, 30, 50), (15, 70), (35, 55)
5 7 5	5	0	3	4	1	7	0	8	6	(10, 25, 30, 50) (15, 35, 55, 70)
2 5 2	2	8	5	4	7	3	0	1	6	(10, 15, 25, 30, 35, 50, 55, 70)

MergeSort1 applied to $a[1:8] = (50, 10, 25, 30, 15, 70, 35, 55)$

Table 3.4 Example of *link* array changes

set. Is merge sort a stable sorting method?

4. Suppose $a[1:m]$ and $b[1:n]$ both contain sorted elements in nondecreasing order. Write an algorithm that merges these items into $c[1:m+n]$. Your algorithm should be shorter than Algorithm 3.8 (Merge) since you can now place a large value in $a[m+1]$ and $b[n+1]$.

5. Given a file of n records that are partially sorted as $x_1 \leq x_2 \leq \cdots \leq x_m$ and $x_{m+1} \leq \cdots \leq x_n$, is it possible to sort the entire file in time $O(n)$ using only a small fixed amount of additional storage?

6. Another way to sort a file of n records is to scan the file, merge consecutive pairs of size one, then merge pairs of size two, and so on. Write an algorithm that carries out this process. Show how your algorithm works on the data set (100, 300, 150, 450, 250, 350, 200, 400, 500).

7. A version of insertion sort is used by Algorithm 3.10 to sort small subarrays. However, its parameters and intent are slightly different from the procedure InsertionSort of Algorithm 3.9. Write a version of insertion sort that will work as Algorithm 3.10 expects.

8. The sequences $X_1, X_2, \ldots, X_\ell$ are sorted sequences such that $\sum_{i=1}^{\ell} |X_i| = n$. Show how to merge these ℓ sequences in time $O(n \log \ell)$.

3.5 QUICKSORT

The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. In merge sort, the file $a[1:n]$ was divided

at its midpoint into subarrays which were independently sorted and later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \leq a[j]$ for all i between 1 and m and all j between $m+1$ and n for some m, $1 \leq m \leq n$. Thus, the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of $a[\ ]$, say $t = a[s]$, and then reordering the other elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as *partitioning*.

Function Partition of Algorithm 3.12 (due to C. A. R. Hoare) accomplishes an in-place partitioning of the elements of $a[m:p-1]$. It is assumed that $a[p] \geq a[m]$ and that $a[m]$ is the partitioning element. If $m = 1$ and $p-1 = n$, then $a[n+1]$ must be defined and must be greater than or equal to all elements in $a[1:n]$. The assumption that $a[m]$ is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice. The function Interchange(a, i, j) exchanges $a[i]$ with $a[j]$.

Example 3.9 As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition$(a, 1, 10)$. The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$. □

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Using Hoare's clever method of partitioning a set of elements about a chosen element, we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to the function Partition, two sets S_1 and S_2 are produced. All elements in S_1 are less than or equal

```
1   Algorithm Partition(a, m, p)
2   // Within a[m], a[m + 1], ..., a[p − 1] the elements are
3   // rearranged in such a manner that if initially t = a[m],
4   // then after completion a[q] = t for some q between m
5   // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6   // for q < k < p. q is returned. Set a[p] = ∞.
7   {
8       v := a[m]; i := m; j := p;
9       repeat
10      {
11          repeat
12              i := i + 1;
13          until (a[i] ≥ v);

14          repeat
15              j := j − 1;
16          until (a[j] ≤ v);

17          if (i < j) then Interchange(a, i, j);

18      } until (i ≥ j);

19      a[m] := a[j]; a[j] := v; return j;
20  }

1   Algorithm Interchange(a, i, j)
2   // Exchange a[i] with a[j].
3   {
4       p := a[i];
5       a[i] := a[j]; a[j] := p;
6   }
```

Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

to the elements in S_2. Hence S_1 and S_2 can be sorted independently. Each set is sorted by reusing the function Partition. Algorithm 3.13 describes the complete process.

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

Algorithm 3.13 Sorting by partitioning

In analyzing QuickSort, we count only the number of element comparisons $C(n)$. It is easy to see that the frequency count of other operations is of the same order as $C(n)$. We make the following assumptions: the n elements to be sorted are distinct, and the input distribution is such that the partition element $v = a[m]$ in the call to Partition(a, m, p) has an equal probability of being the ith smallest element, $1 \leq i \leq p - m$, in $a[m : p - 1]$.

First, let us obtain the worst-case value $C_w(n)$ of $C(n)$. The number of element comparisons in each call of Partition is at most $p - m + 1$. Let r be the total number of elements in all the calls to Partition at any level of recursion. At level one only one call, Partition$(a, 1, n+1)$, is made and $r = n$; at level two at most two calls are made and $r = n - 1$; and so on. At each level of recursion, $O(r)$ element comparisons are made by Partition. At each level, r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence $C_w(n)$ is the sum on r as r varies from 2 to n, or $O(n^2)$. Exercise 7 examines input data on which QuickSort uses $\Omega(n^2)$ comparisons.

The average value $C_A(n)$ of $C(n)$ is much less than $C_w(n)$. Under the assumptions made earlier, the partitioning element v has an equal probability

of being the ith-smallest element, $1 \leq i \leq p-m$, in $a[m:p-1]$. Hence the two subarrays remaining to be sorted are $a[m:j]$ and $a[j+1:p-1]$ with probability $1/(p-m), m \leq j < p$. From this we obtain the recurrence

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} [C_A(k-1)) + C_A(n-k)] \tag{3.5}$$

The number of element comparisons required by Partition on its first call is $n+1$. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.5) by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \tag{3.6}$$

Replacing n by $n-1$ in (3.6) gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Subtracting this from (3.6), we get

$$\begin{aligned} nC_A(n) - (n-1)C_A(n-1) &= 2n + 2C_A(n-1) \\ \textit{or} & \\ C_A(n)/(n+1) &= C_A(n-1)/n + 2/(n+1) \end{aligned}$$

Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots$, we get

$$\begin{aligned} \tfrac{C_A(n)}{n+1} &= \tfrac{C_A(n-2)}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\ &= \tfrac{C_A(n-3)}{n-2} + \tfrac{2}{n-1} + \tfrac{2}{n} + \tfrac{2}{n+1} \\ &\vdots \\ &= \tfrac{C_A(1)}{2} + 2\textstyle\sum_{3 \leq k \leq n+1} \tfrac{1}{k} \\ &= 2\textstyle\sum_{3 \leq k \leq n+1} \tfrac{1}{k} \end{aligned} \tag{3.7}$$

Since

$$\sum_{3 \leq k \leq n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2$$

(3.7) yields

$$C_A(n) \leq 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Even though the worst-case time is $O(n^2)$, the average time is only $O(n \log n)$. Let us now look at the stack space needed by the recursion. In the worst case the maximum depth of recursion may be $n-1$. This happens, for example, when the partition element on each call to Partition is the smallest value in $a[m:p-1]$. The amount of stack space needed can be reduced to $O(\log n)$ by using an iterative version of quicksort in which the smaller of the two subarrays $a[p:j-1]$ and $a[j+1:q]$ is always sorted first. Also, the second recursive call can be replaced by some assignment statements and a jump to the beginning of the algorithm. With these changes, QuickSort takes the form of Algorithm 3.14.

We can now verify that the maximum stack space needed is $O(\log n)$. Let $S(n)$ be the maximum stack space needed. Then it follows that

$$S(n) \leq \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

which is less than $2 \log n$.

As remarked in Section 3.4, InsertionSort is exceedingly fast for n less than about 16. Hence InsertionSort can be used to speed up QuickSort2 whenever $q - p < 16$. The exercises explore various possibilities for selection of the partition element.

3.5.1 Performance Measurement

QuickSort and MergeSort were evaluated on a SUN workstation 10/30. In both cases the recursive versions were used. For QuickSort the Partition function was altered to carry out the median of three rule (i.e. the partitioning element was the median of $a[m]$, $a[\lfloor (m+p-1)/2 \rfloor]$ and $a[p-1]$). Each data set consisted of random integers in the range (0, 1000). Tables 3.5 and 3.6 record the actual computing times in milliseconds. Table 3.5 displays the average computing times. For each n, 50 random data sets were used. Table 3.6 shows the worst-case computing times for the 50 data sets.

Scanning the tables, we immediately see that QuickSort is faster than MergeSort for all values. Even though both algorithms require $O(n \log n)$ time on the average, QuickSort usually performs well in practice. The exercises discuss other tests that would make useful comparisons.

3.5.2 Randomized Sorting Algorithms

Though algorithm QuickSort has an average time of $O(n \log n)$ on n elements, its worst-case time is $O(n^2)$. On the other hand it does not make use of any

```
1   Algorithm QuickSort2(p, q)
2   // Sorts the elements in a[p : q].
3   {
4       // stack is a stack of size 2 log(n).
5       repeat
6       {
7           while (p < q) do
8           {
9               j := Partition(a, p, q + 1);
10              if ((j − p) < (q − j)) then
11              {
12                  Add(j + 1); // Add j + 1 to stack.
13                  Add(q); q := j − 1; // Add q to stack
14              }
15              else
16              {
17                  Add(p); // Add p to stack.
18                  Add(j − 1); p := j + 1; // Add j − 1 to stack
19              }
20          } // Sort the smaller subfile.
21          if stack is empty then return;
22          Delete(q); Delete(p); // Delete q and p from stack.
23      } until (false);
24  }
```

Algorithm 3.14 Iterative version of QuickSort

additional memory as does MergeSort. A possible input on which QuickSort displays worst-case behavior is one in which the elements are already in sorted order. In this case the partition will be such that there will be only one element in one part and the rest of the elements will fall in the other part. The performance of any divide-and-conquer algorithm will be good if the resultant subproblems are as evenly sized as possible. Can QuickSort be modified so that it performs well on every input? The answer is yes. Is the technique of using the median of the three elements $a[p]$, $a[\lfloor (q+p)/2 \rfloor]$, and $a[q]$ the solution? Unfortunately it is possible to construct inputs for which even this method will take $\Omega(n^2)$ time, as is explored in the exercises.

The solution is the use of a randomizer. While sorting the array $a[p : q]$, instead of picking $a[m]$, pick a random element (from among $a[p], \ldots, a[q]$) as the partition element. The resultant randomized algorithm (RQuickSort)

n	1000	2000	3000	4000	5000
MergeSort	72.8	167.2	275.1	378.5	500.6
QuickSort	36.6	85.1	138.9	205.7	269.0
n	6000	7000	8000	9000	10000
MergeSort	607.6	723.4	811.5	949.2	1073.6
QuickSort	339.4	411.0	487.7	556.3	645.2

Table 3.5 Average computing times for two sorting algorithms on random inputs

n	1000	2000	3000	4000	5000
MergeSort	105.7	206.4	335.2	422.1	589.9
QuickSort	41.6	97.1	158.6	244.9	397.8
n	6000	7000	8000	9000	10000
MergeSort	691.3	794.8	889.5	1067.2	1167.6
QuickSort	383.8	497.3	569.9	616.2	738.1

Table 3.6 Worst-case computing times for two sorting algorithms on random inputs

works on any input and runs in an expected $O(n \log n)$ time, where the expectation is over the space of all possible outcomes for the randomizer (rather than the space of all possible inputs). The code for RQuickSort is given in Algorithm 3.15. Note that this is a Las Vegas algorithm since it will always output the correct answer. Every call to the randomizer Random takes a certain amount of time. If there are only a very few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation. For this reason, we invoke the randomizer only if $(q - p) > 5$. But 5 is not a magic number; in the machine employed, this seems to give the best results. In general this number should be determined empirically.

```
1   Algorithm RQuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order. a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then
7       {
8           if ((q − p) > 5) then
9               Interchange(a, Random() mod (q − p + 1) + p, p);
10          j := Partition(a, p, q + 1);
11              // j is the position of the partitioning element.
12          RQuickSort(p, j − 1);
13          RQuickSort(j + 1, q);
14      }
15  }
```

Algorithm 3.15 Randomized quick sort algorithm

The proof of the fact that RQuickSort has an expected $O(n \log n)$ time is the same as the proof of the average time of QuickSort. Let $A(n)$ be the average time of RQuickSort on *any input* of n elements. Then the number of elements in the second part will be $0, 1, 2, \ldots, n-2$, or $n-1$, all with an equal probability of $\frac{1}{n}$ (in the probability space of outcomes for the randomizer). Thus the recurrence relation for $A(n)$ will be

$$A(n) = \frac{1}{n} \sum_{1 \le k \le n} (A(k-1) + A(n-k)) + n + 1$$

This is the same as Equation 3.4, and hence its solution is $O(n \log n)$.

RQuickSort and QuickSort (without employing the median of three elements rule) were evaluated on a SUN 10/30 workstation. Table 3.7 displays

the times for the two algorithms in milliseconds averaged over 100 runs. For each n, the input considered was the sequence of numbers $1, 2, \ldots, n$. As we can see from the table, RQuickSort performs much better than QuickSort. Note that the times shown in this table for QuickSort are much more than the corresponding entries in Tables 3.5 and 3.6. The reason is that QuickSort makes $\Theta(n^2)$ comparisons on inputs that are already in sorted order. However, on random inputs its average performance is very good.

n	1000	2000	3000	4000	5000
QuickSort	195.5	759.2	1728	3165	4829
RQuickSort	9.4	21.0	30.5	41.6	52.8

Table 3.7 Comparison of QuickSort and RQuickSort on the input $a[i] = i$, $1 \leq i \leq n$; times are in milliseconds.

The performance of RQuickSort can be improved in various ways. For example, we could pick a small number (say 11) of the elements in the array $a[\,]$ randomly and use the median of these elements as the partition element. These randomly chosen elements form a random sample of the array elements. We would expect that the median of the sample would also be an approximate median of the array and hence result in an approximately even partitioning of the array.

An even more generalized version of the above random sampling technique is shown in Algorithm 3.16. Here we choose a random sample S of s elements (where s is a function of n) from the input sequence X and sort them using HeapSort, MergeSort, or any other sorting algorithm. Let $\ell_1, \ell_2, \ldots, \ell_s$ be the sorted sample. We partition X into $s+1$ parts using the sorted sample as partition keys. In particular $X_1 = \{x \in X | x \leq \ell_1\}$; $X_i = \{x \in X | \ell_{i-1} < x \leq \ell_i\}$, for $i = 2, 3, \ldots, s$; and $X_{s+1} = \{x \in X | x > \ell_s\}$. After having partitioned X into $s+1$ parts, we sort each part recursively. For a proper choice of s, the number of comparisons made in this algorithm is only $n \log n + \tilde{o}(n \log n)$. Note the constant 1 before $n \log n$. We see in Chapter 10 that this number is very close to the information theoretic lower bound for sorting.

Choose $s = \frac{n}{\log^2 n}$. The sample can be sorted in $O(s \log s) = O(\frac{n}{\log n})$ time and comparisons if we use HeapSort or MergeSort. If we store the sorted sample elements in an array, say $b[\,]$, for each $x \in X$, we can determine which part X_i it belongs to in $\leq \log n$ comparisons using binary search on $b[\,]$. Thus the partitioning process takes $n \log n + O(n)$ comparisons. In the exercises you are asked to show that with high probability the cardinality

```
1   Algorithm RSort(a, n)
2   // Sort the elements a[1 : n].
3   {
4       Randomly sample s elements from a[ ];
5       Sort this sample;
6       Partition the input using the sorted sample as partition keys;
7       Sort each part separately;
8   }
```

Algorithm 3.16 A randomized algorithm for sorting

of each X_i is no more than $\widetilde{O}(\frac{n}{s}\log n) = \widetilde{O}(\log^3 n)$. Using HeapSort or MergeSort to sort each of the X_i's (without employing recursion on any of them), the total cost of sorting the X_i's is

$$\sum_{i=1}^{s+1} O(|X_i|\log|X_i|) = \max_{1\le i\le s+1}\{\log|X_i|\}\sum_{i=1}^{s+1} O(|X_i|)$$

Since each $|X_i|$ is $\widetilde{O}(\log^3 n)$, the cost of sorting the $s+1$ parts is $\widetilde{O}(n\log\log n) = \widetilde{o}(n\log n)$. In summary, the number of comparisons made in this randomized sorting algorithm is $n\log n + \widetilde{o}(n\log n)$.

EXERCISES

1. Show how QuickSort sorts the following sequences of keys: 1, 1, 1, 1, 1, 1, 1 and 5, 5, 8, 3, 4, 3, 2.

2. QuickSort is not a stable sorting algorithm. However, if the key in $a[i]$ is changed to $a[i] * n + i - 1$, then the new keys are all distinct. After sorting, which transformation will restore the keys to their original values?

3. In the function Partition, Algorithm 3.12, discuss the merits or demerits of altering the statement **if** $(i < j)$ to **if** $(i \le j)$. Simulate both algorithms on the data set (5, 4, 3, 2, 5, 8, 9) to see the difference in how they work.

4. Function QuickSort uses the output of function Partition, which returns the position where the partition element is placed. If equal keys are present, then two elements can be properly placed instead of one. Show

how you might change Partition so that QuickSort can take advantage of this situation.

5. In addition to Partition, there are many other ways to partition a set. Consider modifying Partition so that i is incremented while $a[i] \leq v$ instead of $a[i] < v$. Rewrite Partition making all of the necessary changes to it and then compare the new version with the original.

6. Compare the sorting methods MergeSort1 and QuickSort2 (Algorithm 3.10 and 3.14, respectively). Devise data sets that compare both the average- and worst-case times for these two algorithms.

7. (a) On which input data does the algorithm QuickSort exhibit its worst-case behavior?

   (b) Answer part (a) for the case in which the partitioning element is selected according to the median of three rule.

8. With MergeSort we included insertion sorting to eliminate the bookkeeping for small merges. How would you use this technique to improve QuickSort?

9. Take the iterative versions of MergeSort and QuickSort and compare them for the same-size data sets as used in Section 3.5.1.

10. Let S be a sample of s elements from X. If X is partitioned into $s+1$ parts as in Algorithm 3.16, show that the size of each part is $\tilde{O}(\frac{n}{s}\log n)$.

3.6 SELECTION

The Partition algorithm of Section 3.5 can also be used to obtain an efficient solution for the selection problem. In this problem, we are given n elements $a[1:n]$ and are required to determine the kth-smallest element. If the partitioning element v is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the kth-smallest element is in $a[1:j-1]$; if $k = j$, then $a[j]$ is the kth-smallest element; and if $k > j$, then the kth-smallest element is the $(k-j)$th-smallest element in $a[j+1:n]$. The resulting algorithm is function Select1 (Algorithm 3.17). This function places the kth-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \leq a[k]$, $1 \leq i < k$, and $a[i] \geq a[k]$, $k < i \leq n$.

Example 3.10 Let us simulate Select1 as it operates on the same array used to test Partition in Section 3.5. The array has the nine elements 65, 70,

```
1   Algorithm Select1(a, n, k)
2   // Selects the kth-smallest element in a[1 : n] and places it
3   // in the kth position of a[ ]. The remaining elements are
4   // rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
5   // a[m] ≥ a[k] for k < m ≤ n.
6   {
7       low := 1; up := n + 1;
8       a[n + 1] := ∞; // a[n + 1] is set to infinity.
9       repeat
10      {
11          // Each time the loop is entered,
12          // 1 ≤ low ≤ k ≤ up ≤ n + 1.
13          j := Partition(a, low, up);
14                  // j is such that a[j] is the jth-smallest value in a[ ].
15          if (k = j) then return;
16          else if (k < j) then up := j; // j is the new upper limit.
17                  else low := j + 1; // j + 1 is the new lower limit.
18      } until (false);
19  }
```

Algorithm 3.17 Finding the kth-smallest element

75, 80, 85, 60, 55, 50, and 45, with $a[10] = \infty$. If $k = 5$, then the first call of Partition will be sufficient since 65 is placed into $a[5]$. Instead, assume that we are looking for the seventh-smallest element of a, that is, $k = 7$. The next invocation of Partition is Partition(6, 10).

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	85	80	75	70	$+\infty$
	65	70	80	75	85	$+\infty$

This last call of Partition has uncovered the ninth-smallest element of a. The next invocation is Partition(6, 9).

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	80	75	85	$+\infty$

This time, the sixth element has been found. Since $k \neq j$, another call to Partition is made, Partition(7, 9).

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	75	80	85	$+\infty$

Now 80 is the partition value and is correctly placed at $a[8]$. However, Select1 has still not found the seventh-smallest element. It needs one more call to Partition, which is Partition$(7, 8)$. This performs only an interchange between $a[7]$ and $a[8]$ and returns, having found the correct value. □

In analyzing Select1, we make the same assumptions that were made for QuickSort:

1. The n elements are distinct.
2. The input distribution is such that the partition element can be the ith-smallest element of $a[m : p-1]$ with an equal probability for each i, $1 \leq i \leq p - m$.

Partition requires $O(p - m)$ time. On each successive call to Partition, either m increases by at least one or j decreases by at least one. Initially $m = 1$ and $j = n + 1$. Hence, at most n calls to Partition can be made. Thus, the worst-case complexity of Select1 is $O(n^2)$. The time is $\Omega(n^2)$, for example, when the input $a[1 : n]$ is such that the partitioning element on the ith call to Partition is the ith-smallest element and $k = n$. In this case, m increases by one following each call to Partition and j remains unchanged. Hence, n calls are made for a total cost of $O(\sum_1^n i) = O(n^2)$. The average computing time of Select1 is, however, only $O(n)$. Before proving this fact, we specify more precisely what we mean by the average time.

Let $T_A^k(n)$ be the average time to find the kth-smallest element in $a[1 : n]$. This average is taken over all $n!$ different permutations of n distinct elements. Now define $T_A(n)$ and $R(n)$ as follows:

$$T_A(n) = \frac{1}{n} \sum_{1 \leq k \leq n} T_A^k(n)$$

and

$$R(n) = \max_k \{T_A^k(n)\}$$

$T_A(n)$ is the average computing time of Select1. It is easy to see that $T_A(n) \leq R(n)$. We are now ready to show that $T_A(n) = O(n)$.

Theorem 3.3 The average computing time $T_A(n)$ of Select1 is $O(n)$.

Proof: On the first call to Partition, the partitioning element v is the ith-smallest element with probability $\frac{1}{n}, 1 \leq i \leq n$ (this follows from the assumption on the input distribution). The time required by Partition and the **if** statement in Select1 is $O(n)$. Hence, there is a constant $c, c > 0$, such that

$$\begin{aligned} T_A^k(n) &\leq cn + \frac{1}{n}[\sum_{1 \leq i < k} T_A^{k-1}(n-i) + \sum_{k < i \leq n} T_A^k(i-1)], \quad n \geq 2 \\ \text{So, } R(n) &\leq cn + \frac{1}{n} \max_k \{\sum_{1 \leq i < k} R(n-i) + \sum_{k < i \leq n} R(i-1)\} \\ R(n) &\leq cn + \frac{1}{n} \max_k \{\sum_{n-k+1}^{n-1} R(i) + \sum_{k}^{n-1} R(i)\}, \quad n \geq 2 \end{aligned} \tag{3.8}$$

We assume that c is chosen such that $R(1) \leq c$ and show, by induction on n, that $R(n) \leq 4cn$.
Induction Base: For $n = 2$, (3.8) gives

$$\begin{aligned} R(n) &\leq 2c + \frac{1}{2} \max \{R(1), R(1)\} \\ &\leq 2.5c < 4cn \end{aligned}$$

Induction Hypothesis: Assume $R(n) \leq 4cn$ for all $n, 2 \leq n < m$.
Induction Step: For $n = m$, (3.8) gives

$$R(m) \leq cm + \frac{1}{m} \max_k \left\{ \sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i) \right\}$$

Since we know that $R(n)$ is a nondecreasing function of n, it follows that

$$\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)$$

is maximized if $k = \frac{m}{2}$ when m is even and $k = \frac{m+1}{2}$ when m is odd. Thus, if m is even, we obtain

$$R(m) \leq cm + \frac{2}{m} \sum_{m/2}^{m-1} R(i)$$

$$
\begin{aligned}
&\leq cm + \frac{8c}{m}\sum_{m/2}^{m-1} i \\
&< 4cm \\
\text{If } m \text{ is odd, } R(m) &\leq cm + \frac{2}{m}\sum_{(m+1)/2}^{m-1} R(i) \\
&\leq cm + \frac{8c}{m}\sum_{(m+1)/2}^{m-1} \\
&< 4cm
\end{aligned}
$$

Since $T_A(n) \leq R(n)$, it follows that $T_A(n) \leq 4cn$, and so $T_A(n)$ is $O(n)$. □

The space needed by Select1 is $O(1)$.

Algorithm 3.15 is a randomized version of QuickSort in which the partition element is chosen from the array elements randomly with equal probability. The same technique can be applied to Select1 and the partition element can be chosen to be a random array element. The resulting randomized Las Vegas algorithm (call it RSelect) has an expected time of $O(n)$ (where the expectation is over the space of randomizer outputs) on *any input.* The proof of this expected time is the same as in Theorem 3.3.

3.6.1 A Worst-Case Optimal Algorithm

By choosing the partitioning element v more carefully, we can obtain a selection algorithm with worst-case complexity $O(n)$. To obtain such an algorithm, v must be chosen so that at least some fraction of the elements is smaller than v and at least some (other) fraction of elements is greater than v. Such a selection of v can be made using the median of medians (mm) rule. In this rule the n elements are divided into $\lfloor n/r \rfloor$ groups of r elements each (for some $r, r > 1$). The remaining $n - r\lfloor n/r \rfloor$ elements are not used. The median m_i of each of these $\lfloor n/r \rfloor$ groups is found. Then, the median mm of the m_i's, $1 \leq i \leq \lfloor n/r \rfloor$, is found. The median mm is used as the partitioning element. Figure 3.5 illustrates the m_i's and mm when $n = 35$ and $r = 7$. The five groups of elements are $B_i, 1 \leq i \leq 5$. The seven elements in each group have been arranged into nondecreasing order down the column. The middle elements are the m_i's. The columns have been arranged in nondecreasing order of m_i. Hence, the m_i corresponding to column 3 is mm.

Since the median of r elements is the $\lceil r/2 \rceil$th-smallest element, it follows (see Figure 3.5) that at least $\lceil \lfloor n/r \rfloor /2 \rceil$ of the m_i's are less than or equal to mm and at least $\lfloor n/r \rfloor - \lceil \lfloor n/r \rfloor /2 \rceil + 1 \geq \lceil \lfloor n/r \rfloor /2 \rceil$ of the m_i's are greater than or equal to mm. Hence, at least $\lceil r/2 \rceil \lceil \lfloor n/r \rfloor /2 \rceil$ elements are less than

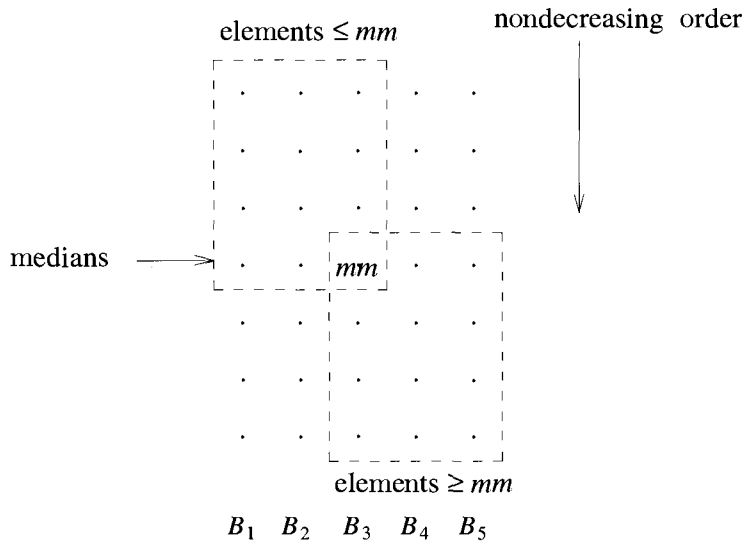


Figure 3.5 The median of medians when $r = 7$, $n = 35$

or equal to (or greater than or equal to) mm. When $r = 5$, this quantity is at least $1.5\lfloor n/5 \rfloor$. Thus, if we use the median of medians rule with $r = 5$ to select $v = mm$, we are assured that at least $1.5\lfloor n/5 \rfloor$ elements will be greater than or equal to v. This in turn implies that at most $n - 1.5\lfloor n/5 \rfloor \leq .7n + 1.2$ elements are less than v. Also, at most $.7n + 1.2$ elements are greater than v. Thus, the median of medians rule satisfies our earlier requirement on v.

The algorithm to select the kth-smallest element uses the median of medians rule to determine a partitioning element. This element is computed by a recursive application of the selection algorithm. A high-level description of the new selection algorithm appears as Select2 (Algorithm 3.18). Select2 can now be analyzed for any given r. First, let us consider the case in which $r = 5$ and all elements in $a[\ ]$ are distinct. Let $T(n)$ be the worst-case time requirement of Select2 when invoked with $up - low + 1 = n$. Lines 4 to 9 and 11 to 12 require at most $O(n)$ time (note that since $r = 5$ is fixed, each $m[i]$ (lines 8 and 9) can be found in $O(1)$ time). The time for line 10 is $T(n/5)$. Let S and R, respectively, denote the elements $a[low : j-1]$ and $a[j+1 : up]$. We see that $|S|$ and $|R|$ are at most $.7n + 1.2$, which is no more than $3n/4$ for $n \geq 24$. So, the time for lines 13 to 16 is at most $T(3n/4)$ when $n \geq 24$. Hence, for $n \geq 24$, we obtain

```
1   Algorithm Select2(a, k, low, up)
2   // Find the k-th smallest in a[low : up].
3   {
4       n := up − low + 1;
5       if (n ≤ r) then sort a[low : up] and return the k-th element;
6       Divide a[low : up] into n/r subsets of size r each;
7       Ignore excess elements;
8       Let m[i], 1 ≤ i ≤ (n/r) be the set of medians of
9       the above n/r subsets.
10      v := Select2(m, ⌈(n/r)/2⌉, 1, n/r);
11      Partition a[low : up] using v as the partition element;
12      Assume that v is at position j;
13      if (k = (j − low + 1)) then return v;
14      elseif (k < (j − low + 1)) then
15              return Select2(a, k, low, j − 1);
16          else return Select2(a, k − (j − low + 1), j + 1, up);
17  }
```

Algorithm 3.18 Selection pseudocode using the median of medians rule

$$T(n) \leq T(n/5) + T(3n/4) + cn \tag{3.9}$$

where c is chosen sufficiently large that

$$T(n) \leq cn \quad \text{for } n \leq 24$$

A proof by induction easily establishes that $T(n) \leq 20cn$ for $n \geq 1$. Algorithm Select2 with $r = 5$ is a linear time algorithm for the selection problem on distinct elements! The exercises examine other values of r that also yield this behavior. Let us now see what happens when the elements of $a[\,]$ are not all distinct. In this case, following a use of Partition (line 11), the size of S or R may be more than $.7n + 1.2$ as some elements equal to v may appear in both S and R. One way to handle the situation is to partition $a[\,]$ into three sets U, S, and R such that U contains all elements equal to v, S has all elements smaller than v, and R has the remainder. Lines 11 to 16 become:

```
Partition a[ ] into U, S, and R as above.
if (|S| ≥ k) then return Select2(a, k, low, low + |S| − 1);
else if ((|S| + |U|) ≥ k) then return v;
     else return Select2(a, k − |S| − |U|, low + |S| + |U|, up);
```

When this is done, the recurrence (3.9) is still valid as $|S|$ and $|R|$ are $\leq .7n + 1.2$. Hence, the new Select2 will be of linear complexity even when elements are not distinct.

Another way to handle the case of nondistinct elements is to use a different r. To see why a different r is needed, let us analyze Select2 with $r = 5$ and nondistinct elements. Consider the case when $.7n+1.2$ elements are less than v and the remaining elements are equal to v. An examination of Partition reveals that at most half the remaining elements may be in S. We can verify that this is the worst case. Hence, $|S| \leq .7n + 1.2 + (.3n - 1.2)/2 = .85n + .6$. Similarly, $|R| \leq .85n + .6$. Since, the total number of elements involved in the two recursive calls (in lines 10 and 15 or 16) is now $1.05n + .6 \geq n$, the complexity of Select2 is not $O(n)$. If we try $r = 9$, then at least $2.5\lfloor n/9 \rfloor$ elements will be less than or equal to v and at least this many will be greater than or equal to v. Hence, the size of S and R will be at most $n - 2.5\lfloor n/9 \rfloor + 1/2(2.5\lfloor n/9 \rfloor) = n - 1.25\lfloor n/9 \rfloor \leq 31/36n + 1.25 \leq 63n/72$ for $n \geq 90$. Hence, we obtain the recurrence

$$T(n) \leq \begin{cases} T(n/9) + T(63n/72) + c_1 n & n \geq 90 \\ c_1 n & n < 90 \end{cases}$$

where c_1 is a suitable constant. An inductive argument shows that $T(n) \leq 72c_1 n, n \geq 1$. Other suitable values of r are obtained in the exercises.

As far as the additional space needed by Select2 is concerned, we see that space is needed for the recursion stack. The recursive call from line 15 or 16 is easily eliminated as this call is the last statement executed in Select2. Hence, stack space is needed only for the recursion from line 10. The maximum depth of recursion is log n. The recursion stack should be capable of handling this depth. In addition to this stack space, space is needed only for some simple variables.

3.6.2 Implementation of Select2

Before attempting to write a pseudocode algorithm implementing Select2, we need to decide how the median of a set of size r is to be found and where we are going to store the $\lfloor n/r \rfloor$ medians of lines 8 and 9. Since, we expect to be using a small r (say $r = 5$ or 9), an efficient way to find the median of r elements is to sort them using InsertionSort(a, i, j). This algorithm is a modification of Algorithm 3.9 to sort $a[i : j]$. The median is now the middle element in $a[i : j]$. A convenient place to store these medians is at the front of the array. Thus, if we are finding the kth-smallest element in $a[low : up]$, then the elements can be rearranged so that the medians are $a[low]$, $a[low+1]$, $a[low+2]$, and so on. This makes it easy to implement line 10 as a selection on consecutive elements of $a[\,]$. Function Select2 (Algorithm 3.19) results from the above discussion and the replacement of the recursive calls of lines 15 and 16 by equivalent code to restart the algorithm.

```
1   Algorithm Select2(a, k, low, up)
2   // Return i such that a[i] is the kth-smallest element in
3   // a[low : up]; r is a global variable as described in the text.
4   {
5       repeat
6       {
7           n := up − low + 1; // Number of elements
8           if (n ≤ r) then
9           {
10              InsertionSort(a, low, up);
11              return low + k − 1;
12          }
13          for i := 1 to ⌊n/r⌋ do
14          {
15              InsertionSort(a, low + (i − 1) ∗ r, low + i ∗ r − 1);
16              // Collect medians in the front part of a[low : up].
17              Interchange(a, low + i − 1,
18                  low + (i − 1) ∗ r + ⌈r/2⌉ − 1);
19          }
20          j := Select2(a, ⌈⌊n/r⌋/2⌉, low, low + ⌊n/r⌋ − 1); // mm
21          Interchange(a, low, j);
22          j := Partition(a, low, up + 1);
23          if (k = (j − low + 1)) then return j;
24          else if (k < (j − low + 1)) then up := j − 1;
25              else
26              {
27                  k := k − (j − low + 1); low := j + 1;
28              }
29      } until (false);
30  }
```

Algorithm 3.19 Algorithm Select2

An alternative to moving the medians to the front of the array $a[low : up]$ (as in the Interchange statement within the **for** loop) is to delete this statement and use the fact that the medians are located at $low + (i-1)r + \lceil r/2 \rceil - 1, 1 \leq i \leq \lfloor n/r \rfloor$. Hence, Select2, Partition, and InsertionSort need to be rewritten to work on arrays for which the interelement distance is $b, b \geq 1$. At the start of the algorithm, all elements are a distance of one apart, i.e., $a[1], a[2], \ldots, a[n]$. On the first call of Select2 we wish to use only elements that are r apart starting with $a[\lceil r/2 \rceil]$. At the next level of recursion, the elements will be r^2 apart and so on. This idea is developed further in the exercises. We refer to arrays with an interelement distance of b as *b-spaced arrays.*

Algorithms Select1 (Algorithm 3.17) and Select2 (Algorithm 3.19) were implemented and run on a SUN Sparcstation 10/30. Table 3.8 summarizes the experimental results obtained. Times shown are in milliseconds. These algorithms were tested on random integers in the range [0, 1000] and the average execution times (over 500 input sets) were computed. Select1 outperforms Select2 on random inputs. But if the input is already sorted (or nearly sorted), Select2 can be expected to be superior to Select1.

n	1,000	2,000	3,000	4,000	5,000
Select1	7.42	23.50	30.44	39.24	52.36
Select2	49.54	104.02	174.54	233.56	288.64
n	6,000	7,000	8,000	9,000	10,000
Select1	70.88	83.14	95.00	101.32	111.92
Select2	341.34	414.06	476.98	532.30	604.40

Table 3.8 Comparison of Select1 and Select2 on random inputs

EXERCISES

1. Rewrite Select2, Partition, and InsertionSort using b-spaced arrays.

2. (a) Assume that Select2 is to be used only when all elements in a are distinct. Which of the following values of r guarantee $O(n)$ worst-case performance: $r = 3, 5, 7, 9$, and 11? Prove your answers.

   (b) Do you expect the computing time of Select2 to increase or decrease if a larger (but still eligible) choice for r is made? Why?

3. Do Exercise 2 for the case in which a is not restricted to distinct elements. Let $r = 7, 9, 11, 13$, and 15 in part (a).

4. Section 3.6 describes an alternative way to handle the situation when $a[\ ]$ is not restricted to distinct elements. Using the partitioning element v, $a[\ ]$ is divided into three subsets. Write algorithms corresponding to Select1 and Select2 using this idea. Using your new version of Select2 show that the worst-case computing time is $O(n)$ even when $r = 5$.

5. Determine optimal r values for worst-case and average performances of function Select2.

6. [Shamos] Let $x[1 : n]$ and $y[1 : n]$ contain two sets of integers, each sorted in nondecreasing order. Write an algorithm that finds the median of the $2n$ combined elements. What is the time complexity of your algorithm? (*Hint*: Use binary search.)

7. Let S be a (not necessarily sorted) sequence of n keys. A key k in S is said to be an *approximate median* of S if $|\{k' \in S : \ k' < k\}| \geq \frac{n}{4}$ and $|\{k' \in S : \ k' > k\}| \geq \frac{n}{4}$. Devise an $O(n)$ time algorithm to find all the approximate medians of S.

8. Input are a sequence S of n distinct keys, not necessarily in sorted order, and two integers m_1 and m_2 $(1 \leq m_1, m_2 \leq n)$. For any x in S, we define the *rank* of x in S to be $|\{k \in S : \ k \leq x\}|$. Show how to output all the keys of S whose ranks fall in the interval $[m_1, m_2]$ in $O(n)$ time.

9. The kth *quantiles* of an n-element set are the $k - 1$ elements from the set that divide the sorted set into k equal-sized sets. Give an $O(n \log k)$ time algorithm to list the kth quantiles of a set.

10. Input is a (not necessarily sorted) sequence $S = k_1, k_2, \ldots, k_n$ of n arbitrary numbers. Consider the collection C of n^2 numbers of the form $\min\{k_i, k_j\}$, for $1 \leq i, j \leq n$. Present an $O(n)$-time and $O(n)$-space algorithm to find the median of C.

11. Given two vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$, $X < Y$ if there exists an $i, 1 \leq i \leq n$, such that $x_j = y_j$ for $1 \leq j < i$ and $x_i < y_i$. Given m vectors each of size n, write an algorithm that determines the minimum vector. Analyze the time complexity of your algorithm.

12. Present an $O(1)$ time Monte Carlo algorithm to find the median of an array of n numbers. The answer output should be correct with probability $\geq \frac{1}{n}$.

13. Input is an array $a[\ ]$ of n numbers. Present an $O(\log n)$ time Monte Carlo algorithm to output any member of $a[\ ]$ that is greater than or equal to the median. The answer should be correct with high probability. Provide a probability analysis.

14. Given a set X of n numbers, how will you find an element of X whose rank in X is at most $\frac{n}{f(n)}$, using a Monte Carlo algorithm? Your algorithm should run in time $O(f(n)\log n)$. Prove that the output will be correct with high probability.

15. In addition to Select1 and Select2, we can think of at least two more selection algorithms. The first of these is very straightforward and appears as Algorithm 3.20 (Algorithm Select3). The time complexity of Select3 is

$$O(n \min\{k, n-k+1\})$$

Hence, it is very fast for values of k close to 1 or close to n. In the worst case, its complexity is $O(n^2)$. Its average complexity is also $O(n^2)$.

Another selection algorithm proceeds by first sorting the n elements into nondecreasing order and then picking out the kth element. A complete sort can be avoided by using a minheap. Now, only k elements need to be removed from the heap. The time to set up the heap is $O(n)$. An additional $O(k \log n)$ time is needed to make k deletions. The total complexity is $O(n + k \log n)$. This basic algorithm can be improved further by using a maxheap when $k > n/2$ and deleting $n - k + 1$ elements. The complexity is now $O(n + \log n \min\{k, n-k+1\})$. Call the resulting algorithm Select4. Now that we have four plausible selection algorithms, we would like to know which is best. On the basis of the asymptotic analyses of the four selection algorithms, we can make the following qualitative statements about our expectations on the relative performance of the four algorithms.

- Because of the overhead involved in Select1, Select2, and Select4 and the relative simplicity of Select3, Select3 will be fastest both on the average and in the worst case for small values of n. It will also be fastest for large n and very small or very large k, for example, $k = 1, 2, n$, or $n - 1$.
- For larger values of n, Select1 will have the best behavior on the average.
- As far as worst-case behavior is concerned, Select2 will out-perform the others when n is suitably large. However, there will probably be a range of n for which Select4 will be faster than both Select2 and Select3. We expect this because of the relatively large

```
1   Algorithm Select3(a, n, k)
2   // Rearrange a[ ] such that a[k] is the k-th smallest.
3   {
4       if (k ≤ ⌊n/2⌋) then
5           for i := 1 to k do
6           {
7               q := i; min := a[i];
8               for j := i + 1 to n do
9                   if (a[j] < min) then
10                  {
11                      q := j; min := a[j];
12                  }
13              Interchange(a, q, i);
14          }
15      else
16          for i := n to k step −1 do
17          {
18              q := i; max := a[i];
19              for j := (i − 1) to 1 step −1 do
20                  if (a[j] > max) then
21                  {
22                      q := j; max := a[j];
23                  }
24              Interchange(a, q, i);
25          }
26  }
```

Algorithm 3.20 Straightforward selection algorithm

overhead in Select2 (i.e., the constant term in $O(n)$ is relatively large).

- As a result of the above assertions, it is desirable to obtain composite algorithms for good average and worst-case performances. The composite algorithm for good worst-case performance will have the form of function Select2 but will include the following after the first **if** statement.

```
if (n < c1) then return Select3(a, m, p, k);
else if (n < c2) then return Select4(a, m, p, k);
```

Since the overheads in Select1 and Select4 are about the same, the constants associated with the average computing times will be about

the same. Hence, Select1 may always be better than Select4 or there may be a small c_3 such that Select4 is better than Select1 for $n < c_3$. In any case, we expect there is a $c_4, c_4 > 0$, such that Select3 is faster than Select1 on the average for $n < c_4$.

To verify the preceding statements and determine c_1, c_2, c_3, and c_4, it is necessary to program the four algorithms in some programming language and run the four corresponding programs on a computer. Once the programs have been written, test data are needed to determine average and worst-case computing times. So, let us now say something about the data needed to obtain computing times from which $c_i, 1 \leq i \leq 4$, can be determined. Since we would also like information regarding the average and worst-case computing times of the resulting composite algorithms, we need test data for this too. We limit our testing to the case of distinct elements.

To obtain worst-case computing times for Select1, we change the algorithm slightly. This change will not affect its worst-case computing time but will enable us to use a rather simple data set to determine this time for various values of n. We dispense with the random selection rule for Partition and instead use $a[m]$ as the partitioning element. It is easy to see that the worst-case time is obtained with $a[i] = i$, $1 \leq i \leq n$, and $k = n$. As far as the average time for any given n is concerned, it is not easy to arrive at one data set and a k that exhibits this time. On the other hand, trying out all $n!$ different input permutations and $k = 1, 2, \ldots, n$ for each of these is not a feasible way to find the average. An approximation to the average computing time can be obtained by trying out a few (say ten) random permutations of the numbers $1, 2, \ldots, n$ and for each of these using a few (say five) random values of k. The average of the times obtained can be used as an approximation to the average computing time. Of course, using more permutations and more k values results in a better approximation. However, the number of permutations and k values we can use is limited by the amount of computational resources (in terms of time) we have available.

For Select2, the average time can be obtained in the same way as for Select1. For the worst-case time we can either try to figure out an input permutation for which the number of elements less than the median of medians is always as large as possible and then use $k = 1$. A simpler approach is to find just an approximation to the worst-case time. This can be done by taking the max of the computing times for all the tests used to obtain the average computing time. Since the computing times for Select2 vary with r, it is first necessary to determine an r that yields optimum behavior. Note that the r's for optimum average and worst-case behaviors may be different.

We can verify that the worst-case data for Select3 are $a[i] = n + 1 - i$, for $1 \leq i \leq n$, and $k = \frac{n}{2}$. The computing time for Select3 is relatively insensitive to the input permutation. This permutation affects only the number of times the second **if** statement of Algorithm 3.20 is executed. On the average, this will be done about half the time. This can be achieved by using $a[i] = n + 1 - i$, $1 \leq i \leq n/2$, and $a[i] = n + 1$, $n/2 < i \leq n$. The k value needed to obtain the average computing time is readily seen to be $n/4$.

(a) What test data would you use to determine worst-case and average times for Select4?

(b) Use the ideas above to obtain a table of worst-case and average times for Select1, Select2, Select3, and Select4.

16. Program Select1 and Select3. Determine when algorithm Select1 becomes better than Select3 on the average and also when Select2 better than Select3 for worst-case performance.

17. [Project] Program the algorithms of Exercise 4 as well as Select3 and Select4. Carry out a complete test along the lines discussed in Exercise 15. Write a detailed report together with graphs explaining the data sets, test strategies, and determination of $c_1, \ldots, c_4$. Write the final composite algorithms and give tables of computing times for these algorithms.

3.7 STRASSEN'S MATRIX MULTIPLICATION

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i,jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get

$$C(i,j) = \sum_{1 \leq k \leq n} A(i,k)B(k,j) \tag{3.10}$$

for all i and j between 1 and n. To compute $C(i,j)$ using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

The divide-and-conquer strategy suggests another way to compute the product of two $n \times n$ matrices. For simplicity we assume that n is a power of 2, that is, that there exists a nonnegative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two

(see the exercises for more on this subject). Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. Then the product AB can be computed by using the above formula for the product of 2×2 matrices: if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{3.11}$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{3.12}$$

If $n = 2$, then formulas (3.11) and (3.12) are computed using a multiplication operation for the elements of A and B. These elements are typically floating point numbers. For $n > 2$, the elements of C can be computed using *matrix* multiplication and addition operations applied to matrices of size $n/2 \times n/2$. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm we are using for the $n \times n$ case. This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small ($n = 2$) so that the product is computed directly.

To compute AB using (3.12), we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c, the overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}'s of (3.12) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U, and V as in (3.13). Then the C_{ij}'s are computed using the formulas in (3.14). As can be seen, P, Q, R, S, T, U, and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \tag{3.13}$$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned} \tag{3.14}$$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \tag{3.15}$$

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n},\ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

EXERCISES

1. Verify by hand that Equations 3.13 and 3.14 yield the correct values for C_{11}, C_{12}, C_{21}, and C_{22}.

2. Write an algorithm that multiplies two $n \times n$ matrices using $O(n^3)$ operations. Determine the precise number of multiplications, additions, and array element accesses.

3. If k is a nonnegative constant, then prove that the recurrence

$$T(n) = \begin{cases} k & n = 1 \\ 3T(n/2) + kn & n > 1 \end{cases} \tag{3.16}$$

has the following solution (for n a power of 2):

$$T(n) = 3kn^{\log_2 3} - 2kn \tag{3.17}$$

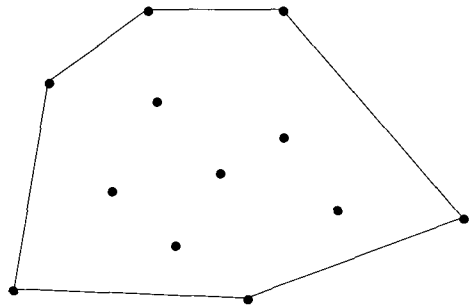


Figure 3.6 Convex hull: an example

(1) obtain the vertices of the convex hull (these vertices are also called *extreme points*), and (2) obtain the vertices of the convex hull in some order (clockwise, for example).

Here is a simple algorithm for obtaining the extreme points of a given set S of points in the plane. To check whether a particular point $p \in S$ is extreme, look at each possible triplet of points and see whether p lies in the triangle formed by these three points. If p lies in any such triangle, it is not extreme; otherwise it is. Testing whether p lies in a given triangle can be done in $\Theta(1)$ time (using the methods described in Section 3.8.1). Since there are $\Theta(n^3)$ possible triangles, it takes $\Theta(n^3)$ time to determine whether a given point is an extreme point or not. Since there are n points, this algorithm runs in a total of $\Theta(n^4)$ time.

Using divide-and-conquer, we can solve both versions of the convex hull problem in $\Theta(n \log n)$ time. We develop three algorithms for the convex hull in this section. The first has a worst-case time of $\Theta(n^2)$ whereas its average time is $\Theta(n \log n)$. This algorithm has a divide-and-conquer structure similar to that of QuickSort. The second has a worst-case time complexity of $\Theta(n \log n)$ and is not based on divide-and-conquer. The third algorithm is based on divide-and-conquer and has a time complexity of $\Theta(n \log n)$ in the worst case. Before giving further details, we digress to discuss some primitive geometric methods that are used in the convex hull algorithms.

3.8.1 Some Geometric Primitives

Let A be an $n \times n$ matrix whose elements are $\{a_{ij}\}$, $1 \leq i, j \leq n$. The ijth *minor* of A, denoted as A_{ij}, is defined to be the submatrix of A obtained by deleting the ith row and jth column. The *determinant* of A, denoted

$\det(A)$, is given by

$$\det(A) = \begin{cases} a_{11} & n = 1 \\ a_{11}\ \det(A_{11}) - a_{12}\ \det(A_{12}) + \cdots + (-1)^{n-1}\ \det(A_{1n}) & n > 1 \end{cases}$$

Consider the directed line segment $\langle p_1, p_2 \rangle$ from some point $p_1 = (x_1, y_1)$ to some other point $p_2 = (x_2, y_2)$. If $q = (x_3, y_3)$ is another point, we say q *is to the left (right) of* $\langle p_1, p_2 \rangle$ if the angle $p_1 p_2 q$ is a left (right) turn. [An angle is said to be a left (right) turn if it is less than or equal to (greater than or equal to) 180°.] We can check whether q is to the left (right) of $\langle p_1, p_2 \rangle$ by evaluating the determinant of the following matrix:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

If this determinant is positive (negative), then q is to the left (right) of $\langle p_1, p_2 \rangle$. If this determinant is zero, the three points are colinear. This test can be used, for example, to check whether a given point p is within a triangle formed by three points, say p_1, p_2, and p_3 (in clockwise order). The point p is within the triangle iff p is to the right of the line segments $\langle p_1, p_2 \rangle$, $\langle p_2, p_3 \rangle$, and $\langle p_3, p_1 \rangle$.

Also, for any three points $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3), the *signed area* formed by the corresponding triangle is given by one-half of the above determinant.

Let $p_1, p_2, \ldots, p_n$ be the vertices of the convex polygon Q in clockwise order. Let p be any other point. It is desired to check whether p lies in the interior of Q or outside. Consider a horizontal line h that extends from $-\infty$ to ∞ and goes through p. There are two possibilities: (1) h does not intersect any of the edges of Q, (2) h intersects some of the edges of Q. If case (1) is true, then, p is outside Q. In case (2), there can be at most two points of intersection. If h intersects Q at a single point, it is counted as two. Count the number of points of intersections that are to the left of p. If this number is even, then p is external to Q; otherwise it is internal to Q. This method of checking whether p is interior to Q takes $\Theta(n)$ time.

3.8.2 The QuickHull Algorithm

An algorithm that is similar to QuickSort can be devised to compute the convex hull of a set X of n points in the plane. This algorithm, called QuickHull, first identifies the two points (call them p_1 and p_2) of X with the smallest and largest x-coordinate values. Assume now that there are no ties. Later we see how to handle ties. Both p_1 and p_2 are extreme points and part of the convex hull. The set X is divided into X_1 and X_2 so that

X_1 has all the points to the left of the line segment $\langle p_1, p_2 \rangle$ and X_2 has all the points to the right of $\langle p_1, p_2 \rangle$. Both X_1 and X_2 include the two points p_1 and p_2. Then, the convex hulls of X_1 and X_2 (called the *upper hull* and *lower hull*, respectively) are computed using a divide-and-conquer algorithm called Hull. The union of these two convex hulls is the overall convex hull.

If there is more than one point with the smallest x-coordinate, let p_1' and p_1'' be the points from among these with the least and largest y-coordinates, respectively. Similarly define p_2' and p_2'' for the points with the largest x-coordinate values. Now X_1 will be all the points to the left of $\langle p_1'', p_2'' \rangle$ (including p_1'' and p_2'') and X_2 will be all the points to the right of $\langle p_1', p_2' \rangle$ (including p_1' and p_2'). In the rest of the discussion we assume for simplicity that there are no ties for p_1 and p_2. Appropriate modifications are needed in the event of ties.

We now describe how Hull computes the convex hull of X_1. We determine a point of X_1 that belongs to the convex hull of X_1 and use it to partition the problem into two independent subproblems. Such a point is obtained by computing the area formed by p_1, p, and p_2 for each p in X_1 and picking the one with the largest (absolute) area. Ties are broken by picking the point p for which the angle pp_1p_2 is maximum. Let p_3 be that point.

Now X_1 is divided into two parts; the first part contains all the points of X_1 that are to the left of $\langle p_1, p_3 \rangle$ (including p_1 and p_3), and the second part contains all the points of X_1 that are to the left of $\langle p_3, p_2 \rangle$ (including p_3 and p_2) (see Figure 3.7). There cannot be any point of X_1 that is to the left of both $\langle p_1, p_3 \rangle$ and $\langle p_3, p_2 \rangle$. Also, all the other points are interior points and can be dropped from future consideration. The convex hull of each part is computed recursively, and the two convex hulls are merged easily by placing one next to the other in the right order.

If there are m points in X_1, we can identify the point of division p_3 in time $O(m)$. Partitioning X_1 into two parts can also be done in $O(m)$ time. Merging the two convex hulls can be done in time $O(1)$. Let $T(m)$ stand for the run time of Hull on a list of m points and let m_1 and m_2 denote the sizes of the two resultant parts. Note that $m_1 + m_2 \leq m$. The recurrence relation for $T(m)$ is $T(m) = T(m_1) + T(m_2) + O(m)$, which is similar to the one for the run time of QuickSort. The worst-case run time is thus $O(m^2)$ on an input of m points. This happens when the partitioning at each level of recursion is highly uneven.

If the partitioning is nearly even at each level of recursion, then the run time will equal $O(m \log m)$ as in the case of QuickSort. Thus the average run time of QuickHull is $O(n \log n)$, on an input of size n, under appropriate assumptions on the input distribution.

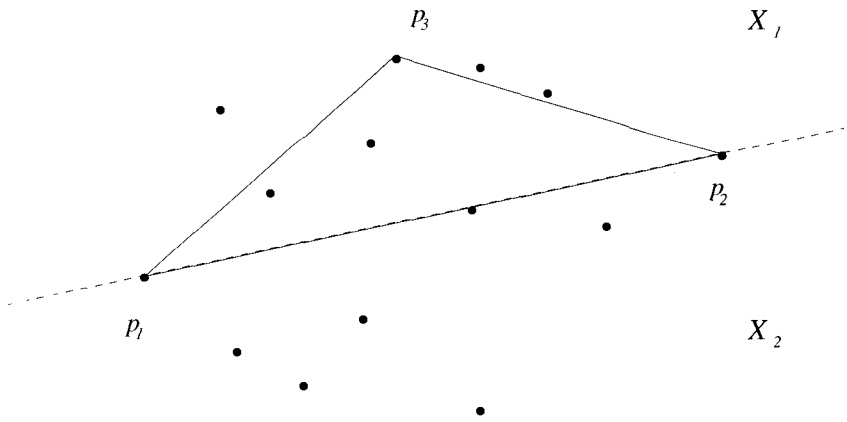


Figure 3.7 Identifying a point on the convex hull of X_1

3.8.3 Graham's Scan

If S is a set of points in the plane, Graham's scan algorithm identifies the point p from S with the lowest y-coordinate value (ties are broken by picking the leftmost among these). It then sorts the points of S according to the angle subtended by the points and p with the positive x-axis. Figure 3.8 gives an example. After having sorted the points, if we scan through the sorted list starting at p, every three successive points will form a left turn if all of these points lie on the hull. On the other hand if there are three successive points, say p_1, p_2, and p_3, that form a right turn, then we can immediately eliminate p_2 since it cannot lie on the convex hull. Notice that it will be an internal point because it lies within the triangle formed by p, p_1, and p_3.

We can eliminate all the interior points using the above procedure. Starting from p, we consider three successive points p_1, p_2, and p_3 at a time. To begin with, $p_1 = p$. If these points form a left turn, we move to the next point in the list (that is, we set $p_1 = p_2$, and so on). If these three points form a right turn, then p_2 is deleted since it is an interior point. We move one point behind in the list by setting p_1 equal to its predecessor. This process of scanning ends when we reach the point p again.

Example 3.11 In Figure 3.8, the first three points looked at are $p, 1$, and 2. Since these form a left turn, we move to $1, 2$, and 3. These form a right turn and hence 2 is deleted. Next, the three points $p, 1$, and 3 are considered. These form a left turn and hence the pointer is moved to point 1. The points

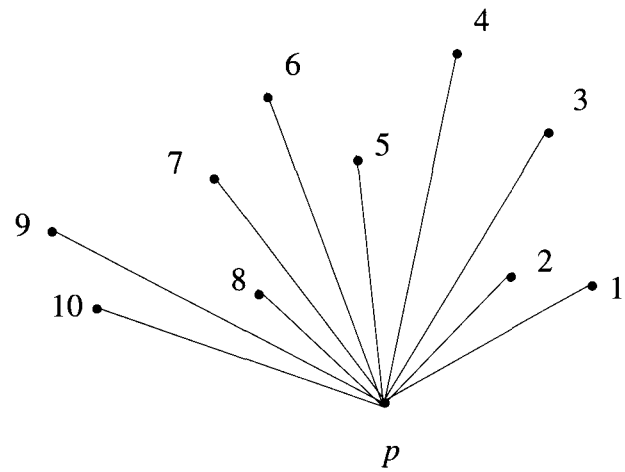


Figure 3.8 Graham's scan algorithm sorts the points first

1, 3, and 4 also form a left turn, and the scan proceeds to 3, 4, and 5 and then to 4, 5, and 6. Now point 5 gets deleted. The triplets 3, 4, 6; 4, 6, 7; and 6, 7, 8 form left turns whereas the next triplet 7, 8, 9 forms a right turn. Therefore, 8 gets deleted and in the next round 7 also gets eliminated. The next three triplets examined are 4, 6, 9; 6, 9, 10; and 9, 10, p, all of which are left turns. The final hull obtained is $p, 1, 3, 4, 6, 9$, and 10, which are points on the hull in counterclockwise (ccw) order. □

This scan process is given in Algorithm 3.21. In this algorithm the set of points is realized as a doubly linked list *ptslist*. Function Scan runs in $O(n)$ time since for each triplet examined, either the scan moves one node ahead or one point gets removed. In the latter case, the scan moves one node back. Also note that for each triplet, the test as to whether a left or right turn is formed can be done in $O(1)$ time. Function Area computes the signed area formed by three points. The major work in the algorithm is in sorting the points. Since sorting takes $O(n \log n)$ time, the total time of Graham's scan algorithm is $O(n \log n)$.

3.8.4 An $O(n \log n)$ Divide-and-Conquer Algorithm

In this section we present a simple divide-and-conquer algorithm, called DCHull, which also takes $O(n \log n)$ time and computes the convex hull in clockwise order.

```
point = record{
    float x; float y;
    point *prev; point *next;
};

1   Algorithm Scan(list)
2   // list is a pointer to the first node in the input list.
3   {
4       *p := list; *p1 := list;
5       repeat
6       {
7           p2 := (p1 → next);
8           if ((p2 → next) ≠ 0) then p3 := (p2 → next);
9           else return; // End of the list
10          temp := Area((p1 → x), (p1 → y), (p2 → x),
11              (p2 → y), (p3 → x), (p3 → y));
12          if (temp ≥ 0.0) then p1 := (p1 → next);
13          // If p1, p2, p3 form a left turn, move one point ahead;
14          // If not, delete p2 and move back.
15          else
16          {
17              (p1 → next) := p3; (p3 → prev) := p1; delete p2;
18              p1 := (p1 → prev);
19          }
20      } until (false);
21  }

1   Algorithm ConvexHull(ptslist)
2   {
3       // ptslist is a pointer to the first item of the input list. Find
4       // the point p in ptslist of lowest y-coordinate. Sort the
5       // points according to the angle made with p and the x-axis.

6       Sort(ptslist); Scan(ptslist); PrintList(ptslist);
7   }
```

Algorithm 3.21 Graham's scan algorithm

Given a set X of n points, like that in the case of QuickHull, the problem is reduced to finding the upper hull and the lower hull separately and then putting them together. Since the computations of the upper and lower hulls are very similar, we restrict our discussion to computing the upper hull. The divide-and-conquer algorithm for computing the upper hull partitions X into two nearly equal halves. Partitioning is done according to the x-coordinate values of points using the median x-coordinate as the splitter (see Section 3.6 for a discussion on median finding). Upper hulls are recursively computed for the two halves. These two hulls are then merged by finding the *line of tangent* (i.e., a straight line connecting a point each from the two halves, such that all the points of X are on one side of the line) (see Figure 3.9).

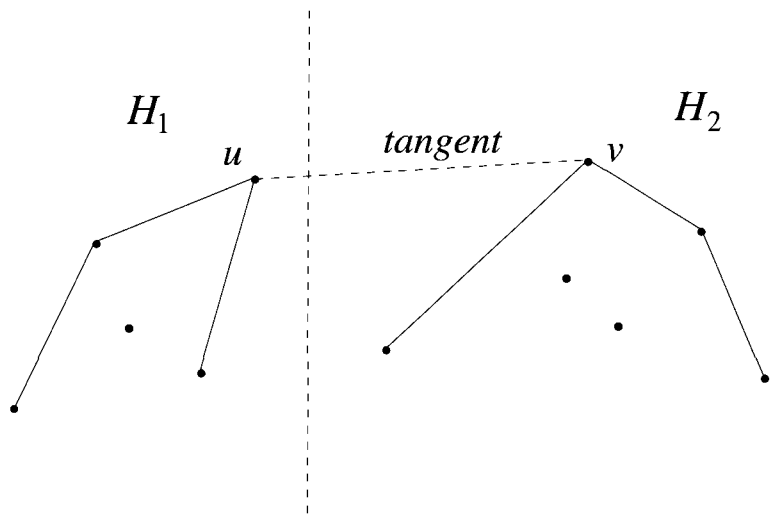


Figure 3.9 Divide and conquer to compute the convex hull

To begin with, the points p_1 and p_2 are identified [where p_1 (p_2) is the point with the least (largest) x-coordinate value]. This can be done in $O(n)$ time. Ties can be handled in exactly the same manner as in QuickHull. So, assume that there are no ties. All the points that are to the left of the line segment $\langle p_1, p_2 \rangle$ are separated from those that are to the right. This separation also can be done in $O(n)$ time. From here on, by "input" and "X" we mean all the points that are to the left of the line segment $\langle p_1, p_2 \rangle$. Also let $|X| = N$.

Sort the input points according to their x-coordinate values. Sorting can be done in $O(N \log N)$ time. This sorting is done only once in the computation of the upper hull. Let $q_1, q_2, \ldots, q_N$ be the sorted order of these

points. Now partition the input into two equal halves with $q_1, q_2, \ldots, q_{N/2}$ in the first half and $q_{N/2+1}, q_{N/2+2}, \ldots, q_N$ in the second half. The upper hull of each half is computed recursively. Let H_1 and H_2 be the upper hulls. Upper hulls are maintained as linked lists in clockwise order. We refer to the first element in the list as the leftmost point and the last element as the rightmost point.

The line of tangent is then found in $O(\log^2 N)$ time. If (u, v) is the line of tangent, then all the points of H_1 that are to the right of u are dropped. Similarly, all the points that are to the left of v in H_2 are dropped. The remaining part of H_1, the line of tangent, and the remaining part of H_2 form the upper hull of the given input set.

If $T(N)$ is the run time of the above recursive algorithm for the upper hull on an input of N points, then we have

$$T(N) = 2T(N/2) + O(\log^2 N)$$

which solves to $T(N) = O(N)$. Thus the run time is dominated by the initial sorting step.

The only part of the algorithm that remains to be specified is how to find the line of tangent $\langle u, v\rangle$ in $O(\log^2 N)$ time. The way to find the tangent is to start from the middle point, call it p, of H_1. Here the middle point refers to the middle element of the corresponding list. Find the tangent of p with H_2. Let $\langle p, q\rangle$ be the tangent. Using $\langle p, q\rangle$, we can determine whether u is to the left of, equal to, or to the right of p in H_1. A binary search in this fashion on the points of H_1 reveals u. Use a similar procedure to isolate v.

Lemma 3.1 Let H_1 and H_2 be two upper hulls with at most m points each. If p is any point of H_1, its point q of tangency with H_2 can be found in $O(\log m)$ time.

Proof. If q' is any point in H_2, we can check whether q' is to the left of, equal to, or to the right of q in $O(1)$ time (see Figure 3.10). In Figure 3.10, x and y are the left and right neighbors of q' in H_2, respectively. If $\angle pq'x$ is a right turn and $\angle pq'y$ is a left turn, then q is to the right of q' (see case 1 of Figure 3.10). If $\angle pq'x$ and $\angle pq'y$ are both right turns, then $q' = q$ (see case 2 of Figure 3.10); otherwise q is to the left of q' (see case 3 of Figure 3.10). Thus we can perform a binary search on the points of H_2 and identify q in $O(\log m)$ time. □

Lemma 3.2 If H_1 and H_2 are two upper hulls with at most m points each, their common tangent can be computed in $O(\log^2 m)$ time.

Proof. Let $u \in H_1$ and $v \in H_2$ be such that $\langle u, v\rangle$ is the line of tangent. Also let p be an arbitrary point of H_1 and let $q \in H_2$ be such that $\langle p, q\rangle$ is a

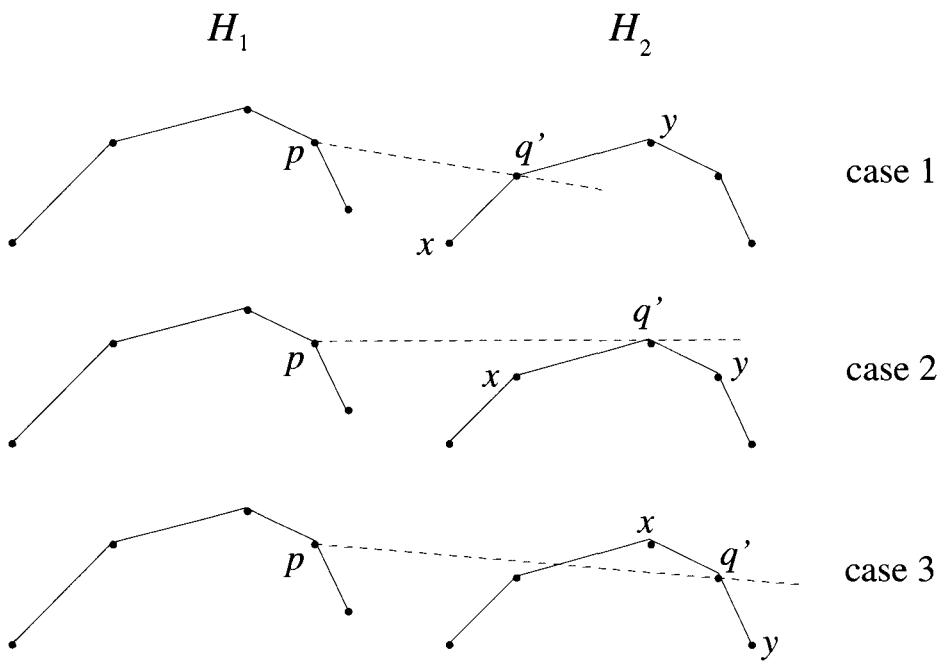


Figure 3.10 Proof of Lemma 3.1

tangent of H_2. Given p and q, we can check in $O(1)$ time whether u is to the left of, equal to, or to the right of p (see Figure 3.11). Here x and y are left and right neighbors, respectively, of p in H_1. If $\langle p, q \rangle$ is also tangential to H_1, then $p = u$. If $\angle xpq$ is a left turn, then u is to the left of p; otherwise u is to the right of p. This suggests a binary search for u. For each point p of H_1 chosen, we have to determine the tangent from p to H_2 and then decide the relative positioning of p with respect to u. We can do this computation in $O(\log m \times \log m) = O(\log^2 m)$ time. □

In summary, given two upper hulls with $\frac{N}{2}$ points each, the line of tangent can be computed in $O(\log^2 N)$ time.

Theorem 3.4 A convex hull of n points in the plane can be computed in $O(n \log n)$ time. □

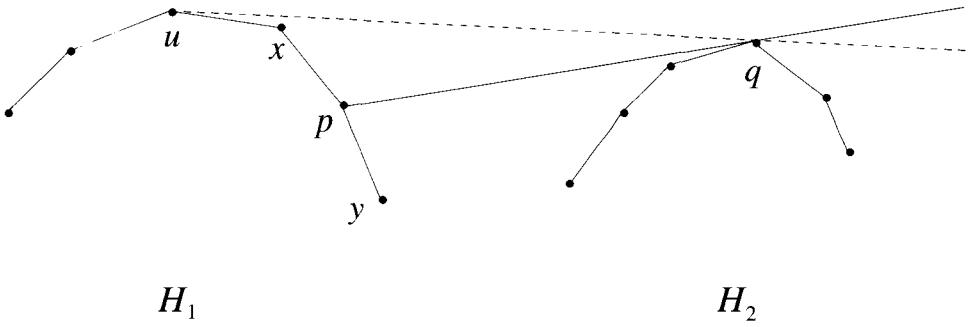


Figure 3.11 Proof of Lemma 3.2

EXERCISES

1. Write an algorithm in pseudocode that implements QuickHull and test it using suitable data.

2. Code the divide-and-conquer algorithm DCHull and test it using appropriate data.

3. Run the three algorithms for convex hull discussed in this section on various random inputs and compare their performances.

4. Algorithm DCHull can be modified as follows: Instead of using the median as the splitter, we could use a randomly chosen point as the splitter. Then X is partitioned into two around this point. The rest of the function DCHull is the same. Write code for this modified algorithm and compare it with DCHull empirically.

5. Let S be a set of n points in the plane. It is given that there is only a constant (say c) number of points on the hull of S. Can you devise a convex hull algorithm for S that runs in time $o(n \log n)$? Conceive of special algorithms for $c = 3$ and $c = 4$ first and then generalize.

3.9 REFERENCES AND READINGS

Algorithm MaxMin (Algorithm 3.6) is due to I. Pohl and the quicksort algorithm (Algorithm 3.13) is due to C. A. R. Haore. The randomized sorting algorithm in Algorithm 3.16 is due to W. D. Frazer and A. C. McKeller and

the selection algorithm of Algorithm 3.19 is due to M. Blum, R. Floyd, V. Pratt, R. Rivest and R. E. Tarjan.

For more on randomized sorting and selection see:

"Expected time bounds for selection," by R. Floyd and R. Rivest, *Communications of the ACM* 18, no. 3 (1975): 165–172.

"Samplesort: A Sampling Approach to Minimal Storage Tree Sorting," by W. D. Frazer and A. C. McKellar, *Journal of the ACM* 17, no. 3 (1970): 496–507.

"Derivation of Randomized Sorting and Selection Algorithms," by S. Rajasekaran and J. H. Reif, in *Parallel Algorithm Derivation and Program Transformation*, edited by R. Paige, J. H. Reif, and R. Wachter, Kluwer Academic Publishers, 1993, pp. 187–205.

The matrix multiplication algorithm in Section 3.7 is due to V. Strassen. For more information on the matrix multiplication problem see "Matrix multiplication via arithmetic progressions," by D. Coppersmith and S. Winograd, *Journal of Symbolic Computation* 9 (1990): 251–280. A complex $O(n^{2.376})$ time algorithm for multiplying two $n \times n$ matrices is given in this paper.

For more applications of divide-and-conquer see:

Computational Geometry, by F. Preparata and M. I. Shamos, Springer-Verlag, 1985.

Computational Geometry: An Introduction Through Randomized Algorithms by K. Mulmuley, Prentice-Hall, 1994.

Introduction to Algorithms: A Creative Approach, by U. Manber, Addison-Wesley, 1989.

3.10 ADDITIONAL EXERCISES

1. What happens to the worst-case run time of quicksort if we use the median of the given keys as the splitter key? (Assume that the selection algorithm of Section 3.6 is employed to determine the median).

2. The sets A and B have n elements each given in the form of sorted arrays. Present an $O(n)$ time algorithm to compute $A \cup B$ and $A \cap B$.

3. The sets A and B have m and n elements (respectively) from a linear order. These sets are not necessarily sorted. Also assume that $m \leq n$. Show how to compute $A \cup B$ and $A \cap B$ in $O(n \log m)$ time.

4. Consider the problem of sorting a sequence X of n keys where each key is either zero or one (i.e., each key is a bit). One way of sorting

X is to start with two empty lists L_0 and L_1. Let $X = k_1, k_2, \ldots, k_n$. For each $1 \leq i \leq n$ do: If $k_i = 0$, then append k_i to L_0. If $k_i = 1$, then append k_i to L_1. After processing all the keys of X in this manner, output the list L_0 followed by the list L_1.

The above idea of sorting can be extended to the case in which each key is of length more than one bit. In particular, if the keys are integers in the range $[0, m-1]$, then we start with m empty lists, $L_0, L_1, \ldots, L_{m-1}$, one list (or *bucket*) for each possible value that a key can take. Then the keys are processed in a similar fashion. In particular, if a key has a value ℓ, then it will be appended to the ℓth list.

Write an algorithm that employs this idea to sort n keys assuming that each key is in the range $[0, m-1]$. Show that the run time of your algorithm is $O(n+m)$. This algorithm is known as the *bucket sort.*

5. Consider the problem of sorting n two-digit integers. The idea of *radix sort* can be employed. We first sort the numbers only with respect to their least significant digits (LSDs). Followed by this, we apply a sort with respect to their second LSDs. More generally, d-digit numbers can be sorted in d phases, where in the ith phase $(1 \leq i \leq d)$ we sort the keys only with respect to their ith LSDs. Will this algorithm always work?

   As an example, let the input be $k_1 = 12, k_2 = 45, k_3 = 23, k_4 = 14, k_5 = 32$, and $k_6 = 57$. After sorting these keys with respect to their LSDs, we end up with: $k_5 = 32, k_1 = 12, k_3 = 23, k_4 = 14, k_2 = 45$, and $k_6 = 57$. When we sort the resultant sequence with respect to the keys' second LSDs (i.e., the next-most significant digits), we get $k_1 = 12, k_4 = 14, k_3 = 23, k_5 = 32, k_2 = 45$, and $k_6 = 57$, which is the correct answer!

   But note that in the second phase of the algorithm, $k_4 = 14, k_1 = 12, k_3 = 23, k_5 = 32, k_2 = 45, k_6 = 57$ is also a valid sort with respect to the second LSDs. The result in any phase of radix sorting can be forced to be correct by enforcing the following condition on the sorting algorithm to be used. "Keys with equal values should remain in the same relative order in the output as they were in the input." Any sorting algorithm that satisfies this is called a *stable sort.*

   Note that in the above example, if the algorithm used to sort the keys in the second phase is stable, then the output will be correct. In summary, radix sort can be employed to sort d-digit numbers in d phases such that the sort applied in each phase (except the first phase) is stable.

   More generally, radix sort can be used to sort integers of arbitrary length. As usual, the algorithm will consist of phases in each of which the keys are sorted only with respect to certain parts of their keys.

The parts used in each phase could be single bits, single digits, or more generally, ℓ bits, for some appropriate ℓ.

In Exercise 4, you showed that n integers in the range $[0, m-1]$ can be sorted in $O(n+m)$ time. Is your algorithm stable? If not, make it stable. As a special case, your algorithm can sort n integers in the range $[0, n-1]$ in $O(n)$ time. Use this algorithm together with the idea of radix sorting to develop an algorithm that can sort n integers in the range $[0, n^c - 1]$ (for any fixed c) in $O(n)$ time.

6. Two sets A and B have n elements each. Assume that each element is an integer in the range $[0, n^{100}]$. These sets are not necessarily sorted. Show how to check whether these two sets are disjoint in $O(n)$ time. Your algorithm should use $O(n)$ space.

7. Input are the sets $S_1, S_2, \ldots,$ and S_ℓ (where $\ell \leq n$). Elements of these sets are integers in the range $[0, n^c - 1]$ (for some fixed c). Also let $\sum_{i=1}^{\ell} |S_i| = n$. The goal is to output S_1 in sorted order, then S_2 in sorted order, and so on. Present an $O(n)$ time algorithm for this problem.

8. Input is an array of n numbers where each number is an integer in the range $[0, N]$ (for some $N >> n$). Present an algorithm that runs in the worst case in time $O\left(n \frac{\log N}{\log n}\right)$ and checks whether all these n numbers are distinct. Your algorithm should use only $O(n)$ space.

9. Let S be a sequence of n^2 integers in the range $[1, n]$. Let $R(i)$ be the number of i's in the sequence (for $i = 1, 2, \ldots, n$). Given S, we have to compute an approximate value of $R(i)$ for each i. If $N(i)$ is an approximation to $R(i), i = 1, \ldots, n$, it should be the case that (with high probability) $N(i) \geq R(i)$ for each i and $\sum_{i=1}^{n} N(i) = O(n^2)$. Of course we can do this computation in deterministic $O(n^2)$ time. Design a randomized algorithm for this problem that runs in time $O(n \log^{O(1)} n)$.

Chapter 4

THE GREEDY METHOD

4.1 THE GENERAL METHOD

The greedy method is perhaps the most straightforward design technique we consider in this text, and what's more it can be applied to a wide variety of problems. Most, though not all, of these problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a *feasible* solution. We need to find a feasible solution that either maximizes or minimizes a given *objective function.* A feasible solution that does this is called an *optimal solution.* There is usually an obvious way to determine a feasible solution but not necessarily an optimal solution.

The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added. The selection procedure itself is based on some optimization measure. This measure may be the objective function. In fact, several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the *subset paradigm.*

We can describe the subset paradigm abstractly, but more precisely than above, by considering the control abstraction in Algorithm 4.1.

The function Select selects an input from $a[\,]$ and removes it. The selected input's value is assigned to x. Feasible is a Boolean-valued function that determines whether x can be included into the solution vector. The function Union combines x with the solution and updates the objective function. The

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

For problems that do not call for the selection of an optimal subset, in the greedy method we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made. Call this version of the greedy method the *ordering paradigm.* Sections 4.2, 4.3, 4.4, and 4.5 consider problems that fit the subset paradigm, and Sections 4.6, 4.7, and 4.8 consider problems that fit the ordering paradigm.

EXERCISE

1. Write a control abstraction for the ordering paradigm.

4.2 KNAPSACK PROBLEM

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \tag{4.2}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

Lemma 4.1 In case the sum of all the weights is $\le m$, then $x_i = 1$, $1 \le i \le n$ is an optimal solution. □

So let us assume the sum of weights exceeds m. Now all the x_i's cannot be 1. Another observation to make is:

Lemma 4.2 All optimal solutions will fill the knapsack exactly. □

Lemma 4.2 is true because we can always increase the contribution of some object i by a fractional amount until the total weight is exactly m.

Note that the knapsack problem calls for selecting a subset of the objects and hence fits the subset paradigm. In addition to selecting a subset, the knapsack problem also involves the selection of an x_i for each object. Several simple greedy strategies to obtain feasible solutions whose sums are identically m suggest themselves. First, we can try to fill the knapsack by including next the object with largest profit. If an object under consideration doesn't fit, then a fraction of it is included to fill the knapsack. Thus each time an object is included (except possibly when the last object is included)

into the knapsack, we obtain the largest possible increase in profit value. Note that if only a fraction of the last object is included, then it may be possible to get a bigger increase by using a different object. For example, if we have two units of space left and two objects with ($p_i = 4$, $w_i = 4$) and ($p_j = 3, w_j = 2$) remaining, then using j is better than using half of i. Let us use this selection strategy on the data of Example 4.1.

Object one has the largest profit value ($p_1 = 25$). So it is placed into the knapsack first. Then $x_1 = 1$ and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Object two has the next largest profit ($p_2 = 24$). However, $w_2 = 15$ and it doesn't fit into the knapsack. Using $x_2 = 2/15$ fills the knapsack exactly with part of object 2 and the value of the resulting solution is 28.2. This is solution 2 and it is readily seen to be suboptimal. The method used to obtain this solution is termed a greedy method because at each step (except possibly the last one) we chose to introduce that object which would increase the objective function value the most. However, this greedy method did not yield an optimal solution. Note that even if we change the above strategy so that in the last step the objective function increases by as much as possible, an optimal solution is not obtained for Example 4.1.

We can formulate at least two other greedy approaches attempting to obtain optimal solutions. From the preceding example, we note that considering objects in order of nonincreasing profit values does not yield an optimal solution because even though the objective function value takes on large increases at each step, the number of steps is few as the knapsack capacity is used up at a rapid rate. So, let us try to be greedy with capacity and use it up as slowly as possible. This requires us to consider the objects in order of nondecreasing weights w_i. Using Example 4.1, solution 3 results. This too is suboptimal. This time, even though capacity is used slowly, profits aren't coming in rapidly enough.

Thus, our next attempt is an algorithm that strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit of capacity used. This means that objects are considered in order of the ratio p_i/w_i. Solution 4 of Example 4.1 is produced by this strategy. If the objects have already been sorted into nonincreasing order of p_i/w_i, then function GreedyKnapsack (Algorithm 4.2) obtains solutions corresponding to this strategy. Note that solutions corresponding to the first two strategies can be obtained using this algorithm if the objects are initially in the appropriate order. Disregarding the time to initially sort the objects, each of the three strategies outlined above requires only $O(n)$ time.

We have seen that when one applies the greedy method to the solution of the knapsack problem, there are at least three different measures one can attempt to optimize when determining which object to include next. These measures are total profit, capacity used, and the ratio of accumulated profit to capacity used. Once an optimization measure has been chosen, the greedy

2. [0/1 Knapsack] Consider the knapsack problem discussed in this section. We add the requirement that $x_i = 1$ or $x_i = 0$, $1 \leq i \leq n$; that is, an object is either included or not included into the knapsack. We wish to solve the problem

$$\max \sum_{1}^{n} p_i x_i$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \leq m$$

$$\text{and} \quad x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

One greedy strategy is to consider the objects in order of nonincreasing density p_i/w_i and add the object into the knapsack if it fits. Show that this strategy doesn't necessarily yield an optimal solution.

4.3 TREE VERTEX SPLITTING

Consider a directed binary tree each edge of which is labeled with a real number (called its *weight*). Trees with edge weights are called *weighted trees*. A weighted tree can be used, for example, to model a distribution network in which electric signals or commodities such as oil are transmitted. Nodes in the tree correspond to receiving stations and edges correspond to transmission lines. It is conceivable that in the process of transmission some loss occurs (drop in voltage in the case of electric signals or drop in pressure in the case of oil). Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network may not be able to tolerate losses beyond a certain level. In places where the loss exceeds the tolerance level, boosters have to be placed. Given a network and a loss tolerance level, the *Tree Vertex Splitting Problem (TVSP)* is to determine an optimal placement of boosters. It is assumed that the boosters can only be placed in the nodes of the tree.

The TVSP can be specified more precisely as follows: Let $T = (V, E, w)$ be a weighted directed tree, where V is the vertex set, E is the edge set, and w is the weight function for the edges. In particular, $w(i, j)$ is the weight of the edge $\langle i, j \rangle \in E$. The weight $w(i, j)$ is undefined for any $\langle i, j \rangle \notin E$. A *source vertex* is a vertex with in-degree zero, and a *sink vertex* is a vertex with out-degree zero. For any path P in the tree, its *delay*, $d(P)$, is defined to be the sum of the weights on that path. The delay of the tree T, $d(T)$, is the maximum of all the path delays.

Let T/X be the forest that results when each vertex u in X is split into two nodes u^i and u^o such that all the edges $\langle u, j \rangle \in E$ ($\langle j, u \rangle \in E$) are

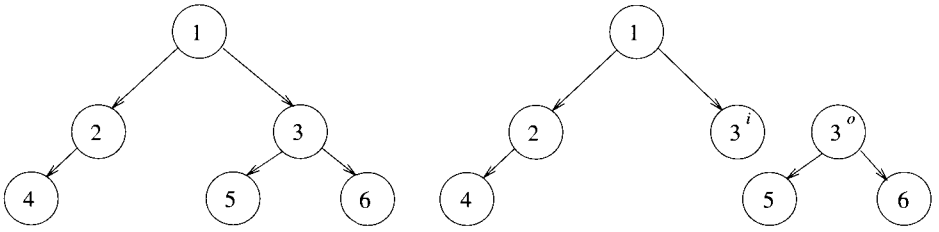


Figure 4.1 A tree before and after splitting the node 3

replaced by edges of the form $\langle u^o, j\rangle$ ($\langle j, u^i\rangle$). In other words, outbound edges from u now leave from u^o and inbound edges to u now enter at u^i. Figure 4.1 shows a tree before and after splitting the node 3. A node that gets split corresponds to a booster station. The TVSP is to identify a set $X \subseteq V$ of minimum cardinality for which $d(T/X) \leq \delta$, for some specified tolerance limit δ. Note that the TVSP has a solution only if the maximum edge weight is $\leq \delta$. Also note that the TVSP naturally fits the subset paradigm.

Given a weighted tree $T(V, E, w)$ and a tolerance limit δ, any subset X of V is a feasible solution if $d(T/X) \leq \delta$. Given an X, we can compute $d(T/X)$ in $O(|V|)$ time. A trivial way of solving the TVSP is to compute $d(T/X)$ for each possible subset X of V. But there are $2^{|V|}$ such subsets! A better algorithm can be obtained using the greedy method.

For the TVSP, the quantity that is optimized (minimized) is the number of nodes in X. A greedy approach to solving this problem is to compute for each node $u \in V$, the maximum delay $d(u)$ from u to any other node in its subtree. If u has a parent v such that $d(u) + w(v, u) > \delta$, then the node u gets split and $d(u)$ is set to zero. Computation proceeds from the leaves toward the root.

In the tree of Figure 4.2, let $\delta = 5$. For each of the leaf nodes $7, 8, 5, 9$, and 10 the delay is zero. The delay for any node is computed only after the delays for its children have been determined. Let u be any node and $C(u)$ be the set of all children of u. Then $d(u)$ is given by

$$d(u) = \max_{v \in C(u)} \{d(v) + w(u, v)\}$$

Using the above formula, for the tree of Figure 4.2, $d(4) = 4$. Since $d(4) + w(2, 4) = 6 > \delta$, node 4 gets split. We set $d(4) = 0$. Now $d(2)$ can be

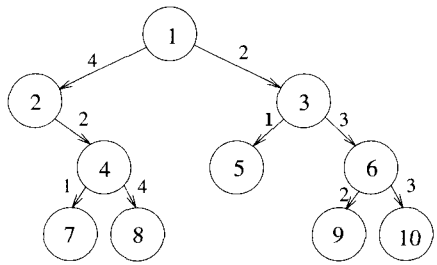


Figure 4.2 An example tree

computed and is equal to 2. Since $d(2) + w(1,2)$ exceeds δ, node 2 gets split and $d(2)$ is set to zero. Then $d(6)$ is equal to 3. Also, since $d(6)+w(3,6) > \delta$, node 6 has to be split. Set $d(6)$ to zero. Now $d(3)$ is computed as 3. Finally, $d(1)$ is computed as 5.

Figure 4.3 shows the final tree that results after splitting the nodes 2, 4, and 6. This algorithm is described in Algorithm 4.3, which is invoked as TVS($root, \delta$), *root* being the root of the tree. The order in which TVS visits (i.e., computes the delay values of) the nodes of the tree is called the *post order* and is studied again in Chapter 6.

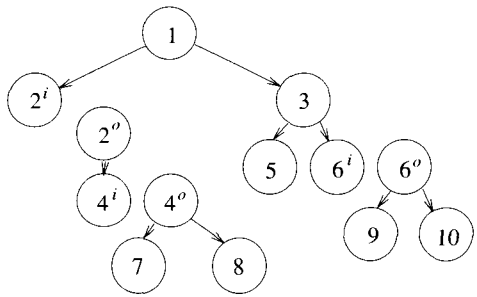


Figure 4.3 The final tree after splitting the nodes 2, 4, and 6

```
1   Algorithm TVS(T, δ)
2   // Determine and output the nodes to be split.
3   // w() is the weighting function for the edges.
4   {
5       if (T ≠ 0) then
6       {
7           d[T] := 0;
8           for each child v of T do
9           {
10              TVS(v, δ);
11              d[T] := max{d[T], d[v] + w(T, v)};
12          }
13          if ((T is not the root)  and
14                  (d[T] + w(parent(T), T) > δ)) then
15          {
16              write (T); d[T] := 0;
17          }
18      }
19  }
```

Algorithm 4.3 The tree vertex splitting algorithm

Algorithm TVS takes $\Theta(n)$ time, where n is the number of nodes in the tree. This can be seen as follows: When TVS is called on any node T, only a constant number of operations are performed (excluding the time taken for the recursive calls). Also, TVS is called only once on each node T in the tree.

Algorithm 4.4 is a revised version of Algorithm 4.3 for the special case of directed binary trees. A sequential representation of the tree (see Section 2.2) has been employed. The tree is stored in the array *tree*[ ] with the root at *tree*[1]. Edge weights are stored in the array *weight*[ ]. If *tree*[*i*] has a tree node, the weight of the incoming edge from its parent is stored in *weight*[*i*]. The delay of node i is stored in $d[i]$. The array $d[\ ]$ is initialized to zero at the beginning. Entries in the arrays *tree*[ ] and *weight*[ ] corresponding to nonexistent nodes will be zero. As an example, for the tree of Figure 4.2, *tree*[ ] will be set to $\{1, 2, 3, 0, 4, 5, 6, 0, 0, 7, 8, 0, 0, 9, 10\}$ starting at cell 1. Also, *weight*[ ] will be set to $\{0, 4, 2, 0, 2, 1, 3, 0, 0, 1, 4, 0, 0, 2, 3\}$ at the beginning, starting from cell 1. The algorithm is invoked as TVS$(1, \delta)$. Now we show that TVS (Algorithm 4.3) will always split a minimal number of nodes.

```
1   Algorithm TVS(i, δ)
2   // Determine and output a minimum cardinality split set.
3   // The tree is realized using the sequential representation.
4   // Root is at tree[1]. N is the largest number such that
5   // tree[N] has a tree node.
6   {
7       if (tree[i] ≠ 0) then // If the tree is not empty
8           if (2i > N) then d[i] := 0; // i is a leaf.
9           else
10          {
11              TVS(2i, δ);
12              d[i] := max(d[i], d[2i] + weight[2i]);
13              if (2i + 1 ≤ N) then
14              {
15                  TVS(2i + 1, δ);
16                  d[i] := max(d[i], d[2i + 1] + weight[2i + 1]);
17              }
18          }
19          if ((tree[i] ≠ 1) and (d[i] + weight[i] > δ)) then
20          {
21              write (tree[i]); d[i] := 0;
22          }
23  }
```

Algorithm 4.4 TVS for the special case of binary trees

Theorem 4.2 Algorithm TVS outputs a minimum cardinality set U such that $d(T/U) \leq \delta$ on any tree T, provided no edge of T has weight $> \delta$.

Proof: The proof is by induction on the number of nodes in the tree. If the tree has a single node, the theorem is true. Assume the theorem for all trees of size $\leq n$. We prove it for trees of size $n+1$ also.

Let T be any tree of size $n+1$ and let U be the set of nodes split by TVS. Also let W be a minimum cardinality set such that $d(T/W) \leq \delta$. We have to show that $|U| \leq |W|$. If $|U| = 0$, this is true. Otherwise, let x be the first vertex split by TVS. Let T_x be the subtree rooted at x. Let T' be the tree obtained from T by deleting T_x except for x. Note that W has to have at least one node, say y, from T_x. Let $W' = W - \{y\}$. If there is a W^* such that $|W^*| < |W'|$ and $d(T'/W^*) \leq \delta$, then since $d(T/(W^* + \{x\})) \leq \delta$, W is not a minimum cardinality split set for T. Thus, W' has to be a minimum cardinality split set such that $d(T'/W') \leq \delta$.

If algorithm TVS is run on tree T', the set of split nodes output is $U-\{x\}$. Since T' has $\leq n$ nodes, $U-\{x\}$ is a minimum cardinality split set for T'. This in turn means that $|W'| \geq |U| - 1$. In other words, $|W| \geq |U|$. □

EXERCISES

1. For the tree of Figure 4.2 solve the TVSP when (a) $\delta = 4$ and (b) $\delta = 6$.

2. Rewrite TVS (Algorithm 4.3) for general trees. Make use of pointers.

4.4 JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Example 4.2 Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

To formulate a greedy algorithm to obtain an optimal solution, we must formulate an optimization measure to determine how the next job is chosen. As a first attempt we can choose the objective function $\sum_{i \in J} p_i$ as our optimization measure. Using this measure, the next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution. This requires us to consider jobs in nonincreasing order of the p_i's. Let us apply this criterion to the data of Example 4.2. We begin with $J = \emptyset$ and $\sum_{i \in J} p_i = 0$. Job 1 is added to J as it has the largest profit and $J = \{1\}$ is a feasible solution. Next, job 4 is considered. The solution $J = \{1, 4\}$ is also feasible. Next, job 3 is considered and discarded as $J = \{1, 3, 4\}$ is not feasible. Finally, job 2 is considered for inclusion into J. It is discarded as $J = \{1, 2, 4\}$ is not feasible. Hence, we are left with the solution $J = \{1, 4\}$ with value 127. This is the optimal solution for the given problem instance. Theorem 4.4 proves that the greedy algorithm just described always obtains an optimal solution to this sequencing problem.

Before attempting the proof, let us see how we can determine whether a given J is a feasible solution. One obvious way is to try out all possible permutations of the jobs in J and check whether the jobs in J can be processed in any one of these permutations (sequences) without violating the deadlines. For a given permutation $\sigma = i_1, i_2, i_3, \ldots, i_k$, this is easy to do, since the earliest time job $i_q, 1 \leq q \leq k$, will be completed is q. If $q > d_{i_q}$, then using σ, at least job i_q will not be completed by its deadline. However, if $|J| = i$, this requires checking $i!$ permutations. Actually, the feasibility of a set J can be determined by checking only one permutation of the jobs in J. This permutation is any one of the permutations in which jobs are ordered in nondecreasing order of deadlines.

Theorem 4.3 Let J be a set of k jobs and $\sigma = i_1, i_2, \ldots, i_k$ a permutation of jobs in J such that $d_{i_1} \leq d_{i_2} \leq \cdots \leq d_{i_k}$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline.

Proof: Clearly, if the jobs in J can be processed in the order σ without violating any deadline, then J is a feasible solution. So, we have only to show that if J is feasible, then σ represents a possible order in which the jobs can be processed. If J is feasible, then there exists $\sigma' = r_1, r_2, \ldots, r_k$ such that $d_{r_q} \geq q$, $1 \leq q \leq k$. Assume $\sigma' \neq \sigma$. Then let a be the least index such that $r_a \neq i_a$. Let $r_b = i_a$. Clearly, $b > a$. In σ' we can interchange r_a and r_b. Since $d_{r_a} \geq d_{r_b}$, the resulting permutation $\sigma'' = s_1, s_2, \ldots, s_k$ represents an order in which the jobs can be processed without violating a deadline. Continuing in this way, σ' can be transformed into σ without violating any deadline. Hence, the theorem is proved. □

Theorem 4.3 is true even if the jobs have different processing times $t_i \geq 0$ (see the exercises).

Theorem 4.4 The greedy method described above always obtains an optimal solution to the job sequencing problem.

Proof: Let $(p_i, d_i), 1 \leq i \leq n$, define any instance of the job sequencing problem. Let I be the set of jobs selected by the greedy method. Let J be the set of jobs in an optimal solution. We now show that both I and J have the same profit values and so I is also optimal. We can assume $I \neq J$ as otherwise we have nothing to prove. Note that if $J \subset I$, then J cannot be optimal. Also, the case $I \subset J$ is ruled out by the greedy method. So, there exist jobs a and b such that $a \in I$, $a \notin J$, $b \in J$, and $b \notin I$. Let a be a highest-profit job such that $a \in I$ and $a \notin J$. It follows from the greedy method that $p_a \geq p_b$ for all jobs b that are in J but not in I. To see this, note that if $p_b > p_a$, then the greedy method would consider job b before job a and include it into I.

Now, consider feasible schedules S_I and S_J for I and J respectively. Let i be a job such that $i \in I$ and $i \in J$. Let i be scheduled from t to $t+1$ in S_I and t' to $t'+1$ in S_J. If $t < t'$, then we can interchange the job (if any) scheduled in $[t', t'+1]$ in S_I with i. If no job is scheduled in $[t', t'+1]$ in I, then i is moved to $[t', t'+1]$. The resulting schedule is also feasible. If $t' < t$, then a similar transformation can be made in S_J. In this way, we can obtain schedules S'_I and S'_J with the property that all jobs common to I and J are scheduled at the same time. Consider the interval $[t_a, t_a + 1]$ in S'_I in which the job a (defined above) is scheduled. Let b be the job (if any) scheduled in S'_J in this interval. From the choice of $a, p_a \geq p_b$. Scheduling a from t_a to $t_a + 1$ in S'_J and discarding job b gives us a feasible schedule for job set $J' = J - \{b\} \cup \{a\}$. Clearly, J' has a profit value no less than that of J and differs from I in one less job than J does.

By repeatedly using the transformation just described, J can be transformed into I with no decrease in profit value. So I must be optimal. □

A high-level description of the greedy algorithm just discussed appears as Algorithm 4.5. This algorithm constructs an optimal set J of jobs that can be processed by their due times. The selected jobs can be processed in the order given by Theorem 4.3.

Now, let us see how to represent the set J and how to carry out the test of lines 7 and 8 in Algorithm 4.5. Theorem 4.3 tells us how to determine whether all jobs in $J \cup \{i\}$ can be completed by their deadlines. We can avoid sorting the jobs in J each time by keeping the jobs in J ordered by deadlines. We can use an array $d[1:n]$ to store the deadlines of the jobs in the order of their p-values. The set J itself can be represented by a one-dimensional array $J[1:k]$ such that $J[r]$, $1 \leq r \leq k$ are the jobs in J and $d[J[1]] \leq d[J[2]] \leq \cdots \leq d[J[k]]$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r$, $1 \leq r \leq k+1$. The insertion of i into J is simplified by the use of a fictitious job 0 with $d[0] = 0$ and $J[0] = 0$. Note also that if job i is to be inserted at position q, then only the positions of jobs $J[q]$, $J[q+1]$,

```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

Algorithm 4.5 High-level description of job sequencing algorithm

$\ldots$, $J[k]$ are changed after the insertion. Hence, it is necessary to verify only that these jobs (and also job i) do not violate their deadlines following the insertion. The algorithm that results from this discussion is function JS (Algorithm 4.6). The algorithm assumes that the jobs are already sorted such that $p_1 \geq p_2 \geq \cdots \geq p_n$. Further it assumes that $n \geq 1$ and the deadline $d[i]$ of job i is at least 1. Note that no job with $d[i] < 1$ can ever be finished by its deadline. Theorem 4.5 proves that JS is a correct implementation of the greedy strategy.

Theorem 4.5 Function JS is a correct implementation of the greedy-based method described above.

Proof: Since $d[i] \geq 1$, the job with the largest p_i will always be in the greedy solution. As the jobs are in nonincreasing order of the p_i's, line 8 in Algorithm 4.6 includes the job with largest p_i. The **for** loop of line 10 considers the remaining jobs in the order required by the greedy method described earlier. At all times, the set of jobs already included in the solution is maintained in J. If $J[i]$, $1 \leq i \leq k$, is the set already included, then J is such that $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$. This allows for easy application of the feasibility test of Theorem 4.3. When job i is being considered, the **while** loop of line 15 determines where in J this job has to be inserted. The use of a fictitious job 0 (line 7) allows easy insertion into position 1. Let w be such that $d[J[w]] \leq d[i]$ and $d[J[q]] > d[i]$, $w < q \leq k$. If job i is included into J, then jobs $J[q]$, $w < q \leq k$, have to be moved one position up in J (line 19). From Theorem 4.3, it follows that such a move retains feasibility of J iff $d[J[q]] \neq q$, $w < q \leq k$. This condition is verified in line 15. In addition, i can be inserted at position $w+1$ iff $d[i] > w$. This is verified in line 16 (note $r = w$ on exit from the **while** loop if $d[J[q]] \neq q$, $w < q \leq k$). The correctness of JS follows from these observations. □

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

Algorithm 4.6 Greedy algorithm for sequencing unit time jobs with deadlines and profits

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k - r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n - 1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J.

Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

The computing time of JS can be reduced from $O(n^2)$ to nearly $O(n)$ by using the disjoint set union and find algorithms (see Section 2.5) and a different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J. The proof of the validity of this statement is left as an exercise.

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

Since there are only n jobs and each job takes one unit of time, it is necessary only to consider the time slots $[i - 1, i]$, $1 \leq i \leq b$, such that $b = \min\{n, \max\{d_i\}\}$. One way to implement the above scheduling rule is to partition the time slots $[i-1, i]$, $1 \leq i \leq b$, into sets. We use i to represent the time slots $[i - 1, i]$. For any slot i, let n_i be the largest integer such that $n_i \leq i$ and slot n_i is free. To avoid end conditions, we introduce a fictitious slot $[-1, 0]$ which is always free. Two slots i and j are in the same set iff $n_i = n_j$. Clearly, if i and j, $i < j$, are in the same set, then $i, i+1, i+2, \ldots, j$ are in the same set. Associated with each set k of slots is a value $f(k)$. Then $f(k) = n_i$ for all slots i in set k. Using the set representation of Section 2.5, each set is represented as a tree. The root node identifies the set. The function f is defined only for root nodes. Initially, all slots are free and we have $b + 1$ sets corresponding to the $b + 1$ slots $[i - 1, i]$, $0 \leq i \leq b$. At this time $f(i) = i$, $0 \leq i \leq b$. We use $p(i)$ to link slot i into its set tree. With the conventions for the union and find algorithms of Section 2.5, $p(i) = -1$, $0 \leq i \leq b$, initially. If a job with deadline d is to be scheduled, then we need to find the root of the tree containing the slot $\min\{n, d\}$. If this root is j,

then $f(j)$ is the nearest free slot, provided $f(j) \neq 0$. Having used this slot, the set with root j should be combined with the set containing slot $f(j)-1$.

Example 4.4 The trees defined by the $p(i)$'s for the first three iterations in Example 4.3 are shown in Figure 4.4. □

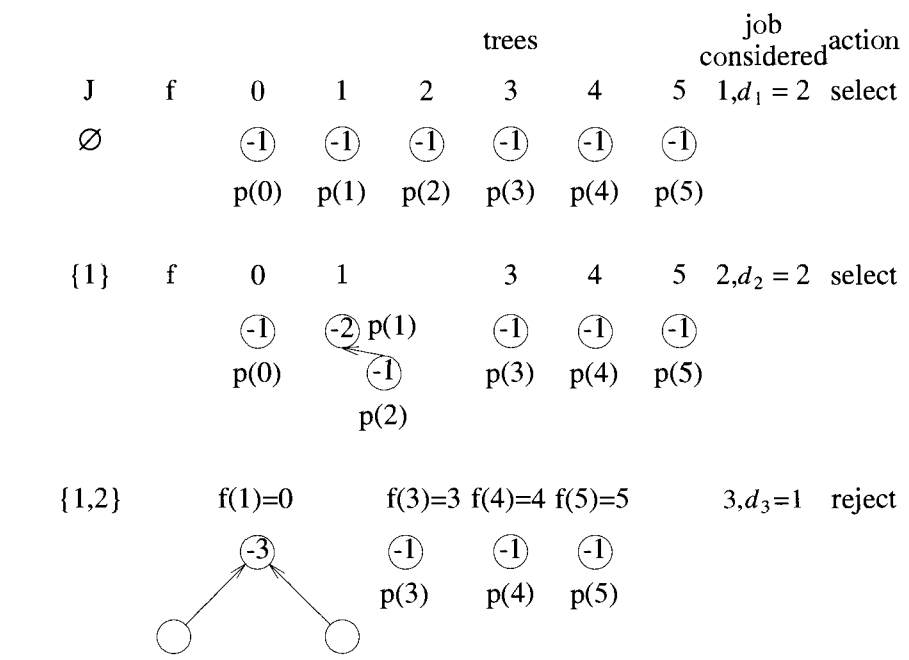


Figure 4.4 Fast job scheduling

The fast algorithm appears as FJS (Algorithm 4.7). Its computing time is readily observed to be $O(n\alpha(2n,n))$ (recall that $\alpha(2n,n)$ is the inverse of Ackermann's function defined in Section 2.5). It needs an additional $2n$ words of space for f and p.

```
1   Algorithm FJS(d, n, b, j)
2   // Find an optimal solution J[1 : k]. It is assumed that
3   // p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
4   {
5       // Initially there are b + 1 single node trees.
6       for i := 0 to b do f[i] := i;
7       k := 0; // Initialize.
8       for i := 1 to n do
9       { // Use greedy rule.
10          q := CollapsingFind(min(n, d[i]));
11          if (f[q] ≠ 0) then
12          {
13              k := k + 1; J[k] := i; // Select job i.
14              m := CollapsingFind(f[q] − 1);
15              WeightedUnion(m, q);
16              f[q] := f[m]; // q may be new root.
17          }
18      }
19  }
```

Algorithm 4.7 Faster algorithm for job sequencing

EXERCISES

1. You are given a set of n jobs. Associated with each job i is a processing time t_i and a deadline d_i by which it must be completed. A feasible schedule is a permutation of the jobs such that if the jobs are processed in that order, then each job finishes by its deadline. Define a greedy schedule to be one in which the jobs are processed in nondecreasing order of deadlines. Show that if there exists a feasible schedule, then all greedy schedules are feasible.

2. [Optimal assignment] Assume there are n workers and n jobs. Let v_{ij} be the value of assigning worker i to job j. An assignment of workers to jobs corresponds to the assignment of 0 or 1 to the variables x_{ij}, $1 \leq i, j \leq n$. Then $x_{ij} = 1$ means worker i is assigned to job j, and $x_{ij} = 0$ means that worker i is not assigned to job j. A valid assignment is one in which each worker is assigned to exactly one job and exactly one worker is assigned to any one job. The value of an assignment is $\sum_i \sum_j v_{ij} x_{ij}$.

For example, assume there are three workers w_1, w_2, and w_3 and three jobs j_1, j_2, and j_3. Let the values of assignment be $v_{11} = 11$, $v_{12} = 5$, $v_{13} = 8$, $v_{21} = 3$, $v_{22} = 7$, $v_{23} = 15$, $v_{31} = 8$, $v_{32} = 12$, and $v_{33} = 9$. Then, a valid assignment is $x_{12} = 1$, $x_{23} = 1$, and $x_{31} = 1$. The rest of the x_{ij}'s are zeros. The value of this assignment is $5 + 15 + 8 = 28$.

An optimal assignment is a valid assignment of maximum value. Write algorithms for two different greedy assignment schemes. One of these assigns a worker to the best possible job. The other assigns to a job the best possible worker. Show that neither of these schemes is guaranteed to yield optimal assignments. Is either scheme always better than the other? Assume $v_{ij} > 0$.

3. (a) What is the solution generated by the function JS when $n = 7$, $(p_1, p_2, \ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$, and $(d_1, d_2, \ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$?

   (b) Show that Theorem 4.3 is true even if jobs have different processing requirements. Associated with job i is a profit $p_i > 0$, a time requirement $t_i > 0$, and a deadline $d_i \geq t_i$.

   (c) Show that for the situation of part (a), the greedy method of this section doesn't necessarily yield an optimal solution.

4. (a) For the job sequencing problem of this section, show that the subset J represents a feasible solution iff the jobs in J can be processed according to the rule: if job i in J hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the least integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free.

   (b) For the problem instance of Exercise 3(a) draw the trees and give the values of $f(i), 0 \leq i \leq n$, after each iteration of the **for** loop of line 8 of Algorithm 4.7.

4.5 MINIMUM-COST SPANNING TREES

Definition 4.1 Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a *spanning tree* of G iff t is a tree. □

Example 4.5 Figure 4.5 shows the complete graph on four nodes together with three of its spanning trees. □

Spanning trees have many applications. For example, they can be used to obtain an independent set of circuit equations for an electric network. First, a spanning tree for the electric network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to

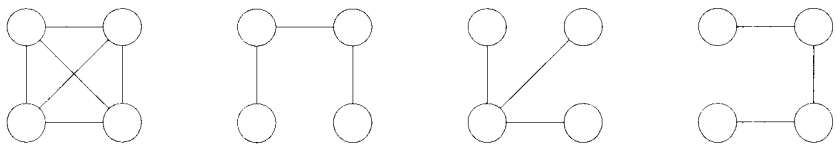


Figure 4.5 An undirected graph and three of its spanning trees

the spanning tree creates a cycle. Kirchoff's second law is used on each cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B that is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it can be shown that the cycles obtained by introducing the edges of B one at a time into the resulting spanning tree form a cycle basis, and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis.

Another application of spanning trees arises from the property that a spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected. (A minimal subgraph is one with the fewest number of edges.) Any connected graph with n vertices must have at least $n-1$ edges and all connected graphs with $n-1$ edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is $n-1$. The spanning trees of G represent all feasible choices.

In practical situations, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree (assuming all weights are positive). If this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle results in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. (The cost of a spanning tree is the sum of the costs of the edges in that tree.) Figure 4.6 shows a graph and one of its minimum-cost spanning trees. Since the identification of a minimum-cost spanning tree involves the selection of a subset of the edges, this problem fits the subset paradigm.

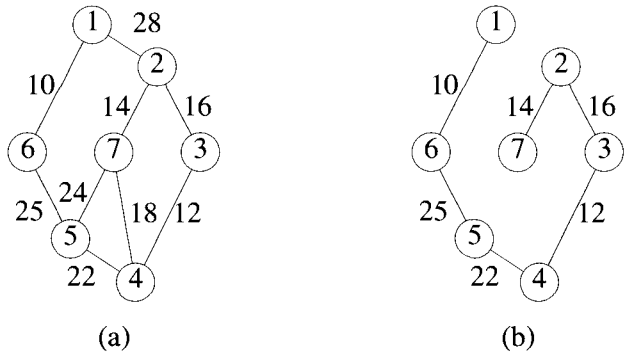


Figure 4.6 A graph and its minimum cost spanning tree

4.5.1 Prim's Algorithm

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $A \cup \{(u, v)\}$ is also a tree. Exercise 2 shows that this selection criterion results in a minimum-cost spanning tree. The corresponding algorithm is known as Prim's algorithm.

Example 4.6 Figure 4.7 shows the working of Prim's method on the graph of Figure 4.6(a). The spanning tree obtained is shown in Figure 4.6(b) and has a cost of 99. □

Having seen how Prim's method works, let us obtain a pseudocode algorithm to find a minimum-cost spanning tree using this method. The algorithm will start with a tree that includes only a minimum-cost edge of G. Then, edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j), $cost[i, j]$, is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value $near[j]$. The value $near[j]$ is a vertex in the tree such that $cost[j, near[j]]$ is minimum among all choices for $near[j]$. We define $near[j] = 0$ for all vertices j that are already in the tree. The next edge

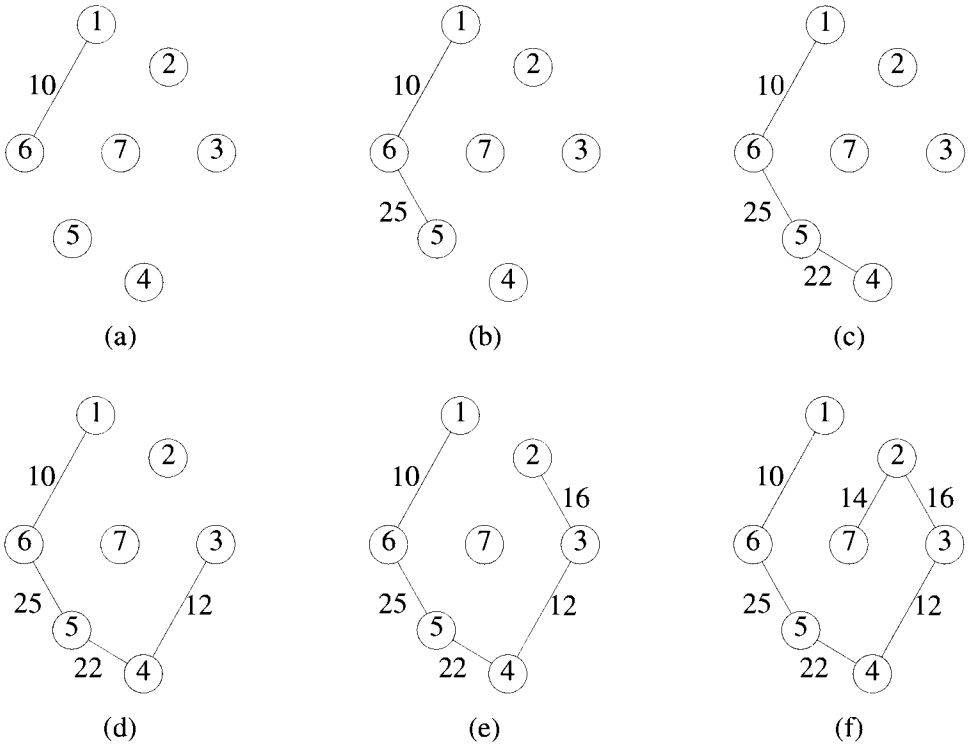


Figure 4.7 Stages in Prim's algorithm

to include is defined by the vertex j such that $near[j] \neq 0$ (j not already in the tree) and $cost[j, near[j]]$ is minimum.

In function Prim (Algorithm 4.8), line 9 selects a minimum-cost edge. Lines 10 to 15 initialize the variables so as to represent a tree comprising only the edge (k, l). In the **for** loop of line 16 the remainder of the spanning tree is built up edge by edge. Lines 18 and 19 select $(j, near[j])$ as the next edge to include. Lines 23 to 25 update $near[\,]$.

The time required by algorithm Prim is $O(n^2)$, where n is the number of vertices in the graph G. To see this, note that line 9 takes $O(|E|)$ time and line 10 takes $\Theta(1)$ time. The **for** loop of line 12 takes $\Theta(n)$ time. Lines 18 and 19 and the **for** loop of line 23 require $O(n)$ time. So, each iteration of the **for** loop of line 16 takes $O(n)$ time. The total time for the **for** loop of line 16 is therefore $O(n^2)$. Hence, Prim runs in $O(n^2)$ time.

If we store the nodes not yet included in the tree as a red-black tree (see Section 2.4.2), lines 18 and 19 take $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). The **for** loop of line 23 has to examine only the nodes adjacent to j. Thus its overall frequency is $O(|E|)$. Updating in lines 24 and 25 also takes $O(\log n)$ time (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

The algorithm can be speeded a bit by making the observation that a minimum-cost spanning tree includes for each vertex v a minimum-cost edge incident to v. To see this, suppose t is a minimum-cost spanning tree for $G = (V, E)$. Let v be any vertex in t. Let (v, w) be an edge with minimum cost among all edges incident to v. Assume that $(v, w) \notin E(t)$ and $cost[v, w] < cost[v, x]$ for all edges $(v, x) \in E(t)$. The inclusion of (v, w) into t creates a unique cycle. This cycle must include an edge (v, x), $x \neq w$. Removing (v, x) from $E(t) \cup \{(v, w)\}$ breaks this cycle without disconnecting the graph $(V, E(t) \cup \{(v, w)\})$. Hence, $(V, E(t) \cup \{(v, w)\} - \{(v, x)\})$ is also a spanning tree. Since $cost[v, w] < cost[v, x]$, this spanning tree has lower cost than t. This contradicts the assumption that t is a minimum-cost spanning tree of G. So, t includes minimum-cost edges as stated above.

From this observation it follows that we can start the algorithm with a tree consisting of any arbitrary vertex and no edge. Then edges can be added one by one. The changes needed are to lines 9 to 17. These lines can be replaced by the lines

```
9'        mincost := 0;
10'       for i := 2 to n do near[i] := 1;
11'           // Vertex 1 is initially in t.
12'       near[1] := 0;
13'-16'   for i := 1 to n - 1 do
17'       { // Find n - 1 edges for t.
```

4.5.2 Kruskal's Algorithm

There is a second possible interpretation of the optimization criteria mentioned earlier in which the edges of the graph are considered in nondecreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to *complete* t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t. We show in Theorem 4.6 that this interpretation of the greedy method also results in a minimum-cost spanning tree. This method is due to Kruskal.

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Algorithm 4.8 Prim's minimum-cost spanning tree algorithm

Example 4.7 Consider the graph of Figure 4.6(a). We begin with no edges selected. Figure 4.8(a) shows the current graph with no edges selected. Edge $(1,6)$ is the first edge considered. It is included in the spanning tree being built. This yields the graph of Figure 4.8(b). Next, the edge $(3,4)$ is selected and included in the tree (Figure 4.8(c)). The next edge to be considered is $(2,7)$. Its inclusion in the tree being built does not create a cycle, so we get the graph of Figure 4.8(d). Edge $(2,3)$ is considered next and included in the tree Figure 4.8(e). Of the edges not yet considered, $(7,4)$ has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded. Edge $(5,4)$ is the next edge to be added to the tree being built. This results in the configuration of Figure 4.8(f). The next edge to be considered is the edge $(7,5)$. It is discarded, as its inclusion creates a cycle. Finally, edge $(6,5)$ is considered and included in the tree being built. This completes the spanning tree. The resulting tree (Figure 4.6(b)) has cost 99. □

For clarity, Kruskal's method is written out more formally in Algorithm 4.9. Initially E is the set of all edges in G. The only functions we wish to perform on this set are (1) determine an edge with minimum cost (line 4) and (2) delete this edge (line 5). Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. It is not essential to sort all the edges so long as the next edge for line 4 can be determined easily. If the edges are maintained as a minheap, then the next edge to consider can be obtained in $O(\log |E|)$ time. The construction of the heap itself takes $O(|E|)$ time.

To be able to perform step 6 efficiently, the vertices in G should be grouped together in such a way that one can easily determine whether the vertices v and w are already connected by the earlier selection of edges. If they are, then the edge (v,w) is to be discarded. If they are not, then (v,w) is to be added to t. One possible grouping is to place all vertices in the same connected component of t into a set (all connected components of t will also be trees). Then, two vertices v and w are connected in t iff they are in the same set. For example, when the edge $(2,6)$ is to be considered, the sets are $\{1,2\}, \{3,4,6\}$, and $\{5\}$. Vertices 2 and 6 are in different sets so these sets are combined to give $\{1,2,3,4,6\}$ and $\{5\}$. The next edge to be considered is $(1,4)$. Since vertices 1 and 4 are in the same set, the edge is rejected. The edge $(3,5)$ connects vertices in different sets and results in the final spanning tree. Using the set representation and the union and find algorithms of Section 2.5, we can obtain an efficient (almost linear) implementation of line 6. The computing time is, therefore, determined by the time for lines 4 and 5, which in the worst case is $O(|E| \log |E|)$.

If the representations discussed above are used, then the pseudocode of Algorithm 4.10 results. In line 6 an initial heap of edges is constructed. In line 7 each vertex is assigned to a distinct set (and hence to a distinct tree). The set t is the set of edges to be included in the minimum-cost spanning

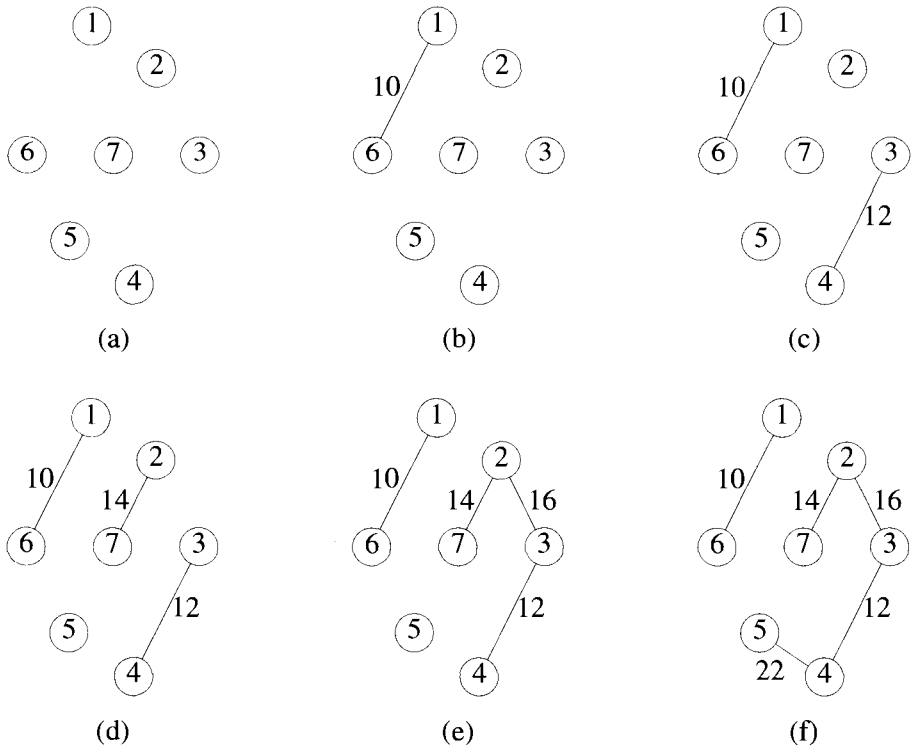


Figure 4.8 Stages in Kruskal's algorithm

tree and i is the number of edges in t. The set t can be represented as a sequential list using a two-dimensional array $t[1:n-1, 1:2]$. Edge (u, v) can be added to t by the assignments $t[i, 1] := u$; and $t[i, 2] := v$;. In the **while** loop of line 10, edges are removed from the heap one by one in nondecreasing order of cost. Line 14 determines the sets containing u and v. If $j \neq k$, then vertices u and v are in different sets (and so in different trees) and edge (u, v) is included into t. The sets containing u and v are combined (line 20). If $u = v$, the edge (u, v) is discarded as its inclusion into t would create a cycle. Line 23 determines whether a spanning tree was found. It follows that $i \neq n - 1$ iff the graph G is not connected. One can verify that the computing time is $O(|E| \log |E|)$, where E is the edge set of G.

Theorem 4.6 Kruskal's algorithm generates a minimum-cost spanning tree for every connected undirected graph G.

```
1  t := ∅;
2  while ((t has less than n − 1 edges) and (E ≠ ∅)) do
3  {
4      Choose an edge (v, w) from E of lowest cost;
5      Delete (v, w) from E;
6      if (v, w) does not create a cycle in t then add (v, w) to t;
7      else discard (v, w);
8  }
```

Algorithm 4.9 Early form of minimum-cost spanning tree algorithm due to Kruskal

```
1  Algorithm Kruskal(E, cost, n, t)
2  // E is the set of edges in G. G has n vertices. cost[u, v] is the
3  // cost of edge (u, v). t is the set of edges in the minimum-cost
4  // spanning tree. The final cost is returned.
5  {
6      Construct a heap out of the edge costs using Heapify;
7      for i := 1 to n do parent[i] := −1;
8      // Each vertex is in a different set.
9      i := 0; mincost := 0.0;
10     while ((i < n − 1) and (heap not empty)) do
11     {
12         Delete a minimum cost edge (u, v) from the heap
13         and reheapify using Adjust;
14         j := Find(u); k := Find(v);
15         if (j ≠ k) then
16         {
17             i := i + 1;
18             t[i, 1] := u; t[i, 2] := v;
19             mincost := mincost + cost[u, v];
20             Union(j, k);
21         }
22     }
23     if (i ≠ n − 1) then write ("No spanning tree");
24     else return mincost;
25 }
```

Algorithm 4.10 Kruskal's algorithm

Proof: Let G be any undirected connected graph. Let t be the spanning tree for G generated by Kruskal's algorithm. Let t' be a minimum-cost spanning tree for G. We show that both t and t' have the same cost.

Let $E(t)$ and $E(t')$ respectively be the edges in t and t'. If n is the number of vertices in G, then both t and t' have $n-1$ edges. If $E(t) = E(t')$, then t is clearly of minimum cost. If $E(t) \neq E(t')$, then let q be a minimum-cost edge such that $q \in E(t)$ and $q \notin E(t')$. Clearly, such a q must exist. The inclusion of q into t' creates a unique cycle (Exercise 5). Let $q, e_1, e_2, \ldots, e_k$ be this unique cycle. At least one of the e_i's, $1 \leq i \leq k$, is not in $E(t)$ as otherwise t would also contain the cycle $q, e_1, e_2, \ldots, e_k$. Let e_j be an edge on this cycle such that $e_j \notin E(t)$. If e_j is of lower cost than q, then Kruskal's algorithm will consider e_j before q and include e_j into t. To see this, note that all edges in $E(t)$ of cost less than the cost of q are also in $E(t')$ and do not form a cycle with e_j. So $cost(e_j) \geq cost(q)$.

Now, reconsider the graph with edge set $E(t') \cup \{q\}$. Removal of any edge on the cycle $q, e_1, e_2, \ldots, e_k$ will leave behind a tree t'' (Exercise 5). In particular, if we delete the edge e_j, then the resulting tree t'' will have a cost no more than the cost of t' (as $cost(e_j) \geq cost(e)$). Hence, t'' is also a minimum-cost tree.

By repeatedly using the transformation described above, tree t' can be transformed into the spanning tree t without any increase in cost. Hence, t is a minimum-cost spanning tree. □

4.5.3 An Optimal Randomized Algorithm (*)

Any algorithm for finding the minimum-cost spanning tree of a given graph $G(V, E)$ will have to spend $\Omega(|V| + |E|)$ time in the worst case, since it has to examine each node and each edge at least once before determining the correct answer. A randomized Las Vegas algorithm that runs in time $\tilde{O}(|V| + |E|)$ can be devised as follows: (1) Randomly sample m edges from G (for some suitable m). (2) Let G' be the induced subgraph; that is, G' has V as its node set and the sampled edges in its edge set. The subgraph G' need not be connected. Recursively find a minimum-cost spanning tree for each component of G'. Let F be the resultant *minimum-cost spanning forest* of G'. (3) Using F, eliminate certain edges (called the *F-heavy edges*) of G that cannot possibly be in a minimum-cost spanning tree. Let G'' be the graph that results from G after elimination of the F-heavy edges. (4) Recursively find a minimum-cost spanning tree for G''. This will also be a minimum-cost spanning tree for G.

Steps 1 to 3 are useful in reducing the number of edges in G. The algorithm can be speeded up further if we can reduce the number of nodes in the input graph as well. Such a node elimination can be effected using the *Borůvka steps*. In a Borůvka step, for each node, an incident edge with minimum weight is chosen. For example in Figure 4.9(a), the edge $(1, 3)$ is

chosen for node 1, the edge $(6,7)$ is chosen for node 7, and so on. All the chosen edges are shown with thick lines. The connected components of the induced graph are found. In the example of Figure 4.9(a), the nodes 1, 2, and 3 form one component, the nodes 4 and 5 form a second component, and the nodes 6 and 7 form another component. Replace each component with a single node. The component with nodes 1, 2, and 3 is replaced with the node a. The other two components are replaced with the nodes b and c, respectively. Edges within the individual components are thrown away. The resultant graph is shown in Figure 4.9(b). In this graph keep only an edge of minimum weight between any two nodes. Delete any isolated nodes.

Since an edge is chosen for every node, the number of nodes after one Borůvka step reduces by a factor of at least two. A minimum-cost spanning tree for the reduced graph can be extended easily to get a minimum-cost spanning tree for the original graph. If E' is the set of edges in the minimum-cost spanning tree of the reduced graph, we simply include into E' the edges chosen in the Borůvka step to obtain the minimum-cost spanning tree edges for the original graph. In the example of Figure 4.9, a minimum-cost spanning tree for (c) will consist of the edges (a,b) and (b,c). Thus a minimum-cost spanning tree for the graph of (a) will have the edges: $(1,3),(3,2),(4,5),(6,7),(3,4)$, and $(2,6)$. More details of the algorithms are given below.

Definition 4.2 Let F be a forest that forms a subgraph of a given weighted graph $G(V,E)$. If u and v are any two nodes in F, let $F(u,v)$ denote the path (if any) connecting u and v in F and let $Fcost(u,v)$ denote the maximum weight of any edge in the path $F(u,v)$. If there is no path between u and v in F, $Fcost(u,v)$ is taken to be ∞. Any edge (x,y) of G is said to be F-heavy if $cost[x,y] > Fcost(x,y)$ and F-light otherwise. □

Note that all the edges of F are F-light. Also, any F-heavy edge cannot belong to a minimum-cost spanning tree of G. The proof of this is left as an exercise. The randomized algorithm applies two Borůvka steps to reduce the number of nodes in the input graph. Next, it samples the edges of G and processes them to eliminate a constant fraction of them. A minimum-cost spanning tree for the resultant reduced graph is recursively computed. From this tree, a spanning tree for G is obtained. A detailed description of the algorithm appears as Algorithm 4.11.

Lemma 4.3 states that Step 4 can be completed in time $O(|V|+|E|)$. The proof of this can be found in the references supplied at the end of this chapter. Step 1 takes $O(|V|+|E|)$ time and step 2 takes $O(|E|)$ time. Step 6 takes $O(|E|)$ time as well. The time taken in all the recursive calls in steps 3 and 5 can be shown to be $O(|V|+|E|)$. For a proof, see the references at the end of the chapter. A crucial fact that is used in the proof is that both the number of nodes and the number of edges are reduced by a constant factor, with high probability, in each level of recursion.

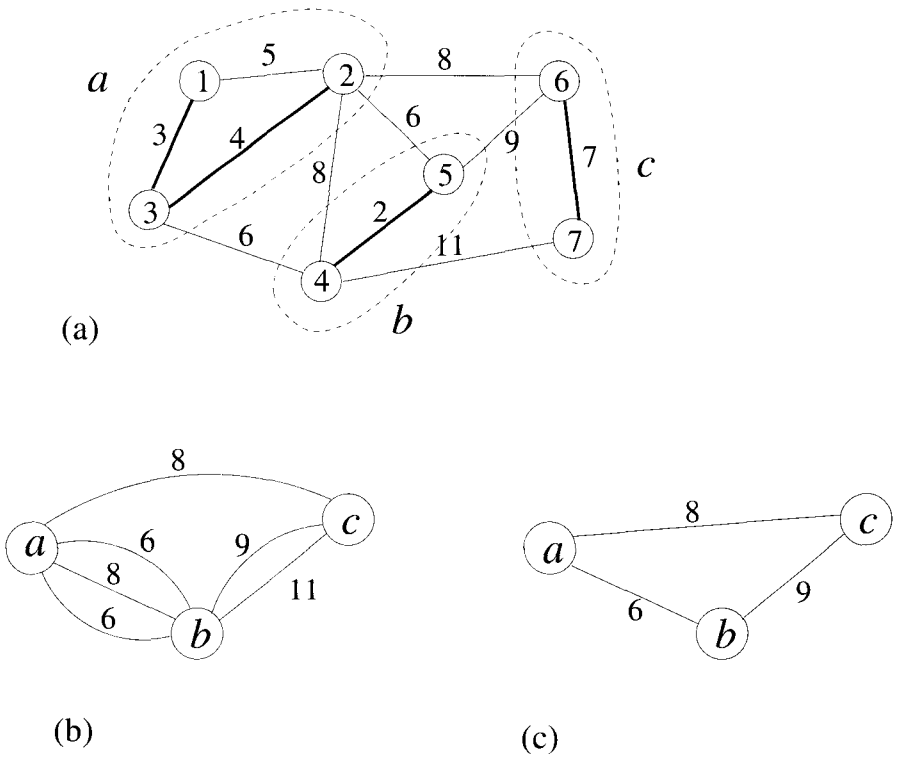


Figure 4.9 A Borůvka step

Lemma 4.3 Let $G(V, E)$ be any weighted graph and let F be a subgraph of G that forms a forest. Then, all the F-heavy edges of G can be identified in time $O(|V| + |E|)$. □

Theorem 4.7 A minimum-weight spanning tree for any given weighted graph can be computed in time $\tilde{O}(|V| + |E|)$. □

EXERCISES

1. Compute a minimum cost spanning tree for the graph of Figure 4.10 using (a) Prim's algorithm and (b) Kruskal's algorithm.

2. Prove that Prim's method of this section generates minimum-cost spanning trees.

Step 1. Apply two Borůvka steps. At the end, the number of nodes will have decreased by a factor at least 4. Let the resultant graph be $\widetilde{G}(\widetilde{V}, \widetilde{E})$.

Step 2. Form a subgraph $G'(V', E')$ of $\widetilde{G}$, where each edge of $\widetilde{G}$ is chosen randomly to be in E' with probability $\frac{1}{2}$. The expected number of edges in E' is $\frac{|\widetilde{E}|}{2}$.

Step 3. Recursively find a minimum-cost spanning forest F for G'.

Step 4. Eliminate all the F-heavy edges from $\widetilde{G}$. With high probability, at least a constant fraction of the edges of $\widetilde{G}$ will be eliminated. Let G'' be the resultant graph.

Step 5. Compute a minimum-cost spanning tree (call it T'') for G'' recursively. The tree T'' will also be a minimum-cost spanning tree for $\widetilde{G}$.

Step 6. Return the edges of T'' together with the edges chosen in the Borůvka steps of step 1. These are the edges of a minimum-cost spanning tree for G.

Algorithm 4.11 An optimal randomized algorithm

3. (a) Rewrite Prim's algorithm under the assumption that the graphs are represented by adjacency lists.
   (b) Program and run the above version of Prim's algorithm against Algorithm 4.9. Compare the two on a representative set of graphs.
   (c) Analyze precisely the computing time and space requirements of your new version of Prim's algorithm using adjacency lists.

4. Program and run Kruskal's algorithm, described in Algorithm 4.10. You will have to modify functions Heapify and Adjust of Chapter 2. Use the same test data you devised to test Prim's algorithm in Exercise 3.

5. (a) Show that if t is a spanning tree for the undirected graph G, then the addition of an edge q, $q \notin E(t)$ and $q \in E(G)$, to t creates a unique cycle.

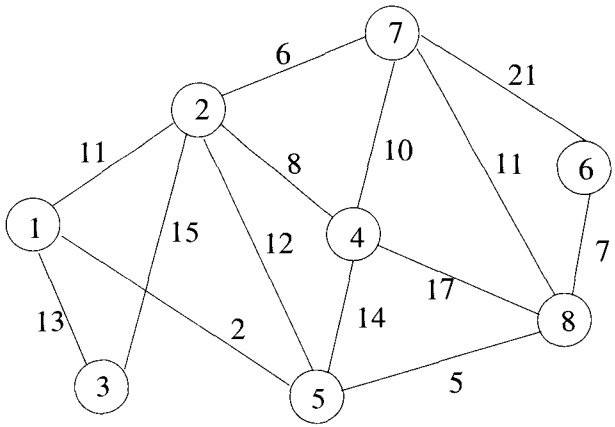


Figure 4.10 Graph for Exercise 1

(b) Show that if any of the edges on this unique cycle is deleted from $E(t) \cup \{q\}$, then the remaining edges form a spanning tree of G.

6. In Figure 4.9, find a minimum-cost spanning tree for the graph of part (c) and extend the tree to obtain a minimum cost spanning tree for the graph of part (a). Verify the correctness of your answer by applying either Prim's algorithm or Kruskal's algorithm on the graph of part (a).

7. Let $G(V, E)$ be any weighted connected graph.

   (a) If C is any cycle of G, then show that the heaviest edge of C cannot belong to a minimum-cost spanning tree of G.

   (b) Assume that F is a forest that is a subgraph of G. Show that any F-heavy edge of G cannot belong to a minimum-cost spanning tree of G.

8. By considering the complete graph with n vertices, show that the number of spanning trees in an n vertex graph can be greater than $2^{n-1}-2$.

4.6 OPTIMAL STORAGE ON TAPES

There are n programs that are to be stored on a computer tape of length l. Associated with each program i is a length $l_i, 1 \leq i \leq n$. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of

the programs is at most l. We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \le k \le j} l_{i_k}$. If all programs are retrieved equally often, then the expected or *mean retrieval time* (MRT) is $(1/n) \sum_{1 \le j \le n} t_j$. In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}$.

Example 4.8 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective d values are:

ordering I	$d(I)$		
$1, 2, 3$	$5 + 5 + 10 + 5 + 10 + 3$	$=$	38
$1, 3, 2$	$5 + 5 + 3 + 5 + 3 + 10$	$=$	31
$2, 1, 3$	$10 + 10 + 5 + 10 + 5 + 3$	$=$	43
$2, 3, 1$	$10 + 10 + 3 + 10 + 3 + 5$	$=$	41
$3, 1, 2$	$3 + 3 + 5 + 3 + 5 + 10$	$=$	29
$3, 2, 1$	$3 + 3 + 10 + 3 + 10 + 5$	$=$	34

The optimal ordering is $3, 1, 2$. □

A greedy approach to building the required permutation would choose the next program on the basis of some optimization measure. One possible measure would be the d value of the permutation constructed so far. The next program to be stored on the tape would be one that minimizes the increase in d. If we have already constructed the permutation $i_1, i_2, \ldots, i_r$, then appending program j gives the permutation $i_1, i_2, \ldots, i_r, i_{r+1} = j$. This increases the d value by $\sum_{1 \le k \le r} l_{i_k} + l_j$. Since $\sum_{1 \le k \le r} l_{i_k}$ is fixed and independent of j, we trivially observe that the increase in d is minimized if the next program chosen is the one with the least length from among the remaining programs.

The greedy algorithm resulting from the above discussion is so simple that we won't bother to write it out. The greedy method simply requires us to store the programs in nondecreasing order of their lengths. This ordering can be carried out in $O(n \log n)$ time using an efficient sorting algorithm (e.g., heap sort from Chapter 2). For the programs of Example 4.8, note that the permutation that yields an optimal solution is the one in which the programs are in nondecreasing order of their lengths. Theorem 4.8 shows that the MRT is minimized when programs are stored in this order.

Theorem 4.8 If $l_1 \leq l_2 \leq \cdots \leq l_n$, then the ordering $i_j = j, 1 \leq j \leq n$, minimizes

$$\sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$$

over all possible permutations of the i_j.

Proof: Let $I = i_1, i_2, \ldots, i_n$ be any permutation of the index set $\{1, 2, \ldots, n\}$. Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j} = \sum_{k=1}^{n} (n - k + 1) l_{i_k}$$

If there exist a and b such that $a < b$ and $l_{i_a} > l_{i_b}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}} (n - k + 1) l_{i_k} \right] + (n - a + 1) l_{i_b} + (n - b + 1) l_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n - a + 1)(l_{i_a} - l_{i_b}) + (n - b + 1)(l_{i_b} - l_{i_a}) \\ &= (b - a)(l_{i_a} - l_{i_b}) \\ &> 0 \end{aligned}$$

Hence, no permutation that is not in nondecreasing order of the l_i's can have minimum d. It is easy to see that all permutations in nondecreasing order of the l_i's have the same d value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the d value. □

The tape storage problem can be extended to several tapes. If there are $m > 1$ tapes, $T_0, \ldots, T_{m-1}$, then the programs are to be distributed over these tapes. For each tape a storage permutation is to be provided. If I_j is the storage permutation for the subset of programs on tape j, then $d(I_j)$ is as defined earlier. The *total retrieval time* (TD) is $\sum_{0 \leq j \leq m-1} d(I_j)$. The objective is to store the programs in such a way as to minimize TD.

The obvious generalization of the solution for the one-tape case is to consider the programs in nondecreasing order of l_i's. The program currently

```
1   Algorithm Store(n, m)
2   // n is the number of programs and m the number of tapes.
3   {
4       j := 0; // Next tape to store on
5       for i := 1 to n do
6       {
7           write ("append program", i,
8               "to permutation for tape", j);
9           j := (j + 1) mod m;
10      }
11  }
```

Algorithm 4.12 Assigning programs to tapes

being considered is placed on the tape that results in the minimum increase in TD. This tape will be the one with the least amount of tape used so far. If there is more than one tape with this property, then the one with the smallest index can be used. If the jobs are initially ordered so that $l_1 \leq l_2 \leq \cdots \leq l_n$, then the first m programs are assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The next m programs will be assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The general rule is that program i is stored on tape $T_{i \bmod m}$. On any given tape the programs are stored in nondecreasing order of their lengths. Algorithm 4.12 presents this rule in pseudocode. It assumes that the programs are ordered as above. It has a computing time of $\Theta(n)$ and does not need to know the program lengths. Theorem 4.9 proves that the resulting storage pattern is optimal.

Theorem 4.9 If $l_1 \leq l_2 \leq \cdots \leq l_n$, then Algorithm 4.12 generates an optimal storage pattern for m tapes.

Proof: In any storage pattern for m tapes, let r_i be one greater than the number of programs following program i on its tape. Then the total retrieval time TD is given by

$$TD = \sum_{i=1}^{n} r_i l_i$$

In any given storage pattern, for any given n, there can be at most m programs for which $r_i = j$. From Theorem 4.8 it follows that TD is minimized if the m longest programs have $r_i = 1$, the next m longest programs have

$r_i = 2$, and so on. When programs are ordered by length, that is, $l_1 \leq l_2 \leq \cdots \leq l_n$, then this minimization criteria is satisfied if $r_i = \lceil (n-i+1)/m \rceil$. Observe that Algorithm 4.12 results in a storage pattern with these r_i's. □

The proof of Theorem 4.9 shows that there are many storage patterns that minimize TD. If we compute $r_i = \lceil (n-i+1)/m \rceil$ for each program i, then so long as all programs with the same r_i are stored on different tapes and have $r_i - 1$ programs following them, TD is the same. If n is a multiple of m, then there are at least $(m!)^{n/m}$ storage patterns that minimize TD. Algorithm 4.12 produces one of these.

EXERCISES

1. Find an optimal placement for 13 programs on three tapes T_0, T_1, and T_2, where the programs are of lengths $12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10$, and 6.

2. Show that replacing the code of Algorithm 4.12 by

```
for i := 1 to n do
    write ("append program", i, "to permutation for
        tape", (i - 1) mod m);
```

   does not affect the output.

3. Let $P_1, P_2, \ldots, P_n$ be a set of n programs that are to be stored on a tape of length l. Program P_i requires a_i amount of tape. If $\sum a_i \leq l$, then clearly all the programs can be stored on the tape. So, assume $\sum a_i > l$. The problem is to select a maximum subset Q of the programs for storage on the tape. (A maximum subset is one with the maximum number of programs in it). A greedy algorithm for this problem would build the subset Q by including programs in nondecreasing order of a_i.

   (a) Assume the P_i are ordered such that $a_1 \leq a_2 \leq \cdots \leq a_n$. Write a function for the above strategy. Your function should output an array $s[1:n]$ such that $s[i] = 1$ if P_i is in Q and $s[i] = 0$ otherwise.

   (b) Show that this strategy always finds a maximum subset Q such that $\sum_{P_i \in Q} a_i \leq l$.

   (c) Let Q be the subset obtained using the above greedy strategy. How small can the tape utilization ratio $(\sum_{P_i \in Q} a_i)/l$ get?

   (d) Suppose the objective now is to determine a subset of programs that maximizes the tape utilization ratio. A greedy approach

would be to consider programs in nonincreasing order of a_i. If there is enough space left on the tape for P_i, then it is included in Q. Assume the programs are ordered so that $a_1 \geq a_2 \geq \cdots \geq a_n$. Write a function incorporating this strategy. What is its time and space complexity?

(e) Show that the strategy of part (d) doesn't necessarily yield a subset that maximizes $(\sum_{P_i \in Q} a_i)/l$. How small can this ratio get? Prove your bound.

4. Assume n programs of lengths $l_1, l_2, \ldots, l_n$ are to be stored on a tape. Program i is to be retrieved with frequency f_i. If the programs are stored in the order $i_1, i_2, \ldots, i_n$, the *expected retrieval time* (ERT) is

$$\left[\sum_j (f_{i_j} \sum_{k=1}^{j} l_{i_k})\right] / \sum f_i$$

(a) Show that storing the programs in nondecreasing order of l_i does not necessarily minimize the ERT.

(b) Show that storing the programs in nonincreasing order of f_i does not necessarily minimize the ERT.

(c) Show that the ERT is minimized when the programs are stored in nonincreasing order of f_i/l_i.

5. Consider the tape storage problem of this section. Assume that two tapes $T1$ and $T2$, are available and we wish to distribute n given programs of lengths $l_1, l_2, \ldots, l_n$ onto these two tapes in such a manner that the maximum retrieval time is minimized. That is, if A and B are the sets of programs on the tapes $T1$ and $T2$ respectively, then we wish to choose A and B such that $\max \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$ is minimized. A possible greedy approach to obtaining A and B would be to start with A and B initially empty. Then consider the programs one at a time. The program currently being considered is assigned to set A if $\sum_{i \in A} l_i = \min \{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$; otherwise it is assigned to B. Show that this does not guarantee optimal solutions even if $l_1 \leq l_2 \leq \cdots \leq l_n$. Show that the same is true if we require $l_1 \geq l_2 \geq \cdots \geq l_n$.

4.7 OPTIMAL MERGE PATTERNS

In Section 3.4 we saw that two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time $O(n+m)$. When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if

files x_1, x_2, x_3, and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then we could merge y_1 and x_3 to get y_2. Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 and get the desired sorted file. Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Example 4.9 The files x_1, x_2, and x_3 are three sorted files of length 30, 20, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first. □

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n-record file and an m-record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files $(x_1, \ldots, x_5)$ with sizes $(20, 30, 10, 5, 30)$, our greedy rule would generate the following merge pattern: merge x_4 and x_3 to get z_1 ($|z_1| = 15$), merge z_1 and x_1 to get z_2 ($|z_2| = 35$), merge x_2 and x_5 to get z_3 ($|z_3| = 60$), and merge z_2 and z_3 to get the answer z_4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.

The merge pattern such as the one just described will be referred to as a *two-way merge pattern* (each merge step involves the merging of two files). The two-way merge patterns can be represented by binary merge trees. Figure 4.11 shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent the given five files. These nodes are called *external nodes*. The remaining nodes are drawn as circles and are called *internal nodes*. Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number of records) of the file represented by that node.

The external node x_4 is at a distance of 3 from the root node z_4 (a node at level i is at a distance of $i - 1$ from the root). Hence, the records of file x_4 are moved three times, once to get z_1, once again to get z_2, and finally one more time to get z_4. If d_i is the distance from the root to the external

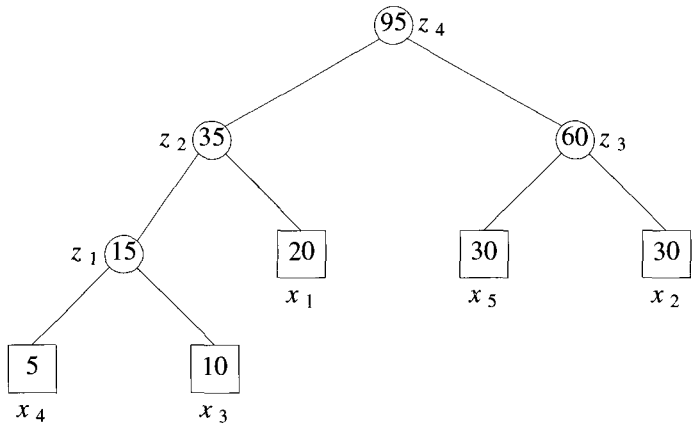


Figure 4.11 Binary merge tree representing a merge pattern

node for file x_i and q_i, the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i$$

This sum is called the *weighted external path length* of the tree.

An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function Tree of Algorithm 4.13 uses the greedy rule stated earlier to obtain a two-way merge tree for n files. The algorithm has as input a list *list* of n trees. Each node in a tree has three fields, *lchild*, *rchild*, and *weight*. Initially, each tree in *list* has exactly one node. This node is an external node and has *lchild* and *rchild* fields zero whereas *weight* is the length of one of the n files to be merged. During the course of the algorithm, for any tree in *list* with root node t, $t \rightarrow weight$ is the length of the merged file it represents ($t \rightarrow weight$ equals the sum of the lengths of the external nodes in tree t). Function Tree uses two functions, Least(*list*) and Insert(*list*, t). Least(*list*) finds a tree in *list* whose root has least *weight* and returns a pointer to this tree. This tree is removed from *list*. Insert(*list*, t) inserts the tree with root t into *list*. Theorem 4.10 shows that Tree (Algorithm 4.13) generates an optimal two-way merge tree.

```
    treenode = record {
        treenode * lchild; treenode * rchild;
        integer weight;
    };

1   Algorithm Tree(n)
2   // list is a global list of n single node
3   // binary trees as described above.
4   {
5       for i := 1 to n - 1 do
6       {
7           pt := new treenode; // Get a new tree node.
8           (pt → lchild) := Least(list); // Merge two trees with
9           (pt → rchild) := Least(list); // smallest lengths.
10          (pt → weight) := ((pt → lchild) → weight)
11                  +((pt → rchild) → weight);
12          Insert(list, pt);
13      }
14      return Least(list); // Tree left in list is the merge tree.
15  }
```

Algorithm 4.13 Algorithm to generate a two-way merge tree

Example 4.10 Let us see how algorithm Tree works when *list* initially represents six files with lengths $(2, 3, 5, 7, 9, 13)$. Figure 4.12 shows *list* at the end of each iteration of the **for** loop. The binary merge tree that results at the end of the algorithm can be used to determine which files are merged. Merging is performed on those files which are lowest (have the greatest depth) in the tree. □

The main **for** loop in Algorithm 4.13 is executed $n - 1$ times. If *list* is kept in nondecreasing order according to the *weight* value in the roots, then Least(*list*) requires only $O(1)$ time and Insert($list, t$) can be done in $O(n)$ time. Hence the total time taken is $O(n^2)$. In case *list* is represented as a minheap in which the root value is less than or equal to the values of its children (Section 2.4), then Least(*list*) and Insert($list, t$) can be done in $O(\log n)$ time. In this case the computing time for Tree is $O(n \log n)$. Some speedup may be obtained by combining the Insert of line 12 with the Least of line 9.

Theorem 4.10 If *list* initially contains $n \geq 1$ single node trees with *weight* values $(q_1, q_2, \ldots, q_n)$, then algorithm Tree generates an optimal two-way merge tree for n files with these lengths.

Proof: The proof is by induction on n. For $n = 1$, a tree with no internal nodes is returned and this tree is clearly optimal. For the induction hypothesis, assume the algorithm generates an optimal two-way merge tree for all $(q_1, q_2, \ldots, q_m)$, $1 \leq m < n$. We show that the algorithm also generates optimal trees for all $(q_1, q_2, \ldots, q_n)$. Without loss of generality, we can assume that $q_1 \leq q_2 \leq \cdots \leq q_n$ and q_1 and q_2 are the values of the *weight* fields of the trees found by algorithm Least in lines 8 and 9 during the first iteration of the **for** loop. Now, the subtree T of Figure 4.13 is created. Let T' be an optimal two-way merge tree for $(q_1, q_2, \ldots, q_n)$. Let p be an internal node of maximum distance from the root. If the children of p are not q_1 and q_2, then we can interchange the present children with q_1 and q_2 without increasing the weighted external path length of T'. Hence, T is also a subtree in an optimal merge tree. If we replace T in T' by an external node with weight $q_1 + q_2$, then the resulting tree T'' is an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. From the induction hypothesis, after replacing T by the external node with value $q_1 + q_2$, function Tree proceeds to find an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. Hence, Tree generates an optimal merge tree for $(q_1, q_2, \ldots, q_n)$. □

The greedy method to generate merge trees also works for the case of k-ary merging. In this case the corresponding merge tree is a k-ary tree. Since all internal nodes must have degree k, for certain values of n there is no corresponding k-ary merge tree. For example, when $k = 3$, there is no k-ary merge tree with $n = 2$ external nodes. Hence, it is necessary to introduce a certain number of dummy external nodes. Each dummy node is assigned a q_i of zero. This dummy value does not affect the weighted external path length of the resulting k-ary tree. Exercise 2 shows that a k-ary tree with all internal nodes having degree k exists only when the number of external nodes n satisfies the equality $n \bmod (k-1) = 1$. Hence, at most $k-2$ dummy nodes have to be added. The greedy rule to generate optimal merge trees is: at each step choose k subtrees with least length for merging. Exercise 3 proves the optimality of this rule.

Huffman Codes

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages $M_1, \ldots, M_{n+1}$. Each code is a binary string that is used for transmission of the corresponding message. At the receiving end the code is decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages.

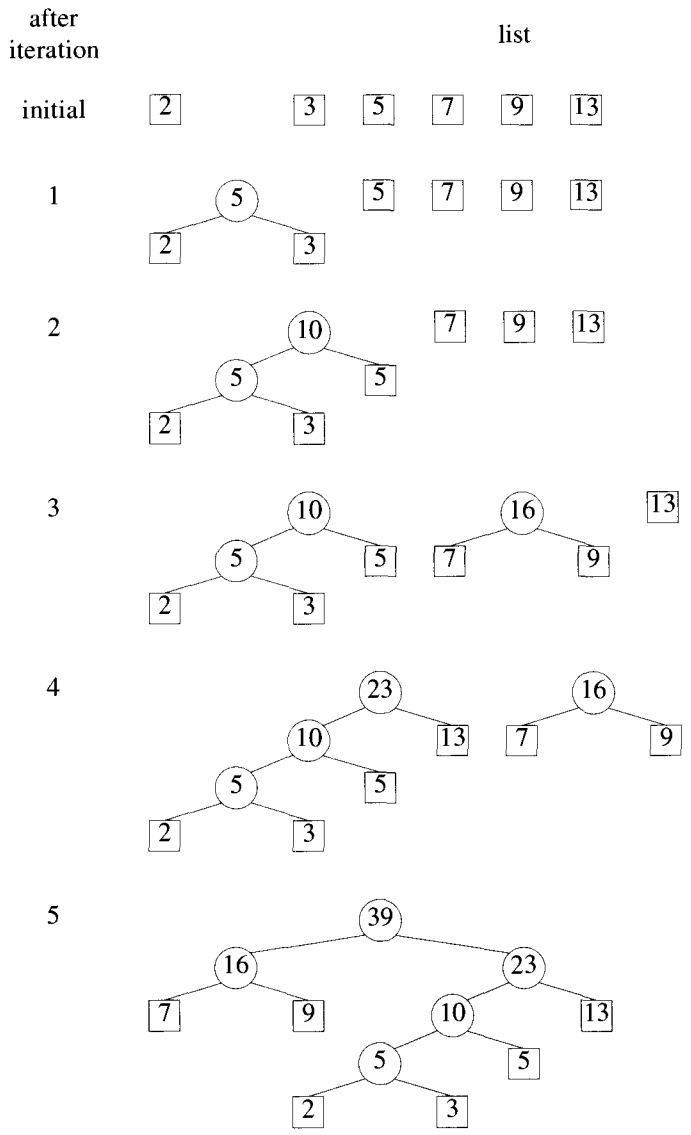


Figure 4.12 Trees in *list* of Tree for Example 4.10

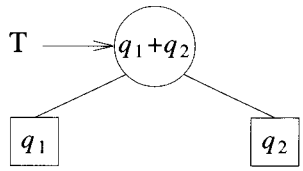


Figure 4.13 The simplest binary merge tree

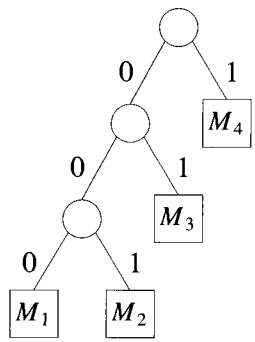


Figure 4.14 Huffman codes

The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure 4.14 corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3, and M_4 respectively. These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\sum_{1 \le i \le n+1} q_i d_i$, where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\sum_{1 \le i \le n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code that minimizes expected decode time also minimizes the expected length of a message.

EXERCISES

1. Find an optimal binary merge pattern for ten files whose lengths are $28, 32, 12, 5, 84, 53, 91, 35, 3,$ and 11.

2. (a) Show that if all internal nodes in a tree have degree k, then the number n of external nodes is such that $n \bmod (k-1) = 1$.
   (b) Show that for every n such that $n \bmod (k-1) = 1$, there exists a k-ary tree T with n external nodes (in a k-ary tree all nodes have degree at most k). Also show that all internal nodes of T have degree k.

3. (a) Show that if $n \bmod (k-1) = 1$, then the greedy rule described following Theorem 4.10 generates an optimal k-ary merge tree for all $(q_1, q_2, \ldots, q_n)$.
   (b) Draw the optimal three-way merge tree obtained using this rule when $(q_1, q_2, \ldots, q_{11}) = (3, 7, 8, 9, 15, 16, 18, 20, 23, 25, 28)$.

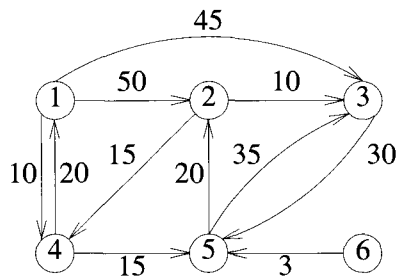
4. Obtain a set of optimal Huffman codes for the messages $(M_1, \ldots, M_7)$ with relative frequencies $(q_1, \ldots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes.

5. Let T be a decode tree. An optimal decode tree minimizes $\sum q_i d_i$. For a given set of q's, let D denote all the optimal decode trees. For any tree $T \in D$, let $L(T) = \max\{d_i\}$ and let $\mathrm{SL}(T) = \sum d_i$. Schwartz has shown that there exists a tree $T^* \in D$ such that $L(T^*) = \min_{T \in D}\{L(T)\}$ and $\mathrm{SL}(T^*) = \min_{T \in D}\{SL(T)\}$.

   (a) For $(q_1, \ldots, q_8) = (1, 1, 2, 2, 4, 4, 4, 4)$ obtain trees $T1$ and $T2$ such that $L(T1) > L(T2)$.
   (b) Using the data of a, obtain $T1$ and $T2 \in D$ such that $L(T1) = L(T2)$ but $SL(T1) > SL(T2)$.
   (c) Show that if the subalgorithm Least used in algorithm Tree is such that in case of a tie it returns the tree with least depth, then Tree generates a tree with the properties of T^*.

4.8 SINGLE-SOURCE SHORTEST PATHS

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:



(a) Graph

Path	*Length*
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Figure 4.15 Graph and shortest paths from vertex 1 to all destinations

- Is there a path from A to B?
- If there is more than one path from A to B, which is the shortest path?

The problems defined by these questions are special cases of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the *source*, and the last vertex the *destination*. The graphs are digraphs to allow for one-way streets. In the problem we consider, we are given a directed graph $G = (V, E)$, a weighting function *cost* for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example 4.11 Consider the directed graph of Figure 4.15(a). The numbers on the edges are the weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is $1, 4, 5, 2$. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path $1, 2$ which is of length 50. There is no path from 1 to 6. Figure 4.15(b) lists the shortest paths from node 1 to nodes $4, 5, 2$, and 3, respectively. The paths have been listed in nondecreasing order of path length. □

To formulate a greedy-based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by

one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in nondecreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on. For the graph of Figure 4.15(a) the nearest vertex to $v_0 = 1$ is 4 ($cost[1,4] = 10$). The path $1,4$ is the first path generated. The second nearest vertex to node 1 is 5 and the distance between 1 and 5 is 25. The path $1,4,5$ is the next path generated. In order to generate the shortest paths in this order, we need to be able to determine (1) the next vertex to which a shortest path must be generated and (2) a shortest path to this vertex. Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated. For w not in S, let $dist[w]$ be the length of the shortest path starting from v_0, going through only those vertices that are in S, and ending at w. We observe that:

1. If the next shortest path is to vertex u, then the path begins at v_0, ends at u, and goes through only those vertices that are in S. To prove this, we must show that all the intermediate vertices on the shortest path to u are in S. Assume there is a vertex w on this path that is not in S. Then, the v_0 to u path also contains a path from v_0 to w that is of length less than the v_0 to u path. By assumption the shortest paths are being generated in nondecreasing order of path length, and so the shorter path v_0 to w must already have been generated. Hence, there can be no intermediate vertex that is not in S.

2. The destination of the next path generated must be that of vertex u which has the minimum distance, $dist[u]$, among all vertices not in S. This follows from the definition of $dist$ and observation 1. In case there are several vertices not in S with the same $dist$, then any of these may be selected.

3. Having selected a vertex u as in observation 2 and generated the shortest v_0 to u path, vertex u becomes a member of S. At this point the length of the shortest paths starting at v_0, going though vertices only in S, and ending at a vertex w not in S may decrease; that is, the value of $dist[w]$ may change. If it does change, then it must be due to a shorter path starting at v_0 and going to u and then to w. The intermediate vertices on the v_0 to u path and the u to w path must all be in S. Further, the v_0 to u path must be the shortest such path; otherwise $dist[w]$ is not defined properly. Also, the u to w path can be chosen so as not to contain any intermediate vertices. Therefore,

we can conclude that if $dist[w]$ is to change (i.e., decrease), then it is because of a path from v_0 to u to w, where the path from v_0 to u is the shortest such path and the path from u to w is the edge $\langle u, w \rangle$. The length of this path is $dist[u] + cost[u, w]$.

The above observations lead to a simple Algorithm 4.14 for the single-source shortest path problem. This algorithm (known as Dijkstra's algorithm) only determines the lengths of the shortest paths from v_0 to all other vertices in G. The generation of the paths requires a minor extension to this algorithm and is left as an exercise. In the function ShortestPaths (Algorithm 4.14) it is assumed that the n vertices of G are numbered 1 through n. The set S is maintained as a bit array with $S[i] = 0$ if vertex i is not in S and $S[i] = 1$ if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with $cost[i, j]$'s being the weight of the edge $\langle i, j \rangle$. The weight $cost[i, j]$ is set to some large number, ∞, in case the edge $\langle i, j \rangle$ is not in $E(G)$. For $i = j$, $cost[i, j]$ can be set to any nonnegative number without affecting the outcome of the algorithm.

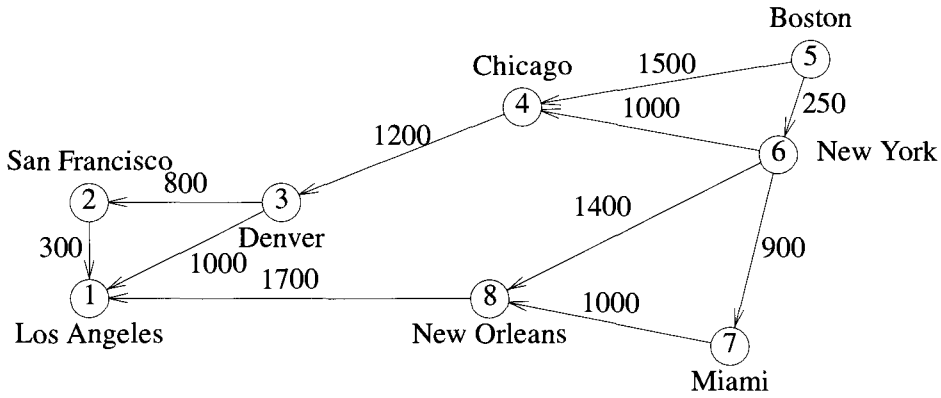
From our earlier discussion, it is easy to see that the algorithm is correct. The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this, note that the **for** loop of line 7 in Algorithm 4.14 takes $\Theta(n)$ time. The **for** loop of line 12 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at lines 15 and 16 to select the next vertex and again at the **for** loop of line 18 to update $dist$. So the total time for this loop is $O(n^2)$. In case a list t of vertices currently not in s is maintained, then the number of nodes on this list would at any time be $n - num$. This would speed up lines 15 and 16 and the **for** loop of line 18, but the asymptotic time would remain $O(n^2)$. This and other variations of the algorithm are explored in the exercises.

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $\Omega(|E|)$. Since cost adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in G, and so any shortest path algorithm using this representation must take $\Omega(n^2)$ time. For this representation then, algorithm ShortestPaths is optimal to within a constant factor. If a change to adjacency lists is made, the overall frequency of the **for** loop of line 18 can be brought down to $O(|E|)$ (since $dist$ can change only for vertices adjacent from u). If $V - S$ is maintained as a red-black tree (see Section 2.4.2), each execution of lines 15 and 16 takes $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). Each update in line 21 takes $O(\log n)$ time as well (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                      dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Algorithm 4.14 Greedy algorithm to generate shortest paths

Example 4.12 Consider the eight vertex digraph of Figure 4.16(a) with cost adjacency matrix as in Figure 4.16(b). The values of *dist* and the vertices selected at each iteration of the **for** loop of line 12 in Algorithm 4.14 for finding all the shortest paths from Boston are shown in Figure 4.17. To begin with, S contains only Boston. In the first iteration of the **for** loop (that is, for $num = 2$), the city u that is not in S and whose $dist[u]$ is minimum is identified to be New York. New York enters the set S. Also the $dist[\,]$ values of Chicago, Miami, and New Orleans get altered since there are shorter paths to these cities via New York. In the next iteration of the **for** loop, the city that enters S is Miami since it has the smallest $dist[\,]$ value from among all the nodes not in S. None of the $dist[\,]$ values are altered. The algorithm continues in a similar fashion and terminates when only seven of the eight vertices are in S. By the definition of *dist*, the distance of the last vertex, in this case Los Angeles, is correct as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. □



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Figure 4.16 Figures for Example 4.12

One can easily verify that the edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G. This spanning tree is called a *shortest-path spanning tree*. Clearly, this spanning tree may be different for different root vertices v. Figure 4.18 shows a graph G, its minimum-cost spanning tree, and a shortest-path spanning tree from vertex 1.

Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Figure 4.17 Action of ShortestPaths

EXERCISES

1. Use algorithm ShortestPaths to obtain in nondecreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the digraph of Figure 4.19.

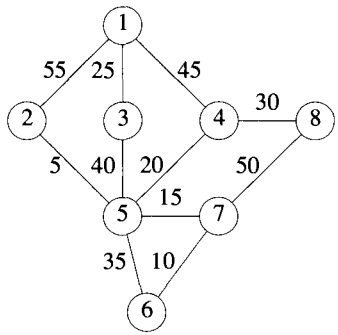
2. Using the directed graph of Figure 4.20 explain why ShortestPaths will not work properly. What is the shortest path between vertices v_1 and v_7 ?

3. Rewrite algorithm ShortestPaths under the following assumptions:

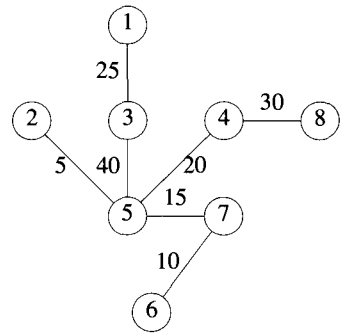
   (a) G is represented by its adjacency lists. The head nodes are HEAD(1),..., HEAD(n) and each list node has three fields: VERTEX, COST, and LINK. COST is the length of the corresponding edge and n the number of vertices in G.

   (b) Instead of representing S, the set of vertices to which the shortest paths have already been found, the set $T = V(G) - S$ is represented using a linked list. What can you say about the computing time of your new algorithm relative to that of ShortestPaths?

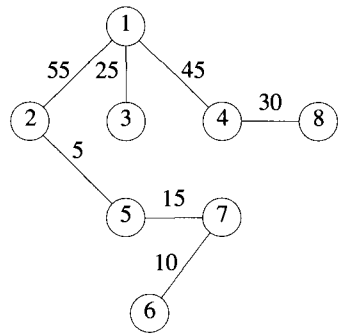
4. Modify algorithm ShortestPaths so that it obtains the shortest paths in addition to the lengths of these paths. What is the computing time of your algorithm?



(a) A Graph



(b) Minimum cost spanning tree



(c) Shortest path spanning tree from vertex 1.

Figure 4.18 Graphs and spanning trees

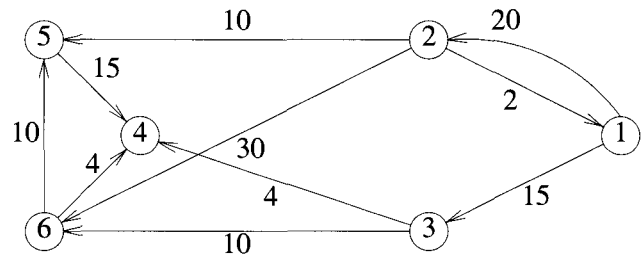


Figure 4.19 Directed graph

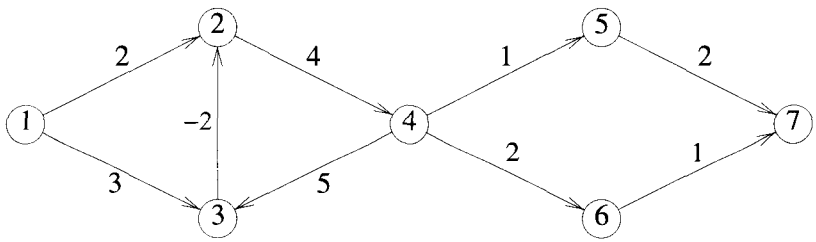


Figure 4.20 Another directed graph

4.9 REFERENCES AND READINGS

The linear time algorithm in Section 4.3 for the tree vertex splitting problem can be found in "Vertex upgrading problems for VLSI," by D. Paik, Ph.D. thesis, Department of Computer Science, University of Minnesota, October 1991.

The two greedy methods for obtaining minimum-cost spanning trees are due to R. C. Prim and J. B. Kruskal, respectively.

An $O(e \log \log v)$ time spanning tree algorithm has been given by A. C. Yao.

The optimal randomized algorithm for minimum-cost spanning trees presented in this chapter appears in "A randomized linear-time algorithm for finding minimum spanning trees," by P. N. Klein and R. E. Tarjan, in *Proceedings of the 26th Annual Symposium on Theory of Computing*, 1994, pp. 9–15. See also "A randomized linear-time algorithm to find minimum spanning trees," by D. R. Karger, P. N. Klein, and R. E. Tarjan, *Journal of the ACM* 42, no. 2 (1995): 321–328.

Proof of Lemma 4.3 can be found in "Verification and sensitivity analysis of minimum spanning trees in linear time," by B. Dixon, M. Rauch, and R. E. Tarjan, *SIAM Journal on Computing* 21 (1992): 1184–1192, and in "A simple minimum spanning tree verification algorithm," by V. King, *Proceedings of the Workshop on Algorithms and Data Structures*, 1995.

A very nearly linear time algorithm for minimum-cost spanning trees appears in "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," by H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, *Combinatorica* 6 (1986): 109–122.

A linear time algorithm for minimum-cost spanning trees on a stronger model where the edge weights can be manipulated in their binary form is given in "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," by M. Fredman and D. E. Willard, in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990, pp. 719–725.

The greedy method developed here to optimally store programs on tapes was first devised for a machine scheduling problem. In this problem n jobs have to be scheduled on m processors. Job i takes t_i amount of time. The time at which a job finishes is the sum of the job times for all jobs preceding and including job i. The average finish time corresponds to the mean access time for programs on tapes. The $(m!)^{n/m}$ schedules referred to in Theorem 4.9 are known as SPT (shortest processing time) schedules. The rule to generate SPT schedules as well as the rule of Exercise 4 (Section 4.6) are due to W. E. Smith.

The greedy algorithm for generating optimal merge trees is due to D. Huffman.

For a given set $\{q_1, \ldots, q_n\}$ there are many sets of Huffman codes minimizing $\sum q_i d_i$. From amongst these code sets there is one that has minimum $\sum d_i$ and minimum max $\{d_i\}$. An algorithm to obtain this code set was given by E. S. Schwartz.

The shortest-path algorithm of the text is due to E. W. Dijkstra.

For planar graphs, the shortest-path problem can be solved in linear time as has been shown in "Faster shortest-path algorithms for planar graphs," by P. Klein, S. Rao, and M. Rauch, in *Proceedings of the ACM Symposium on Theory of Computing*, 1994.

The relationship between greedy methods and matroids is discussed in *Combinatorial Optimization*, by E. Lawler, Holt, Rinehart and Winston, 1976.

4.10 ADDITIONAL EXERCISES

1. [Coin changing] Let $A_n = \{a_1, a_2, \ldots, a_n\}$ be a finite set of distinct coin types (for example, $a_1 = 50¢$, $a_2 = 25¢$, $a_3 = 10¢$, and so on.) We can assume each a_i is an integer and $a_1 > a_2 > \cdots > a_n$. Each type is available in unlimited quantity. The coin-changing problem is to make up an exact amount C using a minimum total number of coins. C is an integer > 0.

(a) Show that if $a_n \neq 1$, then there exists a finite set of coin types and a C for which there is no solution to the coin-changing problem.

(b) Show that there is always a solution when $a_n = 1$.

(c) When $a_n = 1$, a greedy solution to the problem makes change by using the coin types in the order $a_1, a_2, \ldots, a_n$. When coin type a_i is being considered, as many coins of this type as possible are given. Write an algorithm based on this strategy. Show that this algorithm doesn't necessarily generate solutions that use the minimum total number of coins.

(d) Show that if $A_n = \{k^{n-1}, k^{n-2}, \ldots, k^0\}$ for some $k > 1$, then the greedy method of part (c) always yields solutions with a minimum number of coins.

2. [Set cover] You are given a family S of m sets $S_i, 1 \leq i \leq m$. Denote by $|A|$ the size of set A. Let $|S_i| = j_i$; that is, $S_i = \{s_1, s_2, \ldots, s_{j_i}\}$. A subset $T = \{T_1, T_2, \ldots, T_k\}$ of S is a family of sets such that for each $i, 1 \leq i \leq k$, $T_i = S_r$ for some $r, 1 \leq r \leq m$. The subset T is a *cover* of S iff $\cup T_i = \cup S_i$. The size of T, $|T|$, is the number of sets in T. A minimum cover of S is a cover of smallest size. Consider the following greedy strategy: build T iteratively, at the kth iteration $T = \{T_1, \ldots, T_{k-1}\}$, now add to T a set S_j from S that contains the largest number of elements not already in T, and stop when $\cup T_i = \cup S_i$.

(a) Assume that $\cup S_i = \{1, 2, \ldots, n\}$ and $m < n$. Using the strategy outlined above, write an algorithm to obtain set covers. How much time and space does your algorithm require?

(b) Show that the greedy strategy above doesn't necessarily obtain a minimum set cover.

(c) Suppose now that a minimum cover is defined to be one for which $\sum_{i=1}^{k} |T_i|$ is minimum. Does the above strategy always find a minimum cover?

3. [Node cover] Let $G = (V, E)$ be an undirected graph. A node cover of G is a subset U of the vertex set V such that every edge in E is incident to at least one vertex in U. A minimum node cover is one with the fewest number of vertices. Consider the following greedy algorithm for this problem:

```
1   Algorithm Cover(V, E)
2   {
3       U := ∅;
4       repeat
5       {
6           Let q be a vertex from V of maximum degree;
7           Add q to U; Eliminate q from V;
8           E := E − {(x, y) such that x = q or y = q};
9       } until (E = ∅); // U is the node cover.
10  }
```

Does this algorithm always generate a minimum node cover?

4. [Traveling salesperson] Let G be a directed graph with n vertices. Let $length(u, v)$ be the length of the edge $\langle u, v \rangle$. A path starting at a given vertex v_0, going through every other vertex exactly once, and finally returning to v_0 is called a *tour*. The length of a tour is the sum of the lengths of the edges on the path defining the tour. We are concerned with finding a tour of minimum length. A greedy way to construct such a tour is: let (P, v) represent the path so far constructed; it starts at v_0 and ends at v. Initially P is empty and $v = v_0$, if all vertices in G are on P, then include the edge $\langle v, v_0 \rangle$ and stop; otherwise include an edge $\langle v, w \rangle$ of minimum length among all edges from v to a vertex w not on P. Show that this greedy method doesn't necessarily generate a minimum-length tour.