

Introduction to **The Design &
Analysis of Algorithms**

3RD EDITION

Vice President and Editorial Director, ECS	<i>Marcia Horton</i>
Editor-in-Chief	<i>Michael Hirsch</i>
Acquisitions Editor	<i>Matt Goldstein</i>
Editorial Assistant	<i>Chelsea Bell</i>
Vice President, Marketing	<i>Patrice Jones</i>
Marketing Manager	<i>Yezan Alayan</i>
Senior Marketing Coordinator	<i>Kathryn Ferranti</i>
Marketing Assistant	<i>Emma Snider</i>
Vice President, Production	<i>Vince O'Brien</i>
Managing Editor	<i>Jeff Holcomb</i>
Production Project Manager	<i>Kayla Smith-Tarbox</i>
Senior Operations Supervisor	<i>Alan Fischer</i>
Manufacturing Buyer	<i>Lisa McDowell</i>
Art Director	<i>Anthony Gemmellaro</i>
Text Designer	<i>Sandra Rigney</i>
Cover Designer	<i>Anthony Gemmellaro</i>
Cover Illustration	<i>Jennifer Kohnke</i>
Media Editor	<i>Daniel Sandin</i>
Full-Service Project Management	<i>Windfall Software</i>
Composition	<i>Windfall Software, using ZzT_EX</i>
Printer/Binder	<i>Courier Westford</i>
Cover Printer	<i>Courier Westford</i>
Text Font	<i>Times Ten</i>

Copyright © 2012, 2007, 2003 Pearson Education, Inc., publishing as Addison-Wesley. All rights reserved. Printed in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to 201-236-3290.

This is the eBook of the printed book and may not include any media, Website access codes or print supplements that may come packaged with the bound book.

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Levitin, Anany.

Introduction to the design & analysis of algorithms / Anany Levitin. — 3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN-13: 978-0-13-231681-1

ISBN-10: 0-13-231681-1

1. Computer algorithms. I. Title. II. Title: Introduction to the design and analysis of algorithms.

QA76.9.A43L48 2012

005.1—dc23

2011027089

15 14 13 12 11—CRW—10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-231681-1

ISBN 13: 978-0-13-231681-1

Introduction to **The Design &
Analysis of Algorithms**

3RD EDITION

Anany Levitin

Villanova University

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

This page intentionally left blank

Brief Contents

New to the Third Edition	xvii
Preface	xix
1 Introduction	1
2 Fundamentals of the Analysis of Algorithm Efficiency	41
3 Brute Force and Exhaustive Search	97
4 Decrease-and-Conquer	131
5 Divide-and-Conquer	169
6 Transform-and-Conquer	201
7 Space and Time Trade-Offs	253
8 Dynamic Programming	283
9 Greedy Technique	315
10 Iterative Improvement	345
11 Limitations of Algorithm Power	387
12 Coping with the Limitations of Algorithm Power	423
Epilogue	471
APPENDIX A	
 Useful Formulas for the Analysis of Algorithms	475
APPENDIX B	
 Short Tutorial on Recurrence Relations	479
 References	493
 Hints to Exercises	503
 Index	547

This page intentionally left blank

Contents

New to the Third Edition	xvii
Preface	xix
1 Introduction	1
1.1 What Is an Algorithm?	3
Exercises 1.1	7
1.2 Fundamentals of Algorithmic Problem Solving	9
Understanding the Problem	9
Ascertaining the Capabilities of the Computational Device	9
Choosing between Exact and Approximate Problem Solving	11
Algorithm Design Techniques	11
Designing an Algorithm and Data Structures	12
Methods of Specifying an Algorithm	12
Proving an Algorithm's Correctness	13
Analyzing an Algorithm	14
Coding an Algorithm	15
Exercises 1.2	17
1.3 Important Problem Types	18
Sorting	19
Searching	20
String Processing	20
Graph Problems	21
Combinatorial Problems	21
Geometric Problems	22
Numerical Problems	22
Exercises 1.3	23

1.4 Fundamental Data Structures	25
Linear Data Structures	25
Graphs	28
Trees	31
Sets and Dictionaries	35
Exercises 1.4	37
Summary	38
2 Fundamentals of the Analysis of Algorithm Efficiency	41
2.1 The Analysis Framework	42
Measuring an Input's Size	43
Units for Measuring Running Time	44
Orders of Growth	45
Worst-Case, Best-Case, and Average-Case Efficiencies	47
Recapitulation of the Analysis Framework	50
Exercises 2.1	50
2.2 Asymptotic Notations and Basic Efficiency Classes	52
Informal Introduction	52
O -notation	53
Ω -notation	54
Θ -notation	55
Useful Property Involving the Asymptotic Notations	55
Using Limits for Comparing Orders of Growth	56
Basic Efficiency Classes	58
Exercises 2.2	58
2.3 Mathematical Analysis of Nonrecursive Algorithms	61
Exercises 2.3	67
2.4 Mathematical Analysis of Recursive Algorithms	70
Exercises 2.4	76
2.5 Example: Computing the nth Fibonacci Number	80
Exercises 2.5	83
2.6 Empirical Analysis of Algorithms	84
Exercises 2.6	89
2.7 Algorithm Visualization	91
Summary	94

3	Brute Force and Exhaustive Search	97
3.1	Selection Sort and Bubble Sort	98
	Selection Sort	98
	Bubble Sort	100
	Exercises 3.1	102
3.2	Sequential Search and Brute-Force String Matching	104
	Sequential Search	104
	Brute-Force String Matching	105
	Exercises 3.2	106
3.3	Closest-Pair and Convex-Hull Problems by Brute Force	108
	Closest-Pair Problem	108
	Convex-Hull Problem	109
	Exercises 3.3	113
3.4	Exhaustive Search	115
	Traveling Salesman Problem	116
	Knapsack Problem	116
	Assignment Problem	119
	Exercises 3.4	120
3.5	Depth-First Search and Breadth-First Search	122
	Depth-First Search	122
	Breadth-First Search	125
	Exercises 3.5	128
	Summary	130
4	Decrease-and-Conquer	131
4.1	Insertion Sort	134
	Exercises 4.1	136
4.2	Topological Sorting	138
	Exercises 4.2	142
4.3	Algorithms for Generating Combinatorial Objects	144
	Generating Permutations	144
	Generating Subsets	146
	Exercises 4.3	148

4.4 Decrease-by-a-Constant-Factor Algorithms	150
Binary Search	150
Fake-Coin Problem	152
Russian Peasant Multiplication	153
Josephus Problem	154
Exercises 4.4	156
4.5 Variable-Size-Decrease Algorithms	157
Computing a Median and the Selection Problem	158
Interpolation Search	161
Searching and Insertion in a Binary Search Tree	163
The Game of Nim	164
Exercises 4.5	166
Summary	167
5 Divide-and-Conquer	169
5.1 Mergesort	172
Exercises 5.1	174
5.2 Quicksort	176
Exercises 5.2	181
5.3 Binary Tree Traversals and Related Properties	182
Exercises 5.3	185
5.4 Multiplication of Large Integers and Strassen's Matrix Multiplication	186
Multiplication of Large Integers	187
Strassen's Matrix Multiplication	189
Exercises 5.4	191
5.5 The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer	192
The Closest-Pair Problem	192
Convex-Hull Problem	195
Exercises 5.5	197
Summary	198

6	Transform-and-Conquer	201
6.1	Presorting	202
	Exercises 6.1	205
6.2	Gaussian Elimination	208
	<i>LU</i> Decomposition	212
	Computing a Matrix Inverse	214
	Computing a Determinant	215
	Exercises 6.2	216
6.3	Balanced Search Trees	218
	AVL Trees	218
	2-3 Trees	223
	Exercises 6.3	225
6.4	Heaps and Heapsort	226
	Notion of the Heap	227
	Heapsort	231
	Exercises 6.4	233
6.5	Horner's Rule and Binary Exponentiation	234
	Horner's Rule	234
	Binary Exponentiation	236
	Exercises 6.5	239
6.6	Problem Reduction	240
	Computing the Least Common Multiple	241
	Counting Paths in a Graph	242
	Reduction of Optimization Problems	243
	Linear Programming	244
	Reduction to Graph Problems	246
	Exercises 6.6	248
	Summary	250
7	Space and Time Trade-Offs	253
7.1	Sorting by Counting	254
	Exercises 7.1	257
7.2	Input Enhancement in String Matching	258
	Horspool's Algorithm	259

Boyer-Moore Algorithm	263
Exercises 7.2	267
7.3 Hashing	269
Open Hashing (Separate Chaining)	270
Closed Hashing (Open Addressing)	272
Exercises 7.3	274
7.4 B-Trees	276
Exercises 7.4	279
Summary	280
8 Dynamic Programming	283
8.1 Three Basic Examples	285
Exercises 8.1	290
8.2 The Knapsack Problem and Memory Functions	292
Memory Functions	294
Exercises 8.2	296
8.3 Optimal Binary Search Trees	297
Exercises 8.3	303
8.4 Warshall's and Floyd's Algorithms	304
Warshall's Algorithm	304
Floyd's Algorithm for the All-Pairs Shortest-Paths Problem	308
Exercises 8.4	311
Summary	312
9 Greedy Technique	315
9.1 Prim's Algorithm	318
Exercises 9.1	322
9.2 Kruskal's Algorithm	325
Disjoint Subsets and Union-Find Algorithms	327
Exercises 9.2	331
9.3 Dijkstra's Algorithm	333
Exercises 9.3	337

9.4	Huffman Trees and Codes	338
	Exercises 9.4	342
	Summary	344
10	Iterative Improvement	345
10.1	The Simplex Method	346
	Geometric Interpretation of Linear Programming	347
	An Outline of the Simplex Method	351
	Further Notes on the Simplex Method	357
	Exercises 10.1	359
10.2	The Maximum-Flow Problem	361
	Exercises 10.2	371
10.3	Maximum Matching in Bipartite Graphs	372
	Exercises 10.3	378
10.4	The Stable Marriage Problem	380
	Exercises 10.4	383
	Summary	384
11	Limitations of Algorithm Power	387
11.1	Lower-Bound Arguments	388
	Trivial Lower Bounds	389
	Information-Theoretic Arguments	390
	Adversary Arguments	390
	Problem Reduction	391
	Exercises 11.1	393
11.2	Decision Trees	394
	Decision Trees for Sorting	395
	Decision Trees for Searching a Sorted Array	397
	Exercises 11.2	399
11.3	P , NP , and NP -Complete Problems	401
	P and NP Problems	402
	NP -Complete Problems	406
	Exercises 11.3	409

11.4 Challenges of Numerical Algorithms	412
Exercises 11.4	419
Summary	420

12 Coping with the Limitations of Algorithm Power **423**

12.1 Backtracking	424
n -Queens Problem	425
Hamiltonian Circuit Problem	426
Subset-Sum Problem	427
General Remarks	428
Exercises 12.1	430
12.2 Branch-and-Bound	432
Assignment Problem	433
Knapsack Problem	436
Traveling Salesman Problem	438
Exercises 12.2	440
12.3 Approximation Algorithms for <i>NP</i>-Hard Problems	441
Approximation Algorithms for the Traveling Salesman Problem	443
Approximation Algorithms for the Knapsack Problem	453
Exercises 12.3	457
12.4 Algorithms for Solving Nonlinear Equations	459
Bisection Method	460
Method of False Position	464
Newton's Method	464
Exercises 12.4	467
Summary	468

Epilogue **471**

APPENDIX A

Useful Formulas for the Analysis of Algorithms	475
Properties of Logarithms	475
Combinatorics	475
Important Summation Formulas	476
Sum Manipulation Rules	476

Approximation of a Sum by a Definite Integral	477
Floor and Ceiling Formulas	477
Miscellaneous	477

APPENDIX B

Short Tutorial on Recurrence Relations	479
Sequences and Recurrence Relations	479
Methods for Solving Recurrence Relations	480
Common Recurrence Types in Algorithm Analysis	485
 References	 493
 Hints to Exercises	 503
 Index	 547

This page intentionally left blank

1

Introduction

Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second, the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but it has been the algorithm that has made possible the modern world.

—David Berlinski, *The Advent of the Algorithm*, 2000

Why do you need to study algorithms? If you are going to be a computer professional, there are both practical and theoretical reasons to study algorithms. From a practical standpoint, you have to know a standard set of important algorithms from different areas of computing; in addition, you should be able to design new algorithms and analyze their efficiency. From the theoretical standpoint, the study of algorithms, sometimes called *algorithmics*, has come to be recognized as the cornerstone of computer science. David Harel, in his delightful book pointedly titled *Algorithmics: the Spirit of Computing*, put it as follows:

Algorithmics is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology. [Har92, p. 6]

But even if you are not a student in a computing-related program, there are compelling reasons to study algorithms. To put it bluntly, computer programs would not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems—not just answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved. Of course, the precision inherently imposed by algorithmic thinking limits the kinds of problems that can be solved with an algorithm. You will not find, for example, an algorithm for living a happy life or becoming rich and famous. On

the other hand, this required precision has an important educational advantage. Donald Knuth, one of the most prominent computer scientists in the history of algorithmics, put it as follows:

A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm . . . An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way. [Knu96, p. 9]

We take up the notion of algorithm in Section 1.1. As examples, we use three algorithms for the same problem: computing the greatest common divisor. There are several reasons for this choice. First, it deals with a problem familiar to everybody from their middle-school days. Second, it makes the important point that the same problem can often be solved by several algorithms. Quite typically, these algorithms differ in their idea, level of sophistication, and efficiency. Third, one of these algorithms deserves to be introduced first, both because of its age—it appeared in Euclid’s famous treatise more than two thousand years ago—and its enduring power and importance. Finally, investigation of these three algorithms leads to some general observations about several important properties of algorithms in general.

Section 1.2 deals with algorithmic problem solving. There we discuss several important issues related to the design and analysis of algorithms. The different aspects of algorithmic problem solving range from analysis of the problem and the means of expressing an algorithm to establishing its correctness and analyzing its efficiency. The section does not contain a magic recipe for designing an algorithm for an arbitrary problem. It is a well-established fact that such a recipe does not exist. Still, the material of Section 1.2 should be useful for organizing your work on designing and analyzing algorithms.

Section 1.3 is devoted to a few problem types that have proven to be particularly important to the study of algorithms and their application. In fact, there are textbooks (e.g., [Sed11]) organized around such problem types. I hold the view—shared by many others—that an organization based on algorithm design techniques is superior. In any case, it is very important to be aware of the principal problem types. Not only are they the most commonly encountered problem types in real-life applications, they are used throughout the book to demonstrate particular algorithm design techniques.

Section 1.4 contains a review of fundamental data structures. It is meant to serve as a reference rather than a deliberate discussion of this topic. If you need

a more detailed exposition, there is a wealth of good books on the subject, most of them tailored to a particular programming language.

1.1 What Is an Algorithm?

Although there is no universally agreed-on wording to describe this notion, there is general agreement about what the concept means:

An **algorithm** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

This definition can be illustrated by a simple diagram (Figure 1.1).

The reference to “instructions” in the definition implies that there is something or someone capable of understanding and following the instructions given. We call this a “computer,” keeping in mind that before the electronic computer was invented, the word “computer” meant a human being involved in performing numeric calculations. Nowadays, of course, “computers” are those ubiquitous electronic devices that have become indispensable in almost everything we do. Note, however, that although the majority of algorithms are indeed intended for eventual computer implementation, the notion of algorithm does not depend on such an assumption.

As examples illustrating the notion of the algorithm, we consider in this section three methods for solving the same problem: computing the greatest common divisor of two integers. These examples will help us to illustrate several important points:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

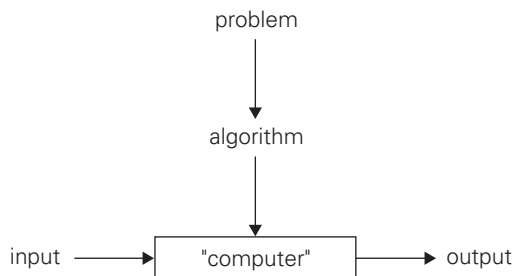


FIGURE 1.1 The notion of the algorithm.

- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Recall that the greatest common divisor of two nonnegative, not-both-zero integers m and n , denoted $\gcd(m, n)$, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero. Euclid of Alexandria (third century B.C.) outlined an algorithm for solving this problem in one of the volumes of his *Elements* most famous for its systematic exposition of geometry. In modern terms, **Euclid's algorithm** is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n),$$

where $m \bmod n$ is the remainder of the division of m by n , until $m \bmod n$ is equal to 0. Since $\gcd(m, 0) = m$ (why?), the last value of m is also the greatest common divisor of the initial m and n .

For example, $\gcd(60, 24)$ can be computed as follows:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

(If you are not impressed by this algorithm, try finding the greatest common divisor of larger numbers, such as those in Problem 6 in this section's exercises.)

Here is a more structured description of this algorithm:

Euclid's algorithm for computing $\gcd(m, n)$

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r .

Step 3 Assign the value of n to m and the value of r to n . Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM *Euclid*(m, n)

//Computes $\gcd(m, n)$ by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

How do we know that Euclid's algorithm eventually comes to a stop? This follows from the observation that the second integer of the pair gets smaller with each iteration and it cannot become negative. Indeed, the new value of n on the next iteration is $m \bmod n$, which is always smaller than n (why?). Hence, the value of the second integer eventually becomes 0, and the algorithm stops.

Just as with many other problems, there are several algorithms for computing the greatest common divisor. Let us look at the other two methods for this problem. The first is simply based on the definition of the greatest common divisor of m and n as the largest integer that divides both numbers evenly. Obviously, such a common divisor cannot be greater than the smaller of these numbers, which we will denote by $t = \min\{m, n\}$. So we can start by checking whether t divides both m and n : if it does, t is the answer; if it does not, we simply decrease t by 1 and try again. (How do we know that the process will eventually stop?) For example, for numbers 60 and 24, the algorithm will try first 24, then 23, and so on, until it reaches 12, where it stops.

Consecutive integer checking algorithm for computing $\gcd(m, n)$

Step 1 Assign the value of $\min\{m, n\}$ to t .

Step 2 Divide m by t . If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t . If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the set of an algorithm's inputs explicitly and carefully.

The third procedure for finding the greatest common divisor should be familiar to you from middle school.

Middle-school procedure for computing $\gcd(m, n)$

Step 1 Find the prime factors of m .

Step 2 Find the prime factors of n .

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n , respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$

$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

Nostalgia for the days when we learned this method should not prevent us from noting that the last procedure is much more complex and slower than Euclid's algorithm. (We will discuss methods for finding and comparing running times of algorithms in the next chapter.) In addition to inferior efficiency, the middle-school procedure does not qualify, in the form presented, as a legitimate algorithm. Why? Because the prime factorization steps are not defined unambiguously: they

require a list of prime numbers, and I strongly suspect that your middle-school math teacher did not explain how to obtain such a list. This is not a matter of unnecessary nitpicking. Unless this issue is resolved, we cannot, say, write a program implementing this procedure. Incidentally, Step 3 is also not defined clearly enough. Its ambiguity is much easier to rectify than that of the factorization steps, however. How would you find common elements in two sorted lists?

So, let us introduce a simple algorithm for generating consecutive primes not exceeding any given integer $n > 1$. It was probably invented in ancient Greece and is known as the *sieve of Eratosthenes* (ca. 200 B.C.). The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n . Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. (In this straightforward version, there is an overhead because some numbers, such as 6, are eliminated more than once.) No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

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

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.

What is the largest number p whose multiples can still remain on the list to make further iterations of the algorithm necessary? Before we answer this question, let us first note that if p is a number whose multiples are being eliminated on the current pass, then the first multiple we should consider is $p \cdot p$ because all its smaller multiples $2p, \dots, (p-1)p$ have been eliminated on earlier passes through the list. This observation helps to avoid eliminating the same number more than once. Obviously, $p \cdot p$ should not be greater than n , and therefore p cannot exceed \sqrt{n} rounded down (denoted $\lfloor \sqrt{n} \rfloor$ using the so-called *floor function*). We assume in the following pseudocode that there is a function available for computing $\lfloor \sqrt{n} \rfloor$; alternatively, we could check the inequality $p \cdot p \leq n$ as the loop continuation condition there.

ALGORITHM *Sieve*(n)

//Implements the sieve of Eratosthenes

//Input: A positive integer $n > 1$

//Output: Array L of all prime numbers less than or equal to n

```

for  $p \leftarrow 2$  to  $n$  do  $A[p] \leftarrow p$ 
for  $p \leftarrow 2$  to  $\lfloor \sqrt{n} \rfloor$  do //see note before pseudocode
    if  $A[p] \neq 0$  //p hasn't been eliminated on previous passes
         $j \leftarrow p * p$ 
        while  $j \leq n$  do
             $A[j] \leftarrow 0$  //mark element as eliminated
             $j \leftarrow j + p$ 
//copy the remaining elements of  $A$  to array  $L$  of the primes
 $i \leftarrow 0$ 
for  $p \leftarrow 2$  to  $n$  do
    if  $A[p] \neq 0$ 
         $L[i] \leftarrow A[p]$ 
         $i \leftarrow i + 1$ 
return  $L$ 

```

So now we can incorporate the sieve of Eratosthenes into the middle-school procedure to get a legitimate algorithm for computing the greatest common divisor of two positive integers. Note that special care needs to be exercised if one or both input numbers are equal to 1: because mathematicians do not consider 1 to be a prime number, strictly speaking, the method does not work for such inputs.

Before we leave this section, one more comment is in order. The examples considered in this section notwithstanding, the majority of algorithms in use today—even those that are implemented as computer programs—do not deal with mathematical problems. Look around for algorithms helping us through our daily routines, both professional and personal. May this ubiquity of algorithms in today's world strengthen your resolve to learn more about these fascinating engines of the information age.

Exercises 1.1

1. Do some research on al-Khorezmi (also al-Khwarizmi), the man from whose name the word “algorithm” is derived. In particular, you should learn what the origins of the words “algorithm” and “algebra” have in common.
2. Given that the official purpose of the U.S. patent system is the promotion of the “useful arts,” do you think algorithms are patentable in this country? Should they be?
3.
 - a. Write down driving directions for going from your school to your home with the precision required from an algorithm's description.
 - b. Write down a recipe for cooking your favorite dish with the precision required by an algorithm.
4. Design an algorithm for computing $\lfloor \sqrt{n} \rfloor$ for any positive integer n . Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations.

5. Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists 2, 5, 5, 5 and 2, 2, 3, 5, 5, 7, the output should be 2, 5, 5. What is the maximum number of comparisons your algorithm makes if the lengths of the two given lists are m and n , respectively?
6.
 - a. Find $\text{gcd}(31415, 14142)$ by applying Euclid's algorithm.
 - b. Estimate how many times faster it will be to find $\text{gcd}(31415, 14142)$ by Euclid's algorithm compared with the algorithm based on checking consecutive integers from $\min\{m, n\}$ down to $\text{gcd}(m, n)$.
7. Prove the equality $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$ for every pair of positive integers m and n .
8. What does Euclid's algorithm do for a pair of integers in which the first is smaller than the second? What is the maximum number of times this can happen during the algorithm's execution on such an input?
9.
 - a. What is the minimum number of divisions made by Euclid's algorithm among all inputs $1 \leq m, n \leq 10$?
 - b. What is the maximum number of divisions made by Euclid's algorithm among all inputs $1 \leq m, n \leq 10$?
10.
 - a. Euclid's algorithm, as presented in Euclid's treatise, uses subtractions rather than integer divisions. Write pseudocode for this version of Euclid's algorithm.
 - b. *Euclid's game* (see [Bog]) starts with two unequal positive integers on the board. Two players move in turn. On each move, a player has to write on the board a positive number equal to the difference of two numbers already on the board; this number must be new, i.e., different from all the numbers already on the board. The player who cannot move loses the game. Should you choose to move first or second in this game?
11. The **extended Euclid's algorithm** determines not only the greatest common divisor d of two positive integers m and n but also integers (not necessarily positive) x and y , such that $mx + ny = d$.
 - a. Look up a description of the extended Euclid's algorithm (see, e.g., [KnuI, p. 13]) and implement it in the language of your choice.
 - b. Modify your program to find integer solutions to the Diophantine equation $ax + by = c$ with any set of integer coefficients a , b , and c .
12. *Locker doors* There are n lockers in a hallway, numbered sequentially from 1 to n . Initially, all the locker doors are closed. You make n passes by the lockers, each time starting with locker #1. On the i th pass, $i = 1, 2, \dots, n$, you toggle the door of every i th locker: if the door is closed, you open it; if it is open, you close it. After the last pass, which locker doors are open and which are closed? How many of them are open?



1.2 Fundamentals of Algorithmic Problem Solving

Let us start by reiterating an important point made in the introduction to this chapter:

We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics, whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties.

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often. We review them in the next section. If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms. But often you will not find a readily available algorithm and will have to design your own. The sequence of steps outlined in this section should help you in this exciting but not always easy task.

An input to an algorithm specifies an *instance* of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle. (As an example, recall the variations in the set of instances for the three greatest common divisor algorithms discussed in the previous section.) If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some “boundary” value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Do not skimp on this first step of the algorithmic problem-solving process; otherwise, you will run the risk of unnecessary rework.

Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of

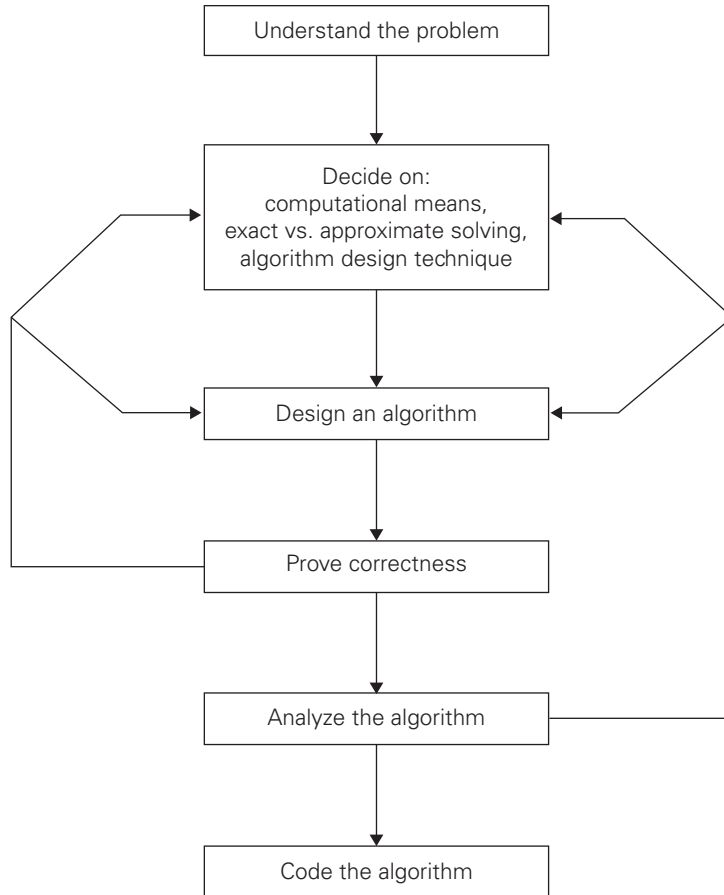


FIGURE 1.2 Algorithm design and analysis process.

algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903–1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called *random-access machine (RAM)*. Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called *sequential algorithms*.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called *parallel algorithms*. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Should you worry about the speed and amount of memory of a computer at your disposal? If you are designing an algorithm as a scientific exercise, the answer is a qualified no. As you will see in Section 2.1, most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer. If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve. Even the “slow” computers of today are almost unimaginably fast. Consequently, in many situations you need not worry about a computer being too slow for the task. There are important problems, however, that are very complex by their nature, or have to process huge volumes of data, or deal with applications where the time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an *exact algorithm*; in the latter case, an algorithm is called an *approximation algorithm*. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem’s intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices; you will see examples of such difficult problems in Chapters 3, 11, and 12. Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem? This is the main question this book seeks to answer by teaching you several general design techniques.

What is an algorithm design technique?

An *algorithm design technique* (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Check this book’s table of contents and you will see that a majority of its chapters are devoted to individual design techniques. They distill a few key ideas that have proven to be useful in designing algorithms. Learning these techniques is of utmost importance for the following reasons.

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning

to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

Second, algorithms are the cornerstone of computer science. Every science is interested in classifying its principal subject, and computer science is no exception. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques. Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer. With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Of course, one should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes introduced in Section 1.1 would run longer if we used a linked list instead of an array in its implementation (why?). Also note that some of the algorithm design techniques discussed in Chapters 6 and 7 depend intimately on structuring or restructuring data specifying a problem's instance. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures = Programs* [Wir76]. In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms. We review basic data structures in Section 1.4.

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, Euclid's algorithm is described in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

Pseudocode is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than natural language, and its

usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors with a need to design their own “dialects.” Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

This book’s dialect was selected to cause minimal difficulty for a reader. For the sake of simplicity, we omit declarations of variables and use indentation to show the scope of such statements as **for**, **if**, and **while**. As you saw in the previous section, we use an arrow “ \leftarrow ” for the assignment operation and two slashes “//” for comments.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a *flowchart*, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm’s steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm’s description—be it in a natural language or pseudocode—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm’s implementation.

Proving an Algorithm’s Correctness

Once an algorithm has been specified, you have to prove its *correctness*. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid’s algorithm for computing the greatest common divisor stems from the correctness of the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 7 in Exercises 1.1), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm’s iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm’s performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm’s correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit. You can find examples of such investigations in Chapter 12.

Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is *efficiency*. In fact, there are two kinds of algorithm efficiency: *time efficiency*, indicating how fast the algorithm runs, and *space efficiency*, indicating how much extra memory it uses. A general framework and specific techniques for analyzing an algorithm's efficiency appear in Chapter 2.

Another desirable characteristic of an algorithm is *simplicity*. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that Euclid's algorithm is simpler than the middle-school procedure for computing $\text{gcd}(m, n)$, but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm. Still, simplicity is an important algorithm characteristic to strive for. Why? Because simpler algorithms are easier to understand and easier to program; consequently, the resulting programs usually contain fewer bugs. There is also the undeniable aesthetic appeal of simplicity. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

Yet another desirable characteristic of an algorithm is *generality*. There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts. On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms. Consider, for example, the problem of determining whether two integers are relatively prime, i.e., whether their only common divisor is equal to 1. It is easier to design an algorithm for a more general problem of computing the greatest common divisor of two integers and, to solve the former problem, check whether the gcd is 1 or not. There are situations, however, where designing a more general algorithm is unnecessary or difficult or even impossible. For example, it is unnecessary to sort a list of n numbers to find its median, which is its $\lceil n/2 \rceil$ th smallest element. To give another example, the standard formula for roots of a quadratic equation cannot be generalized to handle polynomials of arbitrary degrees.

As to the set of inputs, your main concern should be designing an algorithm that can handle a set of inputs that is natural for the problem at hand. For example, excluding integers equal to 1 as possible inputs for a greatest common divisor algorithm would be quite unnatural. On the other hand, although the standard formula for the roots of a quadratic equation holds for complex coefficients, we would normally not implement it on this level of generality unless this capability is explicitly required.

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm. In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions. Recall the three different algorithms in the previous section for computing the greatest common divisor: generally, you should not expect to get the best algorithm on the first try. At the very least, you should try to fine-tune the algorithm you

already have. For example, we made several improvements in our implementation of the sieve of Eratosthenes compared with its initial outline in Section 1.1. (Can you identify them?) You will do well if you keep in mind the following observation of Antoine de Saint-Exupéry, the French writer, pilot, and aircraft designer: “A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away.”¹

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct. They have developed special techniques for doing such proofs (see [Gri81]), but the power of these techniques of formal verification is limited so far to very small programs.

As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn. Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

Also note that throughout the book, we assume that inputs to algorithms belong to the specified sets and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm’s power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode. Still, you need to be aware of such standard tricks as computing a loop’s invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, and so on. (See [Ker99] and [Ben00] for a good discussion of code tuning and other issues related to algorithm programming.) Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.

1. I found this call for design simplicity in an essay collection by Jon Bentley [Ben00]; the essays deal with a variety of issues in algorithm design and implementation and are justifiably titled *Programming Pearls*. I wholeheartedly recommend the writings of both Jon Bentley and Antoine de Saint-Exupéry.

A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. Such an analysis is based on timing the program on several inputs and then analyzing the results obtained. We discuss the advantages and disadvantages of this approach to analyzing algorithms in Section 2.6.

In conclusion, let us emphasize again the main lesson of the process depicted in Figure 1.2:

As a rule, a good algorithm is a result of repeated effort and rework.

Even if you have been fortunate enough to get an algorithmic idea that seems perfect, you should still try to see whether it can be improved.

Actually, this is good news since it makes the ultimate result so much more enjoyable. (Yes, I did think of naming this book *The Joy of Algorithms*.) On the other hand, how does one know when to stop? In the real world, more often than not a project's schedule or the impatience of your boss will stop you. And so it should be: perfection is expensive and in fact not always called for. Designing an algorithm is an engineering-like activity that calls for compromises among competing goals under the constraints of available resources, with the designer's time being one of the resources.

In the academic world, the question leads to an interesting but usually difficult investigation of an algorithm's *optimality*. Actually, this question is not about the efficiency of an algorithm but about the complexity of the problem it solves: What is the minimum amount of effort *any* algorithm will need to exert to solve the problem? For some problems, the answer to this question is known. For example, any algorithm that sorts an array by comparing values of its elements needs about $n \log_2 n$ comparisons for some arrays of size n (see Section 11.2). But for many seemingly easy problems such as integer multiplication, computer scientists do not yet have a final answer.

Another important issue of algorithmic problem solving is the question of whether or not every problem can be solved by an algorithm. We are not talking here about problems that do not have a solution, such as finding real roots of a quadratic equation with a negative discriminant. For such cases, an output indicating that the problem does not have a solution is all we can and should expect from an algorithm. Nor are we talking about ambiguously stated problems. Even some unambiguous problems that must have a simple yes or no answer are “undecidable,” i.e., unsolvable by any algorithm. An important example of such a problem appears in Section 11.3. Fortunately, a vast majority of problems in practical computing *can* be solved by an algorithm.

Before leaving this section, let us be sure that you do not have the misconception—possibly caused by the somewhat mechanical nature of the diagram of Figure 1.2—that designing an algorithm is a dull activity. There is nothing further from the truth: inventing (or discovering?) algorithms is a very creative and rewarding process. This book is designed to convince you that this is the case.

Exercises 1.2



1. *Old World puzzle* A peasant finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)



2. *New World puzzle* There are four people who want to cross a rickety bridge; they all begin on the same side. You have 17 minutes to get them all across to the other side. It is night, and they have one flashlight. A maximum of two people can cross the bridge at one time. Any party that crosses, either one or two people, must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, for example. Person 1 takes 1 minute to cross the bridge, person 2 takes 2 minutes, person 3 takes 5 minutes, and person 4 takes 10 minutes. A pair must walk together at the rate of the slower person's pace. (Note: According to a rumor on the Internet, interviewers at a well-known software company located near Seattle have given this problem to interviewees.)
3. Which of the following formulas can be considered an algorithm for computing the area of a triangle whose side lengths are given positive numbers a , b , and c ?
 - a. $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = (a+b+c)/2$
 - b. $S = \frac{1}{2}bc \sin A$, where A is the angle between sides b and c
 - c. $S = \frac{1}{2}ah_a$, where h_a is the height to base a
4. Write pseudocode for an algorithm for finding real roots of equation $ax^2 + bx + c = 0$ for arbitrary real coefficients a , b , and c . (You may assume the availability of the square root function $\text{sqrt}(x)$.)
5. Describe the standard algorithm for finding the binary representation of a positive decimal integer
 - a. in English.
 - b. in pseudocode.
6. Describe the algorithm used by your favorite ATM machine in dispensing cash. (You may give your description in either English or pseudocode, whichever you find more convenient.)
7.
 - a. Can the problem of computing the number π be solved exactly?
 - b. How many instances does this problem have?
 - c. Look up an algorithm for this problem on the Internet.

8. Give an example of a problem other than computing the greatest common divisor for which you know more than one algorithm. Which of them is simpler? Which is more efficient?
9. Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

ALGORITHM *MinDistance*($A[0..n - 1]$)

//Input: Array $A[0..n - 1]$ of numbers

//Output: Minimum distance between two of its elements

$dmin \leftarrow \infty$

for $i \leftarrow 0$ **to** $n - 1$ **do**

for $j \leftarrow 0$ **to** $n - 1$ **do**

if $i \neq j$ **and** $|A[i] - A[j]| < dmin$

$dmin \leftarrow |A[i] - A[j]|$

return $dmin$

Make as many improvements as you can in this algorithmic solution to the problem. If you need to, you may change the algorithm altogether; if not, improve the implementation given.

10. One of the most influential books on problem solving, titled *How To Solve It* [Pol57], was written by the Hungarian-American mathematician George Pólya (1887–1985). Pólya summarized his ideas in a four-point summary. Find this summary on the Internet or, better yet, in his book, and compare it with the plan outlined in Section 1.2. What do they have in common? How are they different?

1.3 Important Problem Types

In the limitless sea of problems one encounters in computing, there are a few areas that have attracted particular attention from researchers. By and large, their interest has been driven either by the problem's practical importance or by some specific characteristics making the problem an interesting research subject; fortunately, these two motivating forces reinforce each other in most cases.

In this section, we are going to introduce the most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

These problems are used in subsequent chapters of the book to illustrate different algorithm design techniques and methods of algorithm analysis.

Sorting

The *sorting problem* is to rearrange the items of a given list in nondecreasing order. Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering. (Mathematicians would say that there must exist a relation of total ordering.) As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees. In the case of records, we need to choose a piece of information to guide sorting. For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a *key*. Computer scientists often talk about sorting a list of keys even when the list's items are not records but, say, just integers.

Why would we want a sorted list? To begin with, a sorted list can be a required output of a task such as ranking Internet search results or ranking students by their GPA scores. Further, sorting makes many questions about the list easier to answer. The most important of them is searching: it is why dictionaries, telephone books, class lists, and so on are sorted. You will see other examples of the usefulness of list presorting in Section 6.1. In a similar vein, sorting is used as an auxiliary step in several important algorithms in other areas, e.g., geometric algorithms and data compression. The greedy approach—an important algorithm design technique discussed later in the book—requires a sorted input.

By now, computer scientists have discovered dozens of different sorting algorithms. In fact, inventing a new sorting algorithm has been likened to designing the proverbial mousetrap. And I am happy to report that the hunt for a better sorting mousetrap continues. This perseverance is admirable in view of the following facts. On the one hand, there are a few good sorting algorithms that sort an arbitrary array of size n using about $n \log_2 n$ comparisons. On the other hand, no algorithm that sorts by key comparisons (as opposed to, say, comparing small pieces of keys) can do substantially better than that.

There is a reason for this embarrassment of algorithmic riches in the land of sorting. Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations. Some of the algorithms are simple but relatively slow, while others are faster but more complex; some work better on randomly ordered inputs, while others do better on almost-sorted lists; some are suitable only for lists residing in the fast memory, while others can be adapted for sorting large files stored on a disk; and so on.

Two properties of sorting algorithms deserve special mention. A sorting algorithm is called *stable* if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions i and j where $i < j$, then in the sorted list they have to be in positions i' and j' ,

respectively, such that $i' < j'$. This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically. Generally speaking, algorithms that can exchange keys located far apart are not stable, but they usually work faster; you will see how this general comment applies to important sorting algorithms later in the book.

The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be *in-place* if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in-place and those that are not.

Searching

The *searching problem* deals with finding a given value, called a *search key*, in a given set (or a multiset, which permits several elements to have the same value). There are plenty of searching algorithms to choose from. They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching. The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.

For searching, too, there is no single algorithm that fits all situations best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on. Unlike with sorting algorithms, there is no stability problem, but different issues arise. Specifically, in applications where the underlying data may change frequently relative to the number of searches, searching has to be considered in conjunction with two other operations: an addition to and deletion from the data set of an item. In such situations, data structures and algorithms should be chosen to strike a balance among the requirements of each operation. Also, organizing very large data sets for efficient searching poses special challenges with important implications for real-world applications.

String Processing

In recent decades, the rapid proliferation of applications dealing with nonnumerical data has intensified the interest of researchers and computing practitioners in string-handling algorithms. A *string* is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}. It should be pointed out, however, that string-processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it **string matching**. Several algorithms that exploit the special nature of this type of searching have been invented. We introduce one very simple algorithm in Chapter 3 and discuss two algorithms based on a remarkable idea by R. Boyer and J. Moore in Chapter 7.

Graph Problems

One of the oldest and most interesting areas in algorithmics is graph algorithms. Informally, a **graph** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges. (A more formal definition is given in the next section.) Graphs are an interesting subject to study, for both theoretical and practical reasons. Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games. Studying different technical and social aspects of the Internet in particular is one of the active areas of current research involving computer scientists, economists, and social scientists (see, e.g., [Eas10]).

Basic graph algorithms include graph-traversal algorithms (how can one reach all the points in a network?), shortest-path algorithms (what is the best route between two cities?), and topological sorting for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?). Fortunately, these algorithms can be considered illustrations of general design techniques; accordingly, you will find them in corresponding chapters of the book.

Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem. The **traveling salesman problem (TSP)** is the problem of finding the shortest tour through n cities that visits every city exactly once. In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray crystallography, and genetic engineering. The **graph-coloring problem** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color. This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule.

Combinatorial Problems

From a more abstract perspective, the traveling salesman problem and the graph-coloring problem are examples of **combinatorial problems**. These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints. A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost.

Generally speaking, combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint. Their difficulty stems from the following facts. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time. Moreover, most computer scientists believe that such algorithms do not exist. This conjecture has been neither proved nor disproved, and it remains the most important unresolved issue in theoretical computer science. We discuss this topic in more detail in Section 11.3.

Some combinatorial problems can be solved by efficient algorithms, but they should be considered fortunate exceptions to the rule. The shortest-path problem mentioned earlier is among such exceptions.

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. The ancient Greeks were very much interested in developing procedures (they did not call them algorithms, of course) for solving a variety of geometric problems, including problems of constructing simple geometric shapes—triangles, circles, and so on—with an unmarked ruler and a compass. Then, for about 2000 years, intense interest in geometric algorithms disappeared, to be resurrected in the age of computers—no more rulers and compasses, just bits, bytes, and good old human ingenuity. Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography.

We will discuss algorithms for only two classic problems of computational geometry: the closest-pair problem and the convex-hull problem. The *closest-pair problem* is self-explanatory: given n points in the plane, find the closest pair among them. The *convex-hull problem* asks to find the smallest convex polygon that would include all the points of a given set. If you are interested in other geometric algorithms, you will find a wealth of material in such specialized monographs as [deB10], [ORo98], and [Pre85].

Numerical Problems

Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately. Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately. Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off

error to a point where it can drastically distort an output produced by a seemingly sound algorithm.

Many sophisticated algorithms have been developed over the years in this area, and they continue to play a critical role in many scientific and engineering applications. But in the last 30 years or so, the computing industry has shifted its focus to business applications. These new applications require primarily algorithms for information storage, retrieval, transportation through networks, and presentation to users. As a result of this revolutionary change, numerical analysis has lost its formerly dominating position in both industry and computer science programs. Still, it is important for any computer-literate person to have at least a rudimentary idea about numerical algorithms. We discuss several classical numerical algorithms in Sections 6.2, 11.4, and 12.4.

Exercises 1.3

1. Consider the algorithm for the sorting problem that sorts an array by counting, for each of its elements, the number of smaller elements and then uses this information to put the element in its appropriate position in the sorted array:

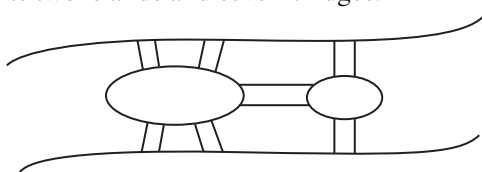
ALGORITHM *ComparisonCountingSort*($A[0..n-1]$)

```
//Sorts an array by comparison counting
//Input: Array  $A[0..n-1]$  of orderable values
//Output: Array  $S[0..n-1]$  of  $A$ 's elements sorted
//  in nondecreasing order
for  $i \leftarrow 0$  to  $n-1$  do
     $Count[i] \leftarrow 0$ 
for  $i \leftarrow 0$  to  $n-2$  do
    for  $j \leftarrow i+1$  to  $n-1$  do
        if  $A[i] < A[j]$ 
             $Count[j] \leftarrow Count[j] + 1$ 
        else  $Count[i] \leftarrow Count[i] + 1$ 
for  $i \leftarrow 0$  to  $n-1$  do
     $S[Count[i]] \leftarrow A[i]$ 
return  $S$ 
```

- a. Apply this algorithm to sorting the list 60, 35, 81, 98, 14, 47.
 - b. Is this algorithm stable?
 - c. Is it in-place?
2. Name the algorithms for the searching problem that you already know. Give a good succinct description of each algorithm in English. If you know no such algorithms, use this opportunity to design one.
 3. Design a simple algorithm for the string-matching problem.



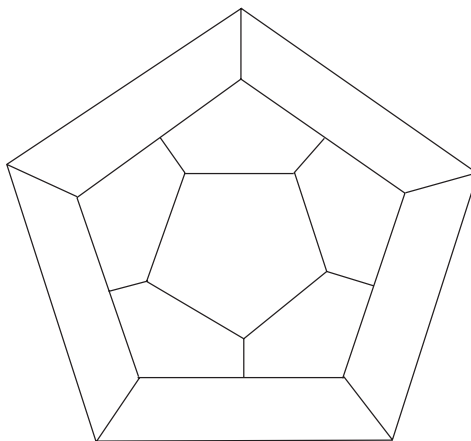
4. *Königsberg bridges* The Königsberg bridge puzzle is universally accepted as the problem that gave birth to graph theory. It was solved by the great Swiss-born mathematician Leonhard Euler (1707–1783). The problem asked whether one could, in a single stroll, cross all seven bridges of the city of Königsberg exactly once and return to a starting point. Following is a sketch of the river with its two islands and seven bridges:



- a. State the problem as a graph problem.
- b. Does this problem have a solution? If you believe it does, draw such a stroll; if you believe it does not, explain why and indicate the smallest number of new bridges that would be required to make such a stroll possible.



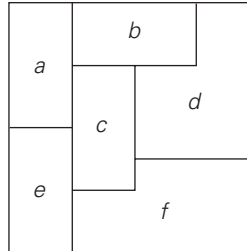
5. *Icosian Game* A century after Euler's discovery (see Problem 4), another famous puzzle—this one invented by the renowned Irish mathematician Sir William Hamilton (1805–1865)—was presented to the world under the name of the Icosian Game. The game's board was a circular wooden board on which the following graph was carved:



Find a **Hamiltonian circuit**—a path that visits all the graph's vertices exactly once before returning to the starting vertex—for this graph.

6. Consider the following problem: Design an algorithm to determine the best route for a subway passenger to take from one designated station to another in a well-developed subway system similar to those in such cities as Washington, D.C., and London, UK.

- a. The problem's statement is somewhat vague, which is typical of real-life problems. In particular, what reasonable criterion can be used for defining the "best" route?
 - b. How would you model this problem by a graph?
7. a. Rephrase the traveling-salesman problem in combinatorial object terms.
 - b. Rephrase the graph-coloring problem in combinatorial object terms.
8. Consider the following map:



- a. Explain how we can use the graph-coloring problem to color the map so that no two neighboring regions are colored the same.
 - b. Use your answer to part (a) to color the map with the smallest number of colors.
9. Design an algorithm for the following problem: Given a set of n points in the Cartesian plane, determine whether all of them lie on the same circumference.
 10. Write a program that reads as its inputs the (x, y) coordinates of the endpoints of two line segments P_1Q_1 and P_2Q_2 and determines whether the segments have a common point.

1.4 Fundamental Data Structures

Since the vast majority of algorithms of interest operate on data, particular ways of organizing data play a critical role in the design and analysis of algorithms. A **data structure** can be defined as a particular scheme of organizing related data items. The nature of the data items is dictated by the problem at hand; they can range from elementary data types (e.g., integers or characters) to data structures (e.g., a one-dimensional array of one-dimensional arrays is often used for implementing matrices). There are a few data structures that have proved to be particularly important for computer algorithms. Since you are undoubtedly familiar with most if not all of them, just a quick review is provided here.

Linear Data Structures

The two most important elementary data structures are the array and the linked list. A (one-dimensional) **array** is a sequence of n items of the same data type that

are stored contiguously in computer memory and made accessible by specifying a value of the array's *index* (Figure 1.3).

In the majority of cases, the index is an integer either between 0 and $n - 1$ (as shown in Figure 1.3) or between 1 and n . Some computer languages allow an array index to range between any two integer bounds *low* and *high*, and some even permit nonnumerical indices to specify, for example, data items corresponding to the 12 months of the year by the month names.

Each and every element of an array can be accessed in the same constant amount of time regardless of where in the array the element in question is located. This feature positively distinguishes arrays from linked lists, discussed below.

Arrays are used for implementing a variety of other data structures. Prominent among them is the *string*, a sequence of characters from an alphabet terminated by a special character indicating the string's end. Strings composed of zeros and ones are called *binary strings* or *bit strings*. Strings are indispensable for processing textual data, defining computer languages and compiling programs written in them, and studying abstract computational models. Operations we usually perform on strings differ from those we typically perform on other arrays (say, arrays of numbers). They include computing the string length, comparing two strings to determine which one precedes the other in *lexicographic* (i.e., alphabetical) *order*, and concatenating two strings (forming one string from two given strings by appending the second to the end of the first).

A *linked list* is a sequence of zero or more elements called *nodes*, each containing two kinds of information: some data and one or more links called *pointers* to other nodes of the linked list. (A special pointer called “null” is used to indicate the absence of a node's successor.) In a *singly linked list*, each node except the last one contains a single pointer to the next element (Figure 1.4).

To access a particular node of a linked list, one starts with the list's first node and traverses the pointer chain until the particular node is reached. Thus, the time needed to access an element of a singly linked list, unlike that of an array, depends on where in the list the element is located. On the positive side, linked lists do



FIGURE 1.3 Array of n elements.

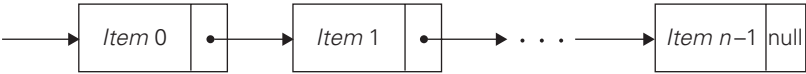


FIGURE 1.4 Singly linked list of n elements.

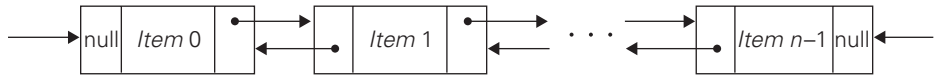


FIGURE 1.5 Doubly linked list of n elements.

not require any preliminary reservation of the computer memory, and insertions and deletions can be made quite efficiently in a linked list by reconnecting a few appropriate pointers.

We can exploit flexibility of the linked list structure in a variety of ways. For example, it is often convenient to start a linked list with a special node called the **header**. This node may contain information about the linked list itself, such as its current length; it may also contain, in addition to a pointer to the first element, a pointer to the linked list's last element.

Another extension is the structure called the **doubly linked list**, in which every node, except the first and the last, contains pointers to both its successor and its predecessor (Figure 1.5).

The array and linked list are two principal choices in representing a more abstract data structure called a linear list or simply a list. A **list** is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element.

Two special types of lists, stacks and queues, are particularly important. A **stack** is a list in which insertions and deletions can be done only at the end. This end is called the **top** because a stack is usually visualized not horizontally but vertically—akin to a stack of plates whose “operations” it mimics very closely. As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in a “last-in–first-out” (LIFO) fashion—exactly like a stack of plates if we can add or remove a plate only from the top. Stacks have a multitude of applications; in particular, they are indispensable for implementing recursive algorithms.

A **queue**, on the other hand, is a list from which elements are deleted from one end of the structure, called the **front** (this operation is called **dequeue**), and new elements are added to the other end, called the **rear** (this operation is called **enqueue**). Consequently, a queue operates in a “first-in–first-out” (FIFO) fashion—akin to a queue of customers served by a single teller in a bank. Queues also have many important applications, including several algorithms for graph problems.

Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a priority queue. A **priority queue** is a collection of data items from a totally ordered universe (most often,

integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element. Of course, a priority queue must be implemented so that the last two operations yield another priority queue. Straightforward implementations of this data structure can be based on either an array or a sorted array, but neither of these options yields the most efficient solution possible. A better implementation of a priority queue is based on an ingenious data structure called the **heap**. We discuss heaps and an important sorting algorithm based on them in Section 6.4.

Graphs

As we mentioned in the previous section, a graph is informally thought of as a collection of points in the plane called “vertices” or “nodes,” some of them connected by line segments called “edges” or “arcs.” Formally, a **graph** $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite nonempty set V of items called **vertices** and a set E of pairs of these items called **edges**. If these pairs of vertices are unordered, i.e., a pair of vertices (u, v) is the same as the pair (v, u) , we say that the vertices u and v are **adjacent** to each other and that they are connected by the **undirected edge** (u, v) . We call the vertices u and v **endpoints** of the edge (u, v) and say that u and v are **incident** to this edge; we also say that the edge (u, v) is incident to its endpoints u and v . A graph G is called **undirected** if every edge in it is undirected.

If a pair of vertices (u, v) is not the same as the pair (v, u) , we say that the edge (u, v) is **directed** from the vertex u , called the edge’s **tail**, to the vertex v , called the edge’s **head**. We also say that the edge (u, v) leaves u and enters v . A graph whose every edge is directed is called **directed**. Directed graphs are also called **digraphs**.

It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 1.6). The graph depicted in Figure 1.6a has six vertices and seven undirected edges:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$

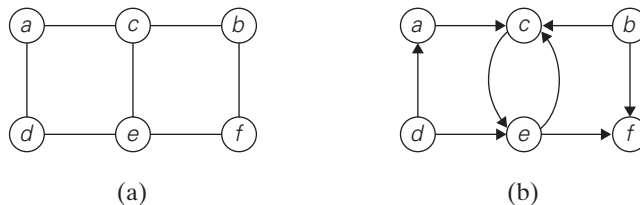


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Our definition of a graph does not forbid **loops**, or edges connecting vertices to themselves. Unless explicitly stated otherwise, we will consider graphs without loops. Since our definition disallows multiple edges between the same vertices of an undirected graph, we have the following inequality for the number of edges $|E|$ possible in an undirected graph with $|V|$ vertices and no loops:

$$0 \leq |E| \leq |V|(|V| - 1)/2.$$

(We get the largest number of edges in a graph if there is an edge connecting each of its $|V|$ vertices with all $|V| - 1$ other vertices. We have to divide product $|V|(|V| - 1)$ by 2, however, because it includes every edge twice.)

A graph with every pair of its vertices connected by an edge is called **complete**. A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$. A graph with relatively few possible edges missing is called **dense**; a graph with few edges relative to the number of its vertices is called **sparse**. Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

Graph Representations Graphs for computer algorithms are usually represented in one of two ways: the adjacency matrix and adjacency lists. The **adjacency matrix** of a graph with n vertices is an $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i th row and the j th column is equal to 1 if there is an edge from the i th vertex to the j th vertex, and equal to 0 if there is no such edge. For example, the adjacency matrix for the graph of Figure 1.6a is given in Figure 1.7a.

Note that the adjacency matrix of an undirected graph is always symmetric, i.e., $A[i, j] = A[j, i]$ for every $0 \leq i, j \leq n - 1$ (why?).

The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge). Usually, such lists start with a header identifying a vertex for which the list is compiled. For example, Figure 1.7b represents the graph in Figure 1.6a via its adjacency lists. To put it another way,

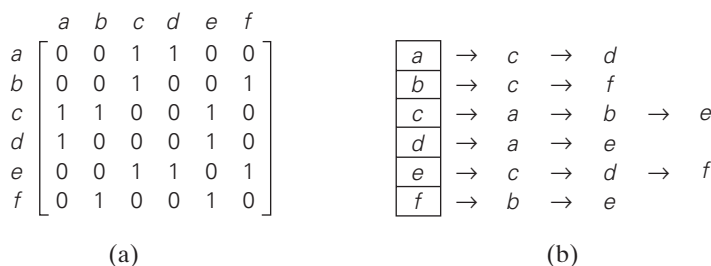


FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a.

adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1's.

If a graph is sparse, the adjacency list representation may use less space than the corresponding adjacency matrix despite the extra storage consumed by pointers of the linked lists; the situation is exactly opposite for dense graphs. In general, which of the two representations is more convenient depends on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (sparse or dense).

Weighted Graphs A **weighted graph** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called **weights** or **costs**. An interest in such graphs is motivated by numerous real-world applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.

Both principal representations of a graph can be easily adopted to accommodate weighted graphs. If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will simply contain the weight of the edge from the i th to the j th vertex if there is such an edge and a special symbol, e.g., ∞ , if there is no such edge. Such a matrix is called the **weight matrix** or **cost matrix**. This approach is illustrated in Figure 1.8b for the weighted graph in Figure 1.8a. (For some applications, it is more convenient to put 0's on the main diagonal of the adjacency matrix.) Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the corresponding edge (Figure 1.8c).

Paths and Cycles Among the many properties of graphs, two are important for a great number of applications: **connectivity** and **acyclicity**. Both are based on the notion of a path. A **path** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v . If all vertices of a path are distinct, the path is said to be **simple**. The **length** of a path is the total number of vertices in the vertex sequence defining the path minus 1, which is the same as the number of edges in the path. For example, a, c, b, f is a simple path of length 3 from a to f in the graph in Figure 1.6a, whereas a, c, e, c, b, f is a path (not simple) of length 5 from a to f .

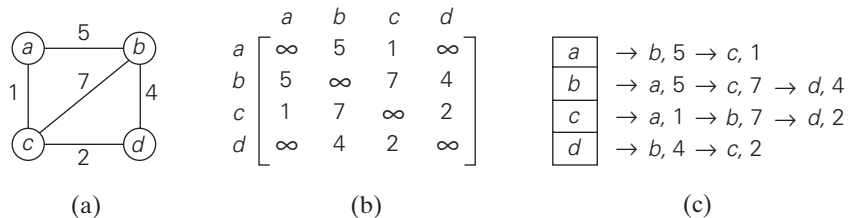


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

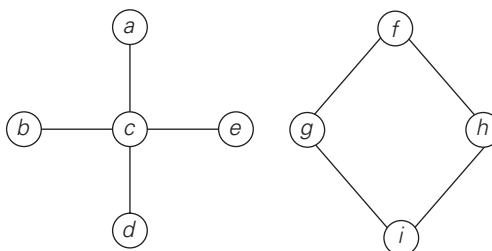


FIGURE 1.9 Graph that is not connected.

In the case of a directed graph, we are usually interested in directed paths. A **directed path** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next. For example, a, c, e, f is a directed path from a to f in the graph in Figure 1.6b.

A graph is said to be **connected** if for every pair of its vertices u and v there is a path from u to v . If we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece. If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph. Formally, a **connected component** is a maximal (not expandable by including another vertex and an edge) connected subgraph² of a given graph. For example, the graphs in Figures 1.6a and 1.8a are connected, whereas the graph in Figure 1.9 is not, because there is no path, for example, from a to f . The graph in Figure 1.9 has two connected components with vertices $\{a, b, c, d, e\}$ and $\{f, g, h, i\}$, respectively.

Graphs with several connected components do happen in real-world applications. A graph representing the Interstate highway system of the United States would be an example (why?).

It is important to know for many applications whether or not a graph under consideration has cycles. A **cycle** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, f, h, i, g, f is a cycle in the graph in Figure 1.9. A graph with no cycles is said to be **acyclic**. We discuss acyclic graphs in the next subsection.

Trees

A **tree** (more accurately, a **free tree**) is a connected acyclic graph (Figure 1.10a). A graph that has no cycles but is not necessarily connected is called a **forest**: each of its connected components is a tree (Figure 1.10b).

2. A **subgraph** of a given graph $G = \langle V, E \rangle$ is a graph $G' = \langle V', E' \rangle$ such that $V' \subseteq V$ and $E' \subseteq E$.

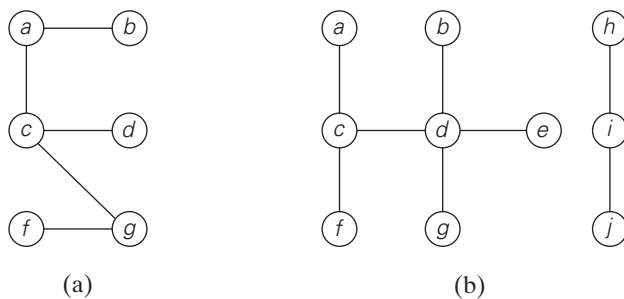


FIGURE 1.10 (a) Tree. (b) Forest.

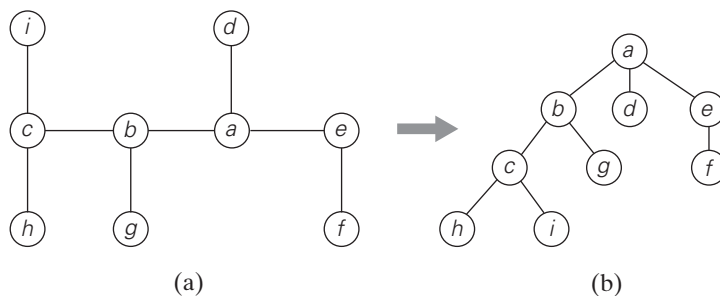


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices:

$$|E| = |V| - 1.$$

As the graph in Figure 1.9 demonstrates, this property is necessary but not sufficient for a graph to be a tree. However, for connected graphs it is sufficient and hence provides a convenient way of checking whether a connected graph has a cycle.

Rooted Trees Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the **root** of the so-called **rooted tree**. A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on. Figure 1.11 presents such a transformation from a free tree to a rooted tree.

Rooted trees play a very important role in computer science, a much more important one than free trees do; in fact, for the sake of brevity, they are often referred to as simply “trees.” An obvious application of trees is for describing hierarchies, from file directories to organizational charts of enterprises. There are many less obvious applications, such as implementing dictionaries (see below), efficient access to very large data sets (Section 7.4), and data encoding (Section 9.4). As we discuss in Chapter 2, trees also are helpful in analysis of recursive algorithms. To finish this far-from-complete list of tree applications, we should mention the so-called *state-space trees* that underline two important algorithm design techniques: backtracking and branch-and-bound (Sections 12.1 and 12.2).

For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called **ancestors** of v . The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as the set of **proper ancestors**. If (u, v) is the last edge of the simple path from the root to vertex v (and $u \neq v$), u is said to be the **parent** of v and v is called a **child** of u ; vertices that have the same parent are said to be **siblings**. A vertex with no children is called a **leaf**; a vertex with at least one child is called **parental**. All the vertices for which a vertex v is an ancestor are said to be **descendants** of v ; the **proper descendants** exclude the vertex v itself. All the descendants of a vertex v with all the edges connecting them form the **subtree** of T rooted at that vertex. Thus, for the tree in Figure 1.11b, the root of the tree is a ; vertices d, g, f, h , and i are leaves, and vertices a, b, e , and c are parental; the parent of b is a ; the children of b are c and g ; the siblings of b are d and e ; and the vertices of the subtree rooted at b are $\{b, c, g, h, i\}$.

The **depth** of a vertex v is the length of the simple path from the root to v . The **height** of a tree is the length of the longest simple path from the root to a leaf. For example, the depth of vertex c of the tree in Figure 1.11b is 2, and the height of the tree is 3. Thus, if we count tree levels top down starting with 0 for the root’s level, the depth of a vertex is simply its level in the tree, and the tree’s height is the maximum level of its vertices. (You should be alert to the fact that some authors define the height of a tree as the number of levels in it; this makes the height of a tree larger by 1 than the height defined as the length of the longest simple path from the root to a leaf.)

Ordered Trees An *ordered tree* is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree’s diagram, all the children are ordered left to right.

A **binary tree** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a **left child** or a **right child** of its parent; a binary tree may also be empty. An example of a binary tree is given in Figure 1.12a. The binary tree with its root at the left (right) child of a vertex in a binary tree is called the **left (right) subtree** of that vertex. Since left and right subtrees are binary trees as well, a binary tree can also be defined recursively. This makes it possible to solve many problems involving binary trees by recursive algorithms.

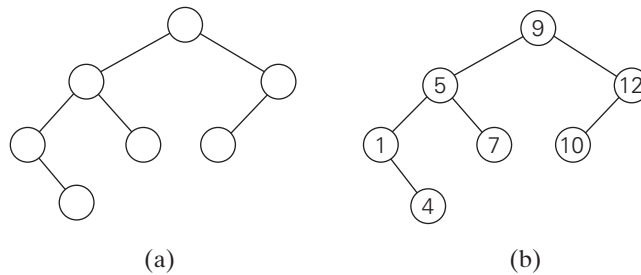


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

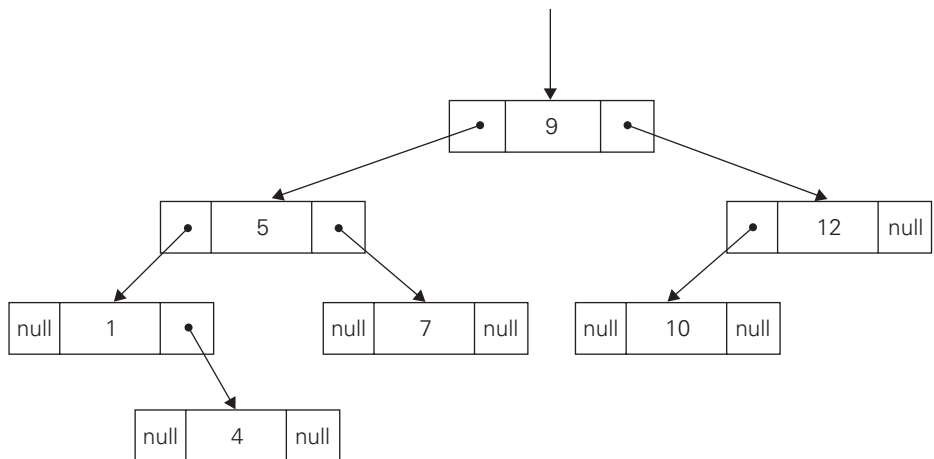


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b.

In Figure 1.12b, some numbers are assigned to vertices of the binary tree in Figure 1.12a. Note that a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called **binary search trees**. Binary trees and binary search trees have a wide variety of applications in computer science; you will encounter some of them throughout the book. In particular, binary search trees can be generalized to more general types of search trees called **multiway search trees**, which are indispensable for efficient access to very large data sets.

As you will see later in the book, the efficiency of most important algorithms for binary search trees and their extensions depends on the tree's height. Therefore, the following inequalities for the height h of a binary tree with n nodes are especially important for analysis of such algorithms:

$$\lceil \log_2 n \rceil \leq h \leq n - 1.$$

A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree. Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively. Figure 1.13 illustrates such an implementation for the binary search tree in Figure 1.12b.

A computer representation of an arbitrary ordered tree can be done by simply providing a parental vertex with the number of pointers equal to the number of its children. This representation may prove to be inconvenient if the number of children varies widely among the nodes. We can avoid this inconvenience by using nodes with just two pointers, as we did for binary trees. Here, however, the left pointer will point to the first child of the vertex, and the right pointer will point to its next sibling. Accordingly, this representation is called the **first child–next sibling representation**. Thus, all the siblings of a vertex are linked via the nodes' right pointers in a singly linked list, with the first element of the list pointed to by the left pointer of their parent. Figure 1.14a illustrates this representation for the tree in Figure 1.11b. It is not difficult to see that this representation effectively transforms an ordered tree into a binary tree said to be associated with the ordered tree. We get this representation by “rotating” the pointers about 45 degrees clockwise (see Figure 1.14b).

Sets and Dictionaries

The notion of a set plays a central role in mathematics. A **set** can be described as an unordered collection (possibly empty) of distinct items called **elements** of the

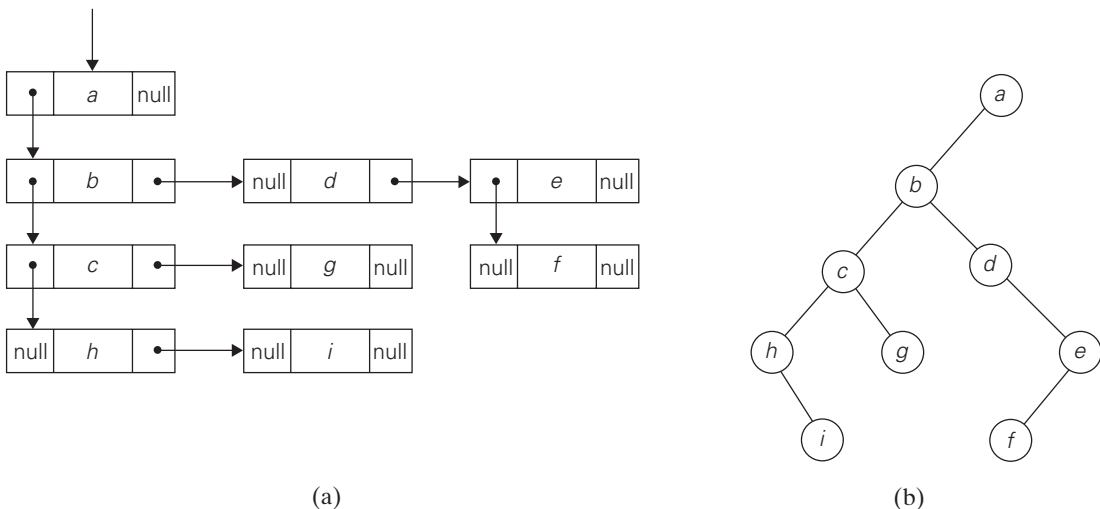


FIGURE 1.14 (a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree representation.

set. A specific set is defined either by an explicit listing of its elements (e.g., $S = \{2, 3, 5, 7\}$) or by specifying a property that all the set's elements and only they must satisfy (e.g., $S = \{n: n \text{ is a prime number smaller than } 10\}$). The most important set operations are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets.

Sets can be implemented in computer applications in two ways. The first considers only sets that are subsets of some large set U , called the **universal set**. If set U has n elements, then any subset S of U can be represented by a bit string of size n , called a **bit vector**, in which the i th element is 1 if and only if the i th element of U is included in set S . Thus, to continue with our example, if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, then $S = \{2, 3, 5, 7\}$ is represented by the bit string 011010100. This way of representing sets makes it possible to implement the standard set operations very fast, but at the expense of potentially using a large amount of storage.

The second and more common way to represent a set for computing purposes is to use the list structure to indicate the set's elements. Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need. Note, however, the two principal points of distinction between sets and lists. First, a set cannot contain identical elements; a list can. This requirement for uniqueness is sometimes circumvented by the introduction of a **multiset**, or **bag**, an unordered collection of items that are not necessarily distinct. Second, a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite. This is an important theoretical distinction, but fortunately it is not important for many applications. It is also worth mentioning that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the **dictionary**. Note the relationship between this data structure and the problem of searching mentioned in Section 1.3; obviously, we are dealing here with searching in a dynamic context. Consequently, an efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations. There are quite a few ways a dictionary can be implemented. They range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees, which we discuss later in the book.

A number of applications in computing require a dynamic partition of some n -element set into a collection of disjoint subsets. After being initialized as a collection of n one-element subsets, the collection is subjected to a sequence of intermixed union and search operations. This problem is called the **set union problem**. We discuss efficient algorithmic solutions to this problem in Section 9.2, in conjunction with one of its important applications.

You may have noticed that in our review of basic data structures we almost always mentioned specific operations that are typically performed for the structure in question. This intimate relationship between the data and operations has been recognized by computer scientists for a long time. It has led them in particular to the idea of an **abstract data type (ADT)**: a set of abstract objects representing data items with a collection of operations that can be performed on them. As illustrations of this notion, reread, say, our definitions of the priority queue and dictionary. Although abstract data types could be implemented in older procedural languages such as Pascal (see, e.g., [Aho83]), it is much more convenient to do this in object-oriented languages such as C++ and Java, which support abstract data types by means of **classes**.

Exercises 1.4


1. Describe how one can implement each of the following operations on an array so that the time it takes does not depend on the array's size n .
 - a. Delete the i th element of an array ($1 \leq i \leq n$).
 - b. Delete the i th element of a sorted array (the remaining array has to stay sorted, of course).
2. If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for
 - a. lists represented as arrays.
 - b. lists represented as linked lists.
3. a. Show the stack after each operation of the following sequence that starts with the empty stack:

$$\text{push}(a), \text{push}(b), \text{pop}, \text{push}(c), \text{push}(d), \text{pop}$$
- b. Show the queue after each operation of the following sequence that starts with the empty queue:

$$\text{enqueue}(a), \text{enqueue}(b), \text{dequeue}, \text{enqueue}(c), \text{enqueue}(d), \text{dequeue}$$
4. a. Let A be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that
 - i. the graph is complete.
 - ii. the graph has a loop, i.e., an edge connecting a vertex to itself.
 - iii. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.
- b. Answer the same questions for the adjacency list representation.
5. Give a detailed description of an algorithm for transforming a free tree into a tree rooted at a given vertex of the free tree.

6. Prove the inequalities that bracket the height of a binary tree with n vertices:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1.$$

7. Indicate how the ADT priority queue can be implemented as
- an (unsorted) array.
 - a sorted array.
 - a binary search tree.
8. How would you implement a dictionary of a reasonably small size n if you knew that all its elements are distinct (e.g., names of the 50 states of the United States)? Specify an implementation of each dictionary operation.
9. For each of the following applications, indicate the most appropriate data structure:
- answering telephone calls in the order of their known priorities
 - sending backlog orders to customers in the order they have been received
 - implementing a calculator for computing simple arithmetical expressions
-  10. *Anagram checking* Design an algorithm for checking whether two given words are anagrams, i.e., whether one word can be obtained by permuting the letters of the other. For example, the words *tea* and *eat* are anagrams.

SUMMARY

- An *algorithm* is a sequence of nonambiguous instructions for solving a problem in a finite amount of time. An input to an algorithm specifies an *instance* of the problem the algorithm solves.
- Algorithms can be specified in a natural language or pseudocode; they can also be implemented as computer programs.
- Among several ways to classify algorithms, the two principal alternatives are:
 - to group algorithms according to types of problems they solve
 - to group algorithms according to underlying design techniques they are based upon
- The important problem types are sorting, searching, string processing, graph problems, combinatorial problems, geometric problems, and numerical problems.
- Algorithm *design techniques* (or “strategies” or “paradigms”) are general approaches to solving problems algorithmically, applicable to a variety of problems from different areas of computing.

- Although designing an algorithm is undoubtedly a creative activity, one can identify a sequence of interrelated actions involved in such a process. They are summarized in Figure 1.2.
- A good algorithm is usually the result of repeated efforts and rework.
- The same problem can often be solved by several algorithms. For example, three algorithms were given for computing the greatest common divisor of two integers: *Euclid's algorithm*, the consecutive integer checking algorithm, and the middle-school method enhanced by the *sieve of Eratosthenes* for generating a list of primes.
- Algorithms operate on data. This makes the issue of data structuring critical for efficient algorithmic problem solving. The most important elementary data structures are the *array* and the *linked list*. They are used for representing more abstract data structures such as the *list*, the *stack*, the *queue*, the *graph* (via its *adjacency matrix* or *adjacency lists*), the *binary tree*, and the *set*.
- An abstract collection of objects with several operations that can be performed on them is called an *abstract data type (ADT)*. The *list*, the *stack*, the *queue*, the *priority queue*, and the *dictionary* are important examples of abstract data types. Modern object-oriented languages support implementation of ADTs by means of classes.

This page intentionally left blank