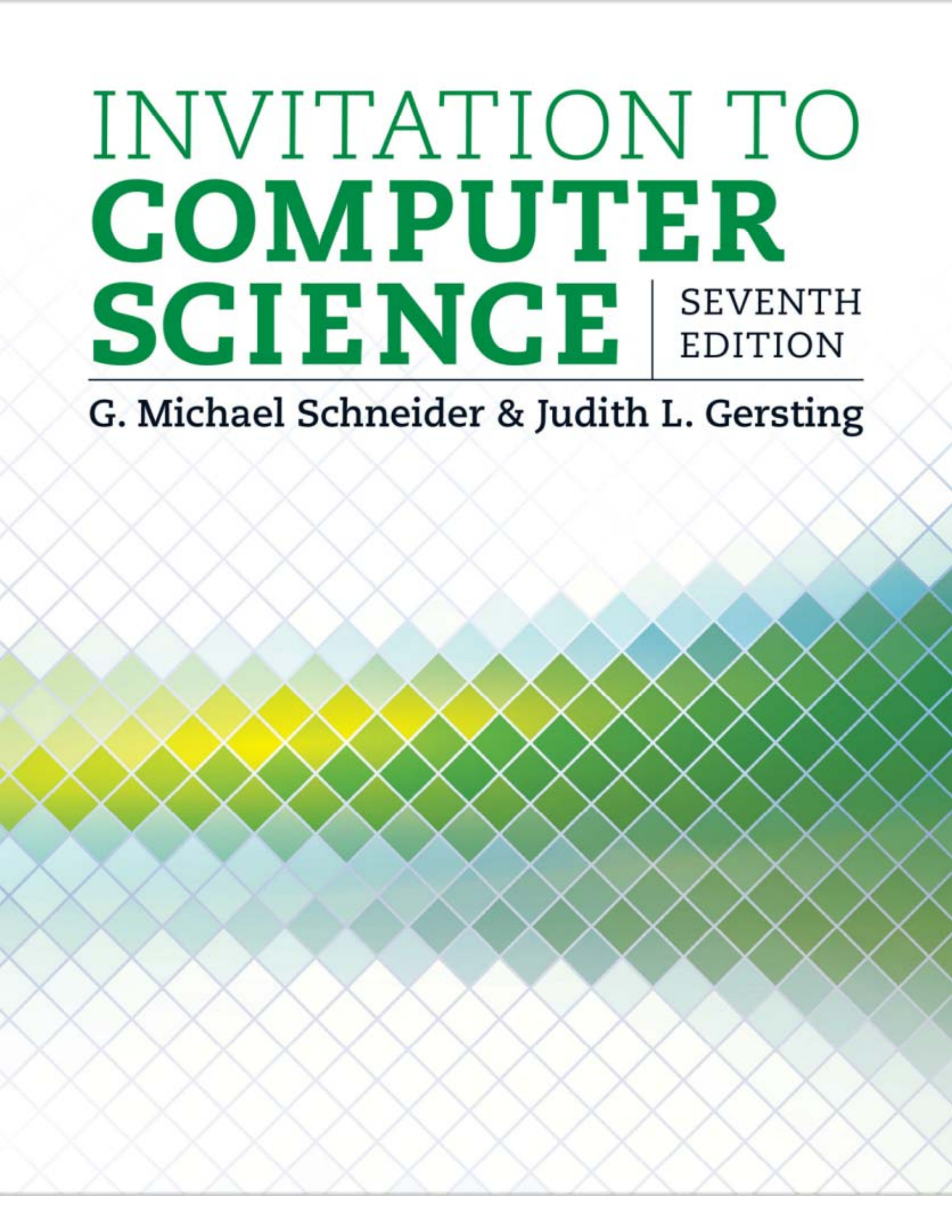
INVITATION TO COMPUTER SCIENCE

SEVENTH EDITION

G. Michael Schneider & Judith L. Gersting



7TH EDITION

Invitation to Computer Science

7TH EDITION

Invitation to Computer Science

G. Michael Schneider
Macalester College

Judith L. Gersting
Indiana University-Purdue University
at Indianapolis

Contributing author:
Bo Brinkman
Miami University

CENGAGE Learning

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

***Invitation to Computer Science*, Seventh Edition**

G. Michael Schneider and Judith L. Gersting

Product Director: Kathleen McMahon

Senior Content Developer: Alyssa Pratt

Development Editor: Deb Kaufmann

Marketing Manager: Eric LaScola

Manufacturing Planner: Julio Esperas

Art Director: Jack Pendleton

Production Management, Copyediting and Composition: Integra Software Services Pvt. Ltd.

Cover Photo: Studio-Pro/GettyImages

Library of Congress Control Number: 2014957763

ISBN: 978-1-305-07577-1

Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at **www.cengage.com/global.**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Cengage Learning Solutions, visit **www.cengage.com.**

Purchase any of our products at your local college store or at our preferred online store **www.cengagebrain.com.**

Printed in the United States of America
Print Number: 01 Print Year: 2015

To my wife, Ruthann, our children, Benjamin, Rebecca, and Trevor, grandson, Liam, and granddaughter, Sena.

G. M. S.

To my husband, John, and to: Adam and Francine; Jason, Cathryn, Sammie, and Johnny.

J. L. G.

Brief Contents

Online Chapters

This text includes five language-specific online-only downloadable chapters on Ada, C++, C#, Java, and Python, available at **www.cengagebrain.com** (search for the ISBN of this book) and on the CourseMate for this text.

Contents

Online Chapters

This text includes five language-specific online-only downloadable chapters on Ada, C++, C#, Java, and Python, available at **www.cengagebrain.com** (search for the ISBN of this book) and on the CourseMate for this text.

Preface to the Seventh Edition

Overview

This text is intended for a one-semester introductory course in computer science. It presents a breadth-first overview of the discipline that assumes no prior background in computer science, programming, or mathematics. It would be appropriate for a service course for students not majoring in computer science, as well as for schools that implement their introductory course for majors using a breadth-first approach that surveys the fundamental aspects of computer science and establishes a context for subsequent courses. It would be quite suitable for a high school computer science course as well. Previous editions of this text have been used in all these types of courses.

The Non-Majors Service Course

The introductory computer science service course (often called CS 0) has undergone numerous changes. In the 1970s and early 1980s, it was usually a class in Fortran, BASIC, or Pascal programming. In the mid-to-late 1980s, a rapid increase in computer use caused the service course to evolve into something called "computer literacy," in which students learned about new applications of computing in fields such as business, medicine, law, and education. With the growth of personal computers and productivity software, a typical early to mid-1990s version of this course would spend a semester teaching students how to use word processors, databases, spreadsheets, and email. The most recent change was its evolution into a web-centric course in which students learned to design and implement webpages using technologies such as HTML, XML, ASP, and Java applets.

In many institutions, the computer science service course is evolving once again. There are two reasons for this change. First, virtually all college and high school students are familiar with personal computers and productivity software. They have been using word processors and search engines since elementary school and are familiar with social networks, online retailing, and email; many have designed webpages and even manage their own

websites and blogs. In this day and age, a course that focuses solely on applications will be of little or no interest.

But a more important reason for rethinking the structure of the CS 0 service course, and the primary reason why we authored this book, is the following observation:

> *Most computer science service courses do not teach students the foundations and fundamental concepts of computer science!*

We believe quite strongly that, in addition to applications of information technology, students in a computer science service course should receive a solid grounding in the basic concepts of the discipline. This parallels the structure of introductory courses in biology, physics, and geology, which introduce the central concepts of the fields. Topics in this type of breadth-first computer science service course would not be limited to "fun" applications such as webpage creation, social networking, game design, and interactive graphics, but would also cover foundational issues such as algorithms, hardware, computer organization, system software, language models, theory of computation, and the social and ethical issues of computing. An introduction to these core ideas exposes students to the overall richness and beauty of the field and allows them not only to use computers and software effectively but also to understand and appreciate the basic ideas underlying the discipline of computer science. As a side benefit, students who complete such a course will have a much better idea of what a major or a minor in computer science would entail.

The First Course for Majors

Since the emergence of computer science as an academic discipline in the 1960s, the first course in the major (often called CS 1) has usually been an introduction to programming—from Fortran to BASIC to Pascal, and, later, C++, Java, and Python. But today there are numerous alternatives in the structure of CS 1, including the breadth-first overview. A first course for computer science majors using the breadth-first model emphasizes early exposure to the sub-disciplines of the field rather than placing exclusive emphasis on programming. This gives new majors a more complete and well-rounded understanding of their chosen field of study, including the many concepts and ways of thinking that are part of computer science.

Our book—intended for either majors or non-majors—is organized around this breadth-first approach as it presents a wide range of subject matter drawn from diverse areas of computer science. However, to avoid drowning students in a sea of seemingly unrelated facts and details, a breadth-first presentation must be carefully woven into a cohesive fabric, a theme, a "big picture" that ties together the individual topics and presents computer science as a unified and integrated discipline. To achieve this, our text divides the study of computer science into a hierarchy of topics, with each layer in the hierarchy building upon concepts presented in earlier chapters.

A Hierarchy of Abstractions

The central theme of this book is that *computer science is the study of algorithms*. Our hierarchy utilizes this definition by initially looking at the algorithmic foundations of computer science and then moving upward from this central theme to higher-level issues such as hardware, systems, software, applications, and ethics. Just as the chemist starts from the basic building blocks of protons, neutrons, and electrons and then builds upon these concepts to form atoms, molecules, and compounds, so, too, does our text build from elementary concepts to higher-level ideas.

The six levels in our computer science hierarchy are:

Level 1. The Algorithmic Foundations of Computer Science
Level 2. The Hardware World
Level 3. The Virtual Machine
Level 4. The Software World
Level 5. Applications
Level 6. Social Issues in Computing

Following an introductory chapter, **Level 1** (Chapters 2–3) introduces "The Algorithmic Foundations of Computer Science," the bedrock on which all other aspects of the discipline are built. It presents fundamental ideas such as the design of algorithms, algorithmic problem solving, abstraction, pseudocode, and iteration and illustrates these ideas using well-known examples. It also introduces the concepts of algorithm efficiency and asymptotic growth and demonstrates that not all algorithms are, at least in terms of running time, created equal.

The discussions in Level 1 assume that our algorithms are executed by something called a "computing agent," an abstract concept for any entity that can carry out the instructions in our solution. However, in **Level 2** (Chapters 4–5), "The Hardware World," we now want our algorithms to be executed by "real" computers to produce "real" results. Thus begins our discussion of hardware, logic design, and computer organization. The initial discussion introduces the basic building blocks of computer systems—binary numbers, Boolean logic, gates, and circuits. It then shows how these elementary concepts can be combined to construct a real computer using the Von Neumann architecture, composed of processors, memory, and input/output. This level presents a simple machine language instruction set and explains how the algorithmic primitives of Level 1, such as assignment and conditional, can be implemented in machine language and run on the Von Neumann hardware of Level 2, conceptually tying together these two areas. It ends with a discussion of important new directions in hardware design—multicore processors and massively parallel machines.

By the end of Level 2 students have been introduced to basic concepts in logic design and computer organization, and they can appreciate the complexity inherent in these ideas. This complexity is the motivation for the material contained in **Level 3** (Chapters 6–8), "The Virtual Machine." This section describes how system software can create a more friendly,

user-oriented problem-solving environment that hides many of the ugly hardware details just described. Level 3 looks at the same problems discussed in Level 2, encoding and executing algorithms, but shows how this can be done much more easily in a virtual environment containing helpful tools like a graphical user interface, editors, translators, file systems, and debuggers. This section discusses the services and responsibilities of the operating system and how it has evolved. It investigates one of the most important virtual environments in current use—networks of computers. It shows how technologies such as Ethernet, the Internet, and the web link together independent systems via transmission media and communications software. This creates a virtual environment in which we seamlessly and transparently use not only the computer on our desk or in our hand but also computing devices located anywhere in the world. This transparency has progressed to the point where we can now use systems located "in the cloud" without regard for where they are, how they provide their services, and even whether they exist as real physical entities. Level 3 concludes with a look at one of the most important services provided by a virtual machine, namely information security, and describes algorithms for protecting the user and the system from accidental or malicious damage.

Once we have created this powerful user-oriented virtual environment, what do we want to do with it? Most likely we want to write programs to solve interesting problems. This is the motivation for **Level 4** (Chapters 9–12), "The Software World." Although this book should not be viewed as a programming text, it contains an overview of the features found in modern procedural programming languages. This gives students an appreciation for the interesting and challenging task of the computer programmer and the power of the problem-solving environment created by a modern high-level language. (Detailed introductions to five important high-level programming languages are available via online, downloadable chapters accessible through the CourseMate for this text, as well as at *www.cengagebrain.com*.) There are many different language models, so Level 4 also includes a discussion of other language types, including special-purpose languages such as SQL, HTML, and JavaScript, as well as the functional, logic, and parallel language paradigms. An introduction to the design and construction of a compiler shows how high-level languages can be translated into machine language for execution. This latter discussion ties together numerous ideas from earlier chapters, as we show how an algorithm (Level 1), expressed in a high-level language (Level 4), can be compiled and executed on a typical Von Neumann machine (Level 2) by using system software tools (Level 3). These "recurring themes" and frequent references to earlier concepts help reinforce the idea of computer science as an integrated set of topics. At the conclusion of Level 4, we introduce the idea of computability and unsolvability to show students that there are provable limits to what programs, computers, and computer science can achieve.

We now have a high-level programming environment in which it is possible to write programs to solve important problems. In **Level 5** (Chapters 13–16), "Applications," we take a look at a few important uses of computers in our modern society. There is no way to cover more than a fraction of

the many applications of computers and information technology in a single section. Indeed, there is hardly a field of study or an aspect of our daily lives that has not been impacted by advances in computation and communication. Instead, we focus on a small set of interesting applications that demonstrate important concepts, tools, and techniques of computer science. This includes applications drawn from the sciences and engineering (simulation and modeling), business and finance (ecommerce, databases), the social sciences (artificial intelligence), and everyday life (computer-generated imagery, video gaming, virtual communities). Our goal is to show students that these applications are not "magic boxes" whose inner workings are totally unfathomable. Rather, they are the direct result of building upon the core concepts of computer science presented in the previous chapters. We hope these discussions encourage readers to seek out information on applications specific to their own areas of interest.

Finally, we reach the highest level of study, **Level 6** (Chapter 17), "Social Issues in Computing," which addresses the social, ethical, and legal issues raised by pervasive computer technology. This section (written by contributing author Professor Bo Brinkman of Miami University) examines issues such as the theft of intellectual property, national security concerns aggravated by information technology, and the erosion of personal privacy caused by the popularity of social networks. This chapter does not attempt to provide easy solutions to these many-faceted problems. Instead, it focuses on techniques that students can use to think about ethical issues and reach their own conclusions. Our goal in this final section is to make students aware of the enormous impact that information technology is having on our society and to give them tools for making informed decisions.

This, then, is the hierarchical structure of our text. It begins with the algorithmic foundations of the discipline and works its way from lower-level hardware concepts through virtual machine environments, high-level languages, software, and applications, to the social issues raised by computer technology. This organizational structure, along with the use of recurring themes, enables students to view computer science as a unified and coherent field of study. The material in Chapters 1–12 is intended to be covered sequentially, but the applications discussed in Chapters 13–16 can be covered in any order and the social issues in Chapter 17 can be presented at any time.

What's New in This Edition

This seventh edition of *Invitation to Computer Science* addresses a number of emerging issues in computer science. We have added significant new material on the Internet of Things, cloud computing, embedded computing, new models of electronic commerce, personal privacy, data mining, robots, and drones.

New and updated Special Interest Boxes highlight interesting historical vignettes, new developments in computing, biographies of important people in the field, and news items showing how computing is affecting our everyday lives. There are new in-chapter Practice Problems (with answers

provided at the end of the text) as well as new end-of-chapter Exercises. There are also new additions to the end-of-chapter Challenge Problems; these more complex questions can be used for longer assignments done either individually or by teams of students.

An Interactive Experience—CourseMate

This edition offers significantly enhanced supplementary material and additional resources available online through CourseMate. CourseMate is a valuable web resource containing an ebook with highlighting and note-taking capabilities, supplementary readings for each chapter, a glossary and flashcards of key technical terms, and links to interesting articles, helpful references, and relevant videos from across the web. The CourseMate encourages a truly interactive experience with study games, objective- and application-based quizzing, and hands-on exploration projects that speak students' language. Instructors may add CourseMate to the textbook package, or students may purchase CourseMate directly through *www.cengagebrain.com*.

An Experimental Science—Laboratory Software and Manual

Another important aspect of computer science education is the realization that, like physics, chemistry, and biology, computer science is an empirical, laboratory-based discipline in which learning comes not only from watching and listening but from doing and trying. Many ideas in computer science cannot be fully understood and appreciated until they are visualized, manipulated, and tested. Today, most computer science faculty see structured laboratories as an essential part of an introductory course. We concur, and this development is fully reflected in our approach to the material.

Associated with this text is a laboratory manual and custom-designed laboratory software that enables students to experiment with the concepts we present. The manual contains 20 laboratory experiences, closely coordinated with the main text, that cover all levels except Level 6. These labs give students the chance to observe, study, analyze, and/or modify an important idea or concept. For example, associated with Level 1 (the algorithmic foundations of computer science) are experiments that animate the algorithms in Chapters 2 and 3 and ask students to observe and discuss what is happening in these animations. There are also labs that allow students to measure the running time of these algorithms for different-sized data sets and discuss their behavior, thus providing concrete observations of an abstract concept like algorithmic efficiency. There are similar labs available for Levels 2, 3, 4, and 5 that highlight and clarify the material presented in the text.

Each of the lab manual experiments includes an explanation of how to use the software, a description of how to conduct the experiment, and problems for students to complete. For these lab projects, students can either work on their own or in teams, and the course may utilize either a closed-lab (formal, scheduled) or open-lab (informal, unscheduled) setting. The manual and software work well with all these laboratory models. The text contains "Laboratory Exercise" boxes that describe each lab and identify the point in the text where it would be most appropriate.

In this new seventh edition, the Laboratory Manual has been integrated into the CourseMate for this text. Instructors may add the CourseMate, including Lab Manual, to their textbook package; contact your sales representative for more information. Students may also purchase the CourseMate/Laboratory Manual directly through *www.cengagebrain.com*.

Programming and Online Language Modules

Programming concepts are presented in the text in the form of a survey of the features each high-level language provides and how they differ based on the computing tasks for which they were intended. Code examples are shown only to illustrate how algorithms can be embedded into the varying syntax of different languages. For instructors who want their students to have additional programming experience, online language modules for Ada, C++, C#, Java, and Python are available. Students may download any or all of these for free by going to the CourseMate for this text or to *www.cengagebrain.com*. At the CengageBrain home page, search for the ISBN of this book (found above the bar code on the back cover). At the Invitation page, click "Free Materials/Access Now". These PDF documents can be read online, downloaded to the student's computer, or printed and read on paper. Each chapter includes language-specific practice problems and exercises.

Computer science is a young and exciting discipline, and we hope that the new material in this edition, along with the laboratory projects and online modules, will convey this feeling of excitement to your students.

Instructor Resources

The following supplemental teaching tools are available when this book is used in a classroom setting. All supplements are available to instructors for download at *sso.cengage.com*.

Electronic Instructor's Manual

The Instructor's Manual follows the text chapter by chapter and includes material to assist in planning and organizing an effective, engaging course. The Instructor's Manual includes Overviews, Chapter Objectives, Teaching

Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms. A sample syllabus is also available.

Solutions

Complete solutions to chapter exercises are provided.

Test Bank

Cengage Learning Powered by Cognero® is a flexible, online system that allows you to:

- Author, edit, and manage test bank content from multiple Cengage Learning solutions
- Create multiple test versions in an instant
- Deliver tests from your Learning Management System (LMS), your classroom, or anywhere you want

PowerPoint Presentations

Microsoft PowerPoint slides to accompany each chapter are available. Slides may be used to guide classroom presentation or to print as classroom handouts, or they may be made available to students for chapter review. Instructors may customize the slides to best suit their course with the complete Figure Files from the text.

Acknowledgments

The authors would like to thank Alyssa Pratt and Deb Kaufmann for their invaluable assistance in developing this new edition, as well as the reviewers for this edition, whose comments were very helpful.

Colt Mazeau, *American National University*
William Oblitey, *Indiana University of Pennsylvania*
Gary Smith, *Elmhurst College*
Michael Verdicchio, *The Citadel*

Any errors, of course, are the fault solely of the authors.

—G. Michael Schneider
Macalester College
schneider@macalester.edu

—Judith L. Gersting
Indiana University-Purdue University at Indianapolis
gersting@iupui.edu

CHAPTER

1 An Introduction to Computer Science

CHAPTER TOPICS

1.1 Introduction

This text is an invitation to learn about one of the youngest and most exciting scientific disciplines—*computer science*. Almost every day our newspapers, televisions, and electronic media carry reports of significant advances in computing, such as high-speed supercomputers that perform more than 30 quadrillion (10^{15}) mathematical operations per second; wireless networks that stream high-definition video and audio to the remotest corners of the globe in fractions of a second; minute computer chips that can be embedded into appliances, clothing, and even our bodies; and artificial intelligence systems that understand and respond to English language questions faster and more accurately than humans. The next few years will see technological breakthroughs that, until a few years ago, existed only in the minds of dreamers and science fiction writers. These are exciting times in computing, and our goal in this text is to provide you with an understanding of computer science and an appreciation for the diverse areas of research and study within this important field.

Although the average person can produce a reasonably accurate description of most scientific fields, even if he or she did not study the subject in school, many people do not have an intuitive understanding of the types of problems studied by computer science professionals. For example, you probably know that biology is the study of living organisms and that chemistry deals with the structure and composition of matter. However, you might not have the same fundamental understanding of the work that goes on in computer science. In fact, many people harbor one or more of the following common misconceptions about this field.

MISCONCEPTION 1: *Computer science is the study of computers.*

This apparently obvious definition is actually incorrect or, to put it more precisely, incomplete. For example, some of the earliest and most fundamental theoretical work in computer science took place from 1920 to 1940, many years before the development of the first computer system. (This pioneering work was initially considered a branch of logic and applied mathematics. Computer science did not come to be recognized as a separate and independent field of scientific study until the late 1950s and early 1960s.) Even today, there are branches of computer science quite distinct from the study of "real" machines. In *theoretical computer science*, for example, researchers

study the logical and mathematical properties of problems and their solutions. Frequently, these researchers investigate problems not with actual computers but rather with *formal models* of computation, which are easier to study and analyze mathematically. Their work involves pencil and paper, not circuit boards and disks.

This distinction between computers and computer science was beautifully expressed by computer scientists Michael R. Fellows and Ian Parberry in an article in the journal *Computing Research News:*

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes, or chemistry is about beakers and test tubes. Science is not about tools. It is about how we use them and what we find out when we do.[1]

Misconception 2: *Computer science is the study of how to write computer programs.*

Many people are introduced to computer science when learning to write programs in a language such as C++, Python, or Java. This almost universal use of programming as the entry to the discipline can create the misunderstanding that computer science is equivalent to computer programming.

Programming is extremely important to the discipline—researchers use it to study new ideas and build and test new solutions—but, like the computer itself, programming is a tool. When computer scientists design and analyze a new approach to solving a problem or create new ways to represent information, they often implement their ideas as programs to test them on an actual computer system. This enables researchers to see how well these new ideas work and whether they perform better than previous methods.

---

[1]Fellows, M. R., and Parberry, I. "Getting Children Excited About Computer Science," *Computing Research News,* vol. 5, no. 1 (January 1993).

For example, searching a list is one of the most common applications of computers, and it is frequently applied to huge problems, such as finding one specific account among the approximately 63,000,000 active listings in the Social Security Administration database. A more efficient lookup method could significantly reduce the time that telephone-based customers must wait before receiving answers to questions regarding their accounts. Assume that we have designed what we believe to be a "new and improved" search technique. After analyzing it theoretically, we would study it empirically by writing a program to implement our new method, executing it on our computer, and measuring its performance. These tests would demonstrate under what conditions our new method is or is not faster than the search procedures currently in use.

In computer science, it is not simply the construction of a high-quality program that is important but also the methods it embodies, the services it provides, and the results it produces. It is possible to become so enmeshed in writing code and getting it to run that we forget that a program is only a means to an end, not an end in itself.

MISCONCEPTION 3: *Computer science is the study of the uses and applications of computers and software.*

If one's introduction to computer science is not programming, then it might be a course on the application of computers and software. Such a course typically teaches the use of a number of popular packages, such as word processors, search engines, database systems, spreadsheets, presentation software, smartphone apps, and web browsers.

These packages are widely used by professionals in all fields. However, learning to use a software package is no more a part of computer science than driver's education is a branch of automotive engineering. A wide range of people *use* computer software, but it is the computer scientist who is responsible for *specifying*, *designing*, *building*, and *testing* these software packages as well as the computer systems on which they run.

These three misconceptions about computer science are not entirely wrong; they are just woefully incomplete. Computers, programming languages, software, and applications *are* part of the discipline of computer science, but neither individually nor combined do they capture the richness and diversity of this field.

We have spent a good deal of time saying what computer science is *not*. What, then, is it? What are its basic concepts? What are the fundamental questions studied by professionals in this field? Is it possible to capture the breadth and scope of the discipline in a single definition? We answer these fundamental questions in the next section and, indeed, in the remainder of the text.

In the Beginning . . .

There is no single date that marks the beginning of computer science. Indeed, there are many "firsts" that could be used to mark this event. For example, some of the earliest theoretical work on the logical foundations of computer science occurred in the 1930s. The first general-purpose, electronic computers appeared during the period 1940–1946. (We will discuss the history of these early machines in Section 1.4.) These first computers were one-of-a-kind experimental systems that never moved outside the research laboratory. The first commercial machine, the UNIVAC I, did not make its appearance until March 1951, a date that marks the real beginning of the computer industry. The first high-level (i.e., based on natural language) programming language was FORTRAN. Some people mark its debut in 1957 as the beginning of the "software" industry. The appearance of these new machines and languages created new occupations, such as programmer, numerical analyst, and computer engineer. To address the intellectual needs of these workers, the first professional society for people in the field of computing, the Association for Computing Machinery (ACM), was established in 1947. (The ACM has more than 100,000 members and is the largest professional computer science society in the world. Its home page is *www.acm.org*.) To help meet the rapidly growing need for computer professionals, the first Department of Computer Science was established at Purdue University in October 1962. It awarded its first M.Sc. degree in 1964 and its first Ph.D. in computer science in 1966. An undergraduate program was established in 1967.

Thus, depending on what you consider the most important "first," the field of computer science is somewhere between 50 and 80 years old. Compared with such classic scientific disciplines as mathematics, physics, chemistry, and biology, computer science is the new kid on the block.

1.2 The Definition of Computer Science

There are many definitions of computer science, but the one that best captures the richness and breadth of ideas embodied in this branch of science was first proposed by professors Norman Gibbs and Allen Tucker.[2] According to their definition, the central concept in computer science is the **algorithm**. It is not possible to understand the field without a thorough understanding of this critically important idea.

[2]Gibbs, N. E., and Tucker, A. B. "A Model Curriculum for a Liberal Arts Degree in Computer Science," *Comm. of the ACM*, vol. 29, no. 3 (March 1986).

DEFINITION

Computer science: the study of algorithms, including

1. Their formal and mathematical properties
2. Their hardware realizations
3. Their linguistic realizations
4. Their applications

The Gibbs and Tucker definition says that it is the task of the computer scientist to design and develop algorithms to solve a range of important problems. This design process includes the following operations:

- Studying the behavior of algorithms to determine if they are correct and efficient (their formal and mathematical properties)
- Designing and building computer systems that are able to execute algorithms (their hardware realizations)
- Designing programming languages and translating algorithms into these languages so that they can be executed by the hardware (their linguistic realizations)
- Identifying important problems and designing correct and efficient software packages to solve these problems (their applications)

Because it is impossible to appreciate this definition fully without knowing what an algorithm is, let's look more closely at this term. The Merriam-Webster dictionary (*www.merriam-webster.com/dictionary/*) defines the word *algorithm* as follows:

> *al • go • rithm n. A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step method for accomplishing some task.*

Informally, an algorithm is an ordered sequence of instructions that is guaranteed to solve a specific problem. It is a list that looks something like this:

Step 1: Do something

Step 2: Do something

Step 3: Do something

. .

. .

. .

Step N: Stop, you are finished

If you are handed this list and carefully follow its instructions in the order specified, when you reach the end you will have solved the task at hand.

All the operations used to construct algorithms belong to one of only three categories:

1. Sequential operations A sequential instruction carries out a single well-defined task. When that task is finished, the algorithm moves on to the next operation. Sequential operations are usually expressed as simple declarative sentences.
   - Add 1 cup of butter to the mixture in the bowl.
   - Subtract the amount of the check from the current account balance.
   - Set the value of x to 1.

2. **Conditional operations** These are the "question-asking" instructions of an algorithm. They ask a question, and the next operation is then selected on the basis of the answer to that question.
   - If the mixture is too dry, then add one-half cup of water to the bowl.
   - If the amount of the check is less than or equal to the current account balance, then cash the check; otherwise, tell the person there are insufficient funds.
   - If x is not equal to 0, then set y equal to $1/x$; otherwise, print an error message that says you cannot perform division by 0.
3. **Iterative operations** These are the "looping" instructions of an algorithm. They tell us not to go on to the next instruction but, instead, to go back and repeat the execution of a previous block of instructions.
   - Repeat the previous two operations until the mixture has thickened.
   - While there are still more checks to be processed, do the following five steps.
   - Repeat Steps 1, 2, and 3 until the value of y is equal to $+1$.

We use algorithms (although we don't call them that) all the time—whenever we follow a set of instructions to assemble a child's toy, bake a cake, balance a checkbook, or go through the college registration process. A good example of an algorithm used in everyday life is the set of instructions shown in Figure 1.1 for programming a DVR to record a collection of television shows. Note the three types of instructions in this algorithm: sequential (Steps 3, 4, 5, and 7), conditional (Steps 1 and 6), and iterative (Step 2).

Mathematicians use algorithms all the time, and much of the work done by early Greek, Roman, Persian, and Indian mathematicians involved the discovery of algorithms for important problems in geometry and arithmetic; an example is *Euclid's algorithm* for finding the greatest common divisor of two positive integers. (Exercise 10 at the end of the chapter presents this

FIGURE 1.1

Step 1	If the clock and the calendar are not correctly set, then go to page 9 of the instruction manual and follow the instructions there before proceeding to Step 2
Step 2	Repeat Steps 3 through 6 for each program that you want to record
Step 3	Enter the channel number that you want to record and press the button labeled CHAN
Step 4	Enter the time that you want recording to start and press the button labeled TIME-START
Step 5	Enter the time that you want recording to stop and press the button labeled TIME-FINISH. This completes the programming of one show
Step 6	If you do not want to record anything else, press the button labeled END-PROG
Step 7	Turn off your DVR. Your DVR It is now in TIMER mode, ready to record

Programming your DVR: An example of an algorithm

2,300-year-old algorithm.) We also studied algorithms in elementary school, even if we didn't know it. For example, in the first grade we learned an algorithm for adding two numbers such as

$$\begin{array}{r} 47 \\ +25 \\ \hline 72 \end{array}$$

The instructions our teachers gave were as follows: First add the rightmost column of numbers (7 + 5), getting the value 12. Write down the 2 under the line and carry the 1 to the next column. Now move left to the next column, adding (4 + 2) and the previous carry value of 1 to get 7. Write this value under the line, producing the correct answer 72.

Although as children we learned this algorithm informally, it can, like the DVR instructions in Figure 1.1, be written formally as an explicit sequence of instructions. Figure 1.2 shows an algorithm for adding two positive m-digit numbers. It expresses formally the operations informally described previously. Again, note the three types of instructions used to construct the algorithm: sequential (Steps 1, 2, 4, 6, 7, 8, and 9), conditional (Step 5), and iterative (Step 3).

Even though it might not appear so, this is the same "decimal addition algorithm" that you learned in grade school; if you follow it rigorously, it is guaranteed to produce the correct result. Let's watch it work.

Add $(47 + 25)$
$m = 2$
$a_1 = 4$ $\quad a_0 = 7$
$b_1 = 2$ $\quad b_0 = 5$

} The input

FIGURE 1.2

Given: $m \geq 1$ and two positive numbers each containing m digits, $a_{m-1}\, a_{m-2} \ldots a_0$ and $b_{m-1}\, b_{m-2} \ldots b_0$

Wanted: $c_m\, c_{m-1}\, c_{m-2} \ldots c_0$, where $c_m c_{m-1} c_{m-2} \ldots c_0 = (a_{m-1}\, a_{m-2} \ldots a_0) + (b_{m-1} b_{m-2} \ldots b_0)$

Algorithm:

Step 1 Set the value of *carry* to 0
Step 2 Set the value of i to 0
Step 3 While the value of i is less than or equal to $m - 1$, repeat the instructions in Steps 4 through 6
Step 4 Add the two digits a_i and b_i to the current value of *carry* to get c_i
Step 5 If $c_i \geq 10$, then reset c_i to $(c_i - 10)$ and reset the value of *carry* to 1; otherwise, set the new value of *carry* to 0
Step 6 Add 1 to i, effectively moving one column to the left
Step 7 Set c_m to the value of *carry*
Step 8 Print out the final answer, $c_m\, c_{m-1}\, c_{m-2} \ldots c_0$
Step 9 Stop

Algorithm for adding two m-digit numbers

Step 1: *carry* = 0

Step 2: *i* = 0

Step 3: We now repeat Steps 4 through 6 while *i* is less than or equal to 1

First repetition of the loop (*i* has the value 0)

Step 4: Add ($a_0 + b_0 + carry$), which is 7 + 5 + 0, so $c_0 = 12$

Step 5: Because $c_0 \geq 10$, we reset **c_0 to 2** and reset *carry* to 1

Step 6: Reset *i* to (0 + 1) = 1. Because *i* is less than or equal to 1, go back to Step 4

Second repetition of the loop (*i* has the value 1)

Step 4: Add ($a_1 + b_1 + carry$), which is 4 + 2 + 1, so **$c_1 = 7$**

Step 5: Because $c_1 < 10$, we reset *carry* to 0

Step 6: Reset *i* to (1 + 1) = 2. Because *i* is greater than 1, we do not repeat the loop but instead go to Step 7

Step 7: Set **$c_2 = 0$**

Step 8: Print out the answer $c_2\ c_1\ c_0 = 072$ (see the **boldface** values)

Step 9: Stop

Abu Ja'far Muhammad ibn Musa Al-Khowarizmi (AD 780–850?)

The word *algorithm* is derived from the last name of Muhammad ibn Musa Al-Khowarizmi, a famous Persian mathematician and author from the eighth and ninth centuries. Al-Khowarizmi was a teacher at the House of Wisdom in Baghdad and the author of the book *Kitab al jabr w'al muqabala,* which in English means "The Concise Book of Calculation by Reduction." Written in AD 820, it is one of the earliest mathematical textbooks, and its title gives us the word *algebra* (the Arabic word *al jabr* means "reduction").

In AD 825, Al-Khowarizmi wrote another book about the base-10 positional numbering system that had recently been developed in India. In this book, he described formalized, step-by-step procedures for doing arithmetic operations, such as addition, subtraction, and multiplication, on numbers represented in this new decimal system, much like the addition algorithm diagrammed in Figure 1.2. In the twelfth century, this book was translated into Latin, introducing the base-10 Hindu-Arabic numbering system to Europe, and Al-Khowarizmi's name became closely associated with these formal numerical techniques. His last name was rendered as Algoritmi in Latin characters, and eventually the formalized procedures that he pioneered and developed became known as *algorithms* in his honor.

We have reached the end of the algorithm, and it has correctly produced the sum of the two numbers 47 and 25, the three-digit result 072. (A more clever algorithm would omit the unnecessary leading zero at the beginning of the number if the last carry value is a zero. That modification is an exercise—Exercise 6—at the end of the chapter.) Try working through the algorithm shown in Figure 1.2 with another pair of numbers to be sure that you understand exactly how it functions.

The addition algorithm shown in Figure 1.2 is a highly formalized representation of a technique that most people learned in the first or second grade and that virtually everyone knows how to do informally. Why would we take such a simple task as adding two numbers and express it in so complicated a fashion? Why are formal algorithms so important in computer science? Because of the following fundamental idea:

If we can specify an algorithm to solve a problem, then we can automate its solution.

Once we have formally specified an algorithm, we can build a machine (or write a program or hire a person) to carry out the steps contained in the algorithm. The machine (or program or person) need not understand the concepts or ideas underlying the solution. It merely has to do Step 1, Step 2, Step 3, ... exactly as written. In computer science terminology, the machine, robot, person, or thing carrying out the steps of the algorithm is called a **computing agent**.

Thus, computer science can also be viewed as the *science of algorithmic problem solving*. Much of the research and development work in computer science involves discovering correct and efficient algorithms for a wide range of interesting problems, studying their properties, designing programming languages into which those algorithms can be encoded, and designing and building computer systems that can automatically execute these algorithms in an efficient manner.

At first glance, it might seem that every problem can be solved algorithmically. However, you will learn in Chapter 12 the startling fact (first proved by the German logician Kurt Gödel in the early 1930s) that there are problems for which no generalized algorithmic solution can possibly exist. These problems are, in a sense, *unsolvable*. No matter how much time and effort is put into obtaining a solution, none will ever be found. Gödel's discovery, which staggered the mathematical world, effectively places a limit on the ultimate capabilities of computers and computer scientists.

There are also problems for which it is theoretically possible to specify an algorithm but a computing agent would take so long to execute it that the solution is essentially useless. For example, to get a computer to play winning chess, we could adopt a *brute force* approach. Given a board position as input, the computer would examine every legal move it could possibly make, then every legal response an opponent could make to each initial move, then every response it could select to that move, and so on. This analysis would continue until the game reached a win, lose, or draw position. With that information, the computer would be able to optimally choose its next move. If, for simplicity's sake, we assume that there are 40 legal moves from any given position on a chessboard, and it takes about 30 moves to

reach a final conclusion, then the total number of board positions that our brute force program would need to evaluate in deciding its first move is

$$\underbrace{40 \times 40 \times 40 \times \ldots \times 40}_{30 \text{ times}} = 40^{30}, \text{ which is roughly } 10^{48}$$

If we use a supercomputer that evaluates 1 quadrillion (10^{15}) board positions per second, it would take about 30,000,000,000,000,000,000,000,000 years for the computer to make its first move! Obviously, a computer could not use a brute force technique to play a real chess game.

There also exist problems that we do not yet know *how* to solve algorithmically. Many of these involve tasks that require a degree of what we term "intelligence." For example, after only a few days a baby recognizes the face of his or her mother from among the many faces he or she sees. In a few months, the baby begins to develop coordinated sensory and motor control skills and can efficiently plan how to use them—how to get from the playpen to the toy on the floor without bumping into either the chair or the desk that is in the way. After a few years, the child begins to develop powerful language skills and abstract-reasoning capabilities.

We take these abilities for granted, but the operations just mentioned—sophisticated visual discrimination, high-level problem solving, abstract reasoning, sophisticated natural-language understanding—cannot be done well (or even at all) using the computer systems and software packages currently available. The primary reason is that researchers do not yet know how to specify these operations algorithmically. That is, they do not yet know how to specify a solution formally in a detailed step-by-step fashion. As humans, we are able to do them simply by using the "algorithms" in our heads. To appreciate this problem, imagine trying to describe algorithmically exactly what steps you follow when you are painting a picture, composing a love poem, or formulating a business plan.

Thus, algorithmic problem solving has many variations. Sometimes solutions do not exist; sometimes a solution is too inefficient to be of any use; sometimes a solution is not yet known. However, discovering an algorithmic solution has enormously important consequences. As we noted earlier, if we can create a correct and efficient algorithm to solve a problem, and if we encode it into a programming language, then we can take advantage of the speed and power of a computer system to automate the solution and produce the desired result. This is what computer science is all about.

1.3 Algorithms

1.3.1 The Formal Definition of an Algorithm

The formal definition of an algorithm is rather imposing and contains a number of important ideas. Let's take it apart, piece by piece, and analyze each of its separate points.

DEFINITION

Algorithm: a well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.

. . .a well-ordered collection. . .

An algorithm is a collection of operations, and there must be a clear and unambiguous *ordering* to these operations. Ordering means that we know which operation to do first and precisely which operation to do next as each step is successfully completed. After all, we cannot expect a computing agent to carry out our instructions correctly if it is confused about which instruction it should be doing next.

Consider the following "algorithm" that was taken from the back of a shampoo bottle and is intended to be instructions on how to use the product.

Step 1: Wet hair

Step 2: Lather

Step 3: Rinse

Step 4: Repeat

At Step 4, what operations should be repeated? If we go back to Step 1, we will be unnecessarily wetting our hair. (It is presumably still wet from the previous operations.) If we go back to Step 3 instead, we will not be getting our hair any cleaner because we have not reused the shampoo. The Repeat instruction in Step 4 is ambiguous in that it does not clearly specify what to do next. Therefore, it violates the well-ordered requirement of an algorithm. (It also has a second and even more serious problem—it never stops! We will have more to say about this second problem shortly.) Statements such as

- Go back and do it again. (Do *what* again?)
- Start over. (From *where*?)
- If you understand this material, you may skip ahead. (How *far*?)
- Do either Part 1 or Part 2. (How do I decide *which* one to do?)

are ambiguous and can leave us confused and unsure about what operation to do next. We must be extremely precise in specifying the order in which operations are to be carried out. One possible way is to number the steps of the algorithm and use these numbers to specify the proper order of execution. For example, the ambiguous operations just shown could be made more precise as follows:

- Go back to Step 3 and continue execution from that point.
- Start over from Step 1.
- If you understand this material, skip ahead to Line 21.
- If you are 18 years of age or older, do Part 1 beginning with Step 9; otherwise, do Part 2 beginning with Step 40.

. . .of unambiguous and effectively computable operations. . .

Algorithms are composed of things called "operations," but what do those operations look like? What types of building blocks can be used to construct an algorithm? The answer to these questions is that the operations

used in an algorithm must meet two criteria—they must be *unambiguous*, and they must be *effectively computable*.

Here is a possible "algorithm" for making a cherry pie:

Step 1: Make the crust

Step 2: Make the cherry filling

Step 3: Pour the filling into the crust

Step 4: Bake at 350°F for 45 minutes

For a professional baker, this algorithm would be fine. He or she would understand how to carry out each of the operations listed. Novice cooks, like most of us, would probably understand the meaning of Steps 3 and 4. However, we would probably look at Steps 1 and 2, throw up our hands in confusion, and ask for clarification. We might then be given more detailed instructions.

Step 1: Make the crust
- 1.1 Take one and one-third cups flour
- 1.2 Sift the flour
- 1.3 Mix the sifted flour with one-half cup butter and one-fourth cup water
- 1.4 Roll into two 9-inch pie crusts

Step 2: Make the cherry filling
- 2.1 Open a 16-ounce can of cherry pie filling and pour into bowl
- 2.2 Add a dash of cinnamon and nutmeg, and stir

With this additional information, most people—even inexperienced cooks—would understand what to do and could successfully carry out this baking algorithm. However, there might be some people, perhaps young children, who still do not fully understand each and every line. For those people, we must go through the simplification process again and describe the ambiguous steps in even more elementary terms.

For example, the computing agent executing the algorithm might not know the meaning of the instruction "Sift the flour" in Step 1.2, and we would have to explain it further.

1.2 Sift the flour
- 1.2.1 Get out the sifter, which is the device shown on page A-9 of your cookbook, and place it directly on top of a 2-quart bowl
- 1.2.2 Pour the flour into the top of the sifter and turn the crank in a counterclockwise direction
- 1.2.3 Let all the flour fall through the sifter into the bowl

Now, even a child should be able to carry out these operations. But if that were not the case, then we would go through the simplification process yet one more time, until every operation, every sentence, every word was clearly understood.

An **unambiguous operation** is one that can be understood and carried out directly by the computing agent without further simplification

or explanation. When an operation is unambiguous, we call it a *primitive operation,* or simply a primitive of the computing agent carrying out the algorithm. An algorithm must be composed entirely of primitives. Naturally, the primitive operations of different individuals (or machines) vary depending on their sophistication, experience, and intelligence, as is the case with the cherry pie recipe, which varies with the baking experience of the person following the instructions. Hence, an algorithm for one computing agent might not be an algorithm for another.

One of the most important questions we will answer in this text is, *What are the primitive operations of a typical modern computer system?* Which operations can a hardware processor "understand" in the sense of being able to carry out directly, and which operations must be further refined and simplified?

However, it is not enough for an operation to be understandable. It must also be *doable* by the computing agent. If an algorithm tells me to flap my arms really quickly and fly, I understand perfectly well what it is asking me to do. However, I am incapable of doing it. "Doable" means there exists a computational process that allows the computing agent to complete that operation successfully. The formal term for "doable" is effectively computable.

For example, the following is an incorrect technique for finding and printing the 100th prime number. (A prime number is a whole number not evenly divisible by any numbers other than 1 and itself, such as 2, 3, 5, 7, 11, 13, ...)

Step 1: Generate a list L of all the prime numbers: L_1, L_2, L_3, ...

Step 2: Sort the list L into ascending order

Step 3: Print out the 100th element in the list, L_{100}

Step 4: Stop

The problem with these instructions is in Step 1, "Generate a list L of *all* the prime numbers...." That operation cannot be completed. There are an infinite number of prime numbers, and it is not possible in a finite amount of time to generate the desired list L. No such computational process exists, and the operation described in Step 1 is not effectively computable. Here are some other examples of operations that, under certain circumstances, may not be effectively computable:

Set *number* to 0.
Set *average* to *(sum ÷ number)*. (Division by 0 is not permitted.)

Set N to -1.
Set the value of *result* to $\sqrt{N}$. (You cannot take the square root of negative values using real numbers.)

Add 1 to the current value of x. (What if *x* currently has no value?)

This last example explains why we had to initialize the value of the variable called *carry* to 0 in Step 1 of Figure 1.2. In Step 4, the algorithm says, "Add the two digits a_i and b_i to the current value of *carry* to get c_i." If

carry has no current value, then when the computing agent tries to perform the instruction in Step 4, it will not know what to do, and this operation is not effectively computable.

. . .that produces a result. . .

Algorithms solve problems. To know whether a solution is correct, an algorithm must produce a result that is observable to a user, such as a numerical answer, a new object, or a change to its environment. Without some observable result, we would not be able to say whether the algorithm is right or wrong or even if it has completed its computations. In the case of the DVR algorithm (Figure 1.1), the result will be a set of recorded TV programs. The addition algorithm (Figure 1.2) produces an m-digit sum.

Note that we use the word *result* rather than *answer*. Sometimes it is not possible for an algorithm to produce the correct answer because for a given set of input, a correct answer does not exist. In those cases, the algorithm may produce something else, such as an error message, a red warning light, or an approximation to the correct answer. Error messages, lights, and approximations, although not necessarily what we wanted, are all observable results.

. . .and halts in a finite amount of time.

Another important characteristic of algorithms is that the result must be produced after the execution of a finite number of operations, and we must guarantee that the algorithm eventually reaches a statement that says, "Stop, you are done" or something equivalent. We have already pointed out that the shampooing algorithm was not well ordered because we did not know which statements to repeat in Step 4. However, even if we knew which block of statements to repeat, the algorithm would still be incorrect because it makes no provision to terminate. It will essentially run forever, or until we run out of hot water, soap, or patience. This is called an **infinite loop**, and it is a common error in the design of algorithms.

Figure 1.3 shows an algorithmic solution to the shampooing problem that meets all the criteria discussed in this section if we assume

FIGURE 1.3

Step	Operation
1	Wet your hair
2	Set the value of *WashCount* to 0
3	Repeat Steps 4 through 6 until the value of *WashCount* equals 2
4	Lather your hair
5	Rinse your hair
6	Add 1 to the value of *WashCount*
7	Stop, you have finished shampooing your hair

A correct solution to the shampooing problem

that you want to wash your hair twice. The algorithm of Figure 1.3 is well ordered. Each step is numbered, and the execution of the algorithm unfolds sequentially, beginning at Step 1 and proceeding from instruction i to instruction $i + 1$, unless the operation specifies otherwise. (For example, the iterative instruction in Step 3 says that after completing Step 6, you should go back and start again at Step 4 until the value of *WashCount* equals 2.) The intent of each operation is (we assume) clear, unambiguous, and doable by the person washing his or her hair. Finally, the algorithm will halt. This is confirmed by observing that *WashCount* is initially set to 0 in Step 2. Step 6 says to add 1 to *WashCount* each time we lather and rinse our hair, so it will take on the values 0, 1, 2, ... However, the iterative statement in Step 3 says stop lathering and rinsing when the value of *WashCount* reaches 2. At that point, the algorithm goes to Step 7 and terminates execution with the desired result: clean hair. (Although it is correct, do not expect to see this algorithm on the back of a shampoo bottle in the near future.)

As is true for any recipe or set of instructions, there is always more than a single way to write a correct solution. For example, the algorithm of Figure 1.3 could also be written as shown in Figure 1.4. Both of these are correct solutions to the shampooing problem. (Although they are both correct, they are not necessarily equally elegant. This point is addressed in Exercise 9 at the end of the chapter.)

1.3.2 The Importance of Algorithmic Problem Solving

The instruction sequences in Figures 1.1, 1.2, 1.3, and 1.4 are examples of the types of algorithmic solutions designed, analyzed, implemented, and tested by computer scientists, although they are much shorter and simpler. The operations shown in these figures could be encoded into some appropriate language and given to a computing agent (such as a personal computer or a robot) to execute. The device would mechanically follow

FIGURE 1.4

Step	Operation
1	Wet your hair
2	Lather your hair
3	Rinse your hair
4	Lather your hair
5	Rinse your hair
6	Stop, you have finished shampooing your hair

Another correct solution to the shampooing problem

these instructions and successfully complete the task. The device could do this without having to understand the creative processes that went into the discovery of the solution and without knowing the principles and concepts that underlie the problem. The robot simply follows the steps in the specified order (a required characteristic of algorithms), successfully completing each operation (another required characteristic), and ultimately producing the desired result after a finite amount of time (also required).

Just as the Industrial Revolution of the nineteenth century allowed machines to take over the drudgery of repetitive physical tasks, the "computer revolution" of the twentieth and twenty-first centuries has enabled us to implement algorithms that mechanize and automate the drudgery of repetitive mental tasks, such as adding long columns of numbers, finding one specific name or account number within a massive database, sorting student records by course number, and retrieving hotel or airline reservations from a file containing millions of pieces of data. This mechanization process offers the prospect of enormous increases in productivity. It also frees people to do those things that humans do much better than computers, such as creating new ideas, setting policy, doing high-level planning, and determining the significance of the results produced by a computer. Certainly, these operations are a much more effective use of that unique computing agent called the human brain.

Practice Problems

Get a copy of the instructions that describe how to do the following:

1. Register for classes at the beginning of the semester.
2. Use the online catalog to see what is available in the college library on a given subject.
3. Use the copying machine in your building.
4. Do an "Advanced Search" using Google.
5. Add someone as a friend to your Facebook account.

Look over the instructions and decide whether they meet the definition of an algorithm given in this section. If not, explain why, and rewrite each set of instructions so that it constitutes a valid algorithm. Also state whether each instruction is a sequential, a conditional, or an iterative operation.

1.4 A Brief History of Computing

Although computer science is not simply a study of computers, there is no doubt that the field was formed and grew in popularity as a direct response to their creation and widespread use. This section takes a brief look at the historical development of computer systems.

The appearance of some technologies, such as the telephone, the lightbulb, and the first heavier-than-air flight, can be traced directly to a single place, a specific individual, and an exact instant in time. Examples include the flight of Orville and Wilbur Wright on December 17, 1903, in Kitty Hawk, North Carolina, and the famous phrase "Mr. Watson—come here—I want to see you." uttered by Alexander Graham Bell over the first telephone on March 10, 1876.

Computers were not like that. They did not appear in a specific room on a given day as the creation of some individual genius. The ideas that led to the design of the first computers evolved over hundreds of years, with contributions coming from many people, each building on and extending the work of earlier discoverers.

1.4.1 The Early Period: Up to 1940

If this were a discussion of the history of mathematics and arithmetic instead of computer science, it would begin 3,000 years ago with the early work of the Greeks, Egyptians, Babylonians, Indians, Chinese, and Persians. All these cultures were interested in and made important contributions to the fields of mathematics, logic, and numerical computation. For example, the Greeks developed the fields of geometry and logic; the Babylonians and Egyptians developed numerical methods for generating square roots, multiplication tables, and trigonometric tables used by early sailors; Indian mathematicians developed both the base-10 decimal numbering system and the concept of zero; and in the ninth century, the Persians developed algorithmic problem solving (as you learned in the Abu Ja'far Muhammad ibn Musa Al-Khowarizmi Special Interest Box earlier in the chapter).

The first half of the seventeenth century saw a number of important developments related to automating and simplifying the drudgery of arithmetic computation. (The motivation for this work appears to be the sudden increase in scientific research during the sixteenth and seventeenth centuries in the areas of astronomy, chemistry, and medicine. This work required the solution of larger and more complex mathematical problems.) In 1614, the Scotsman John Napier invented *logarithms* as a way to simplify difficult mathematical computations. The early seventeenth century also witnessed the development of new and quite powerful mechanical devices designed to help reduce the burden of arithmetic. The first *slide rule* appeared around 1622. In 1642, the French philosopher and mathematician Blaise Pascal designed and built one of the first *mechanical calculators* (named the *Pascaline*) that could do addition and subtraction. A model of this early calculating device is shown in Figure 1.5.

FIGURE 1.5

The Pascaline, one of the earliest mechanical calculators

Source: INTERFOTO/Alamy

The famous German mathematician Gottfried Leibnitz (who, along with Isaac Newton, was one of the inventors of calculus) was also excited by the idea of automatic computation. He studied the work of Pascal and others, and in 1673, he constructed a mechanical calculator called *Leibnitz's Wheel* that could do not only addition and subtraction but multiplication and division as well. Both Pascal's and Leibnitz's machines used interlocking mechanical cogs and gears to store numbers and perform basic arithmetic operations. Considering the state of technology available to Pascal, Leibnitz, and others in the seventeenth century, these first calculating machines truly were mechanical wonders.

These early developments in mathematics and arithmetic were important milestones because they demonstrated how mechanization could simplify and speed up numerical computation. For example, Leibnitz's Wheel enabled seventeenth-century mathematicians to generate tables of mathematical functions many times faster than was possible by hand. (It is hard to believe in our modern high-tech society, but in the seventeenth century the generation of a table of logarithms could represent a *lifetime's* effort of one person!) However, the slide rule and mechanical calculators of Pascal and Leibnitz, although certainly impressive devices, were not computers. Specifically, they lacked two fundamental characteristics:

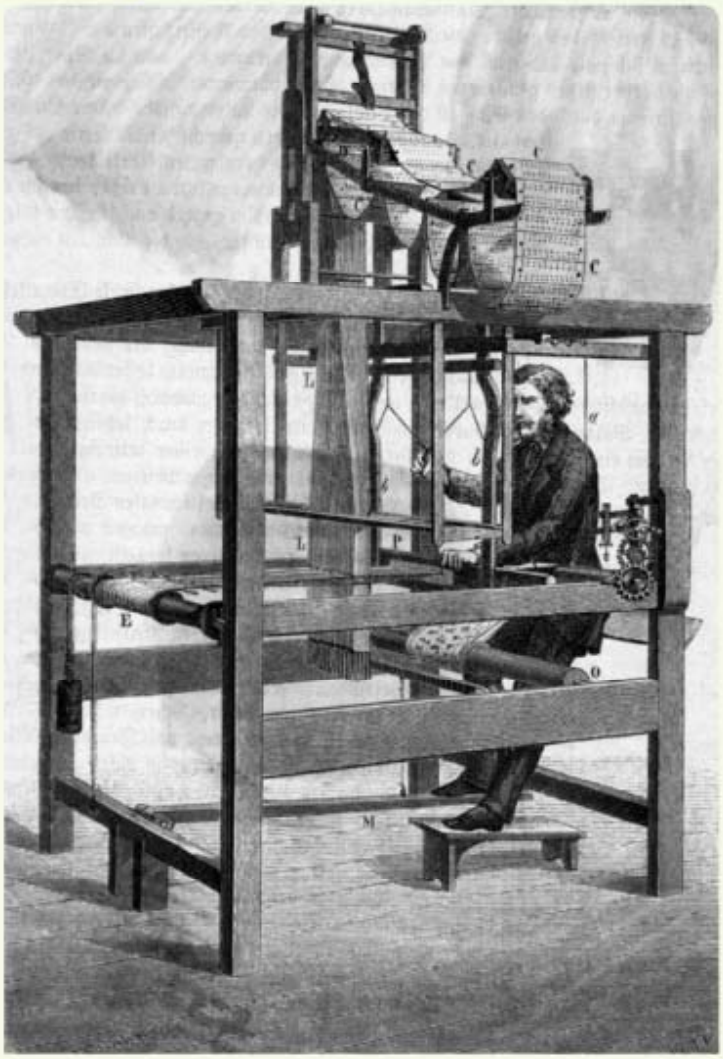
- They did not have a *memory* where information could be stored in machine-readable form.
- They were not *programmable.* A person could not provide *in advance* a sequence of instructions that could be executed by the device without manual intervention.

Surprisingly, the first actual "computing device" to include both these features was not created for the purposes of mathematical computations. Rather, it was a loom used for the manufacture of rugs and clothing. It was

developed in 1801 by the Frenchman Joseph Jacquard. Jacquard wanted to automate the weaving process, at the time a painfully slow and cumbersome task in which each separate row of the pattern had to be set up by the weaver and an apprentice. Because of this, anything but the most basic style of clothing was beyond the means of most people.

Jacquard designed an automated loom that used *punched cards* to create the desired pattern (Figure 1.6). If there was a hole in the card in a particular location, then a hook could pass through the card, grasp a warp thread, and raise it to allow a second thread to pass underneath. If there was no hole in the card, then the hook could not pass through, and the thread would pass over the warp. Depending on whether the thread passed above or below the warp,

FIGURE 1.6



Drawing of the Jacquard loom

Source: © Bettmann/CORBIS

The Original "Technophobia"

The development of the automated Jacquard loom and other technological advances in the weaving industry was so frightening to the craft guilds of the early nineteenth century that in 1811 it led to the formation of a group called the Luddites. The Luddites, named after their leader Ned Ludd of Nottingham, England, were violently opposed to this new manufacturing technology, and they burned down factories that attempted to use it. The movement lasted only a few years and its leaders were all jailed, but their name lives on today as a pejorative term for any group that is frightened and angered by the latest developments in any branch of science and technology, including computers.

a specific design was created. Each punched card described one row of the pattern. Jacquard connected the cards and fed them through his loom, and it automatically sequenced from card to card, weaving the desired pattern. The rows of connected punched cards can be seen at the top of the device.

Jacquard's loom represented an enormously important stage in the development of computers. Not only was it the first programmable device, but it showed how the knowledge of a human expert (in this case, a master weaver) could be captured in machine-readable form and used to control a machine that accomplished the same task automatically. Once the program was created, the expert was no longer needed. The lowliest apprentice could load the cards into the loom, turn it on, and produce a finished, high-quality product over and over again.

These pioneers had enormous influence on the designers and inventors who came after them, among them a mathematics professor at Cambridge University named Charles Babbage. Babbage was interested in automatic computation. In 1823, he extended the ideas of Pascal and Leibnitz and constructed a working model of the largest and most sophisticated mechanical calculator of its time. This machine, called the *Difference Engine,* could do addition, subtraction, multiplication, and division to 6 significant digits, and it could solve polynomial equations and other complex mathematical problems as well. Babbage tried to construct a larger model of the Difference Engine that would be capable of working to an accuracy of 20 significant digits, but after 12 years of work he had to give up his quest. The technology available in the 1820s and 1830s was not sufficiently advanced to manufacture cogs and gears to the precise tolerances his design required. Like Leonardo da Vinci's helicopter or Jules Verne's atomic submarine, Babbage's ideas were fundamentally sound but years ahead of their time. (In 1991, the London Museum

of Science, using Babbage's original plans, built an actual working model of the Difference Engine. It was 7 feet high and 11 feet wide, weighed 5 tons, and had 4,000 moving parts. It worked exactly as Babbage had planned.)

Babbage did not stop his investigations with the Difference Engine. In the 1830s, he designed a more powerful and general-purpose computational machine that could be configured to solve a much wider range of numerical problems. His machine had four basic components: a *mill* to perform the arithmetic manipulation of data, a *store* to hold the data, an *operator* to process the instructions contained on punched cards, and an *output unit* to put the results onto separate punched cards. Although it would be about 110 years before a "real" computer would be built, Babbage's proposed machine, called the Analytical Engine, is amazingly similar in design to a modern computer. The four components of the Analytical Engine are virtually identical in function to the four major components of today's computer systems:

Babbage's Term	*Modern Terminology*
mill	arithmetic/logic unit
store	memory
operator	processor
output unit	input/output

Babbage died before a working steam-powered model of his Analytical Engine could be completed, but his ideas lived on to influence others, and many computer scientists consider the Analytical Engine the first "true" computer system, even if it existed only on paper and in Babbage's dreams.

Another person influenced by the work of Pascal, Jacquard, and Babbage was a young statistician at the U.S. Census Bureau named Herman Hollerith. Because of the rapid increase in immigration to America at the end of the nineteenth century, officials estimated that doing the 1890 enumeration manually would take from 10 to 12 years. The 1900 census would begin before the previous one was finished. Something had to be done.

Hollerith designed and built programmable card-processing machines that could automatically read, tally, and sort data entered on punched cards. Census data were coded onto cards using a machine called a *keypunch*. The cards were taken either to a *tabulator* for counting and tallying or to a *sorter* for ordering alphabetically or numerically. Both of these machines were programmable (via wires and plugs) so that the user could specify such things as which card columns should be tallied and in what order the cards should be sorted. In addition, the machines had a small amount of memory to store results. Thus, they had all four components of Babbage's Analytical Engine.

Hollerith's machines were enormously successful, and they were one of the first examples of the use of automated information processing to solve large-scale, real-world problems. Whereas the 1880 census required 8 years to be completed, the 1890 census was finished in about 1 year, even though there was a 26% increase in the U.S. population during that decade.

These machines were not really general-purpose computers because each machine could do only a single task such as tabulate or sort. Nevertheless,

Hollerith's card machines were a very clear and very successful demonstration of the enormous advantages of automated information processing. This fact was not lost on Hollerith, who left the Census Bureau in 1902 to run his own Tabulating Machine Company to build and sell these machines. He planned to market his new product to a country that was just entering the Industrial Revolution and that, like the Census Bureau, would be generating and processing enormous volumes of inventory, production, accounting, and sales data. His punched-card machines became the dominant form of data-processing equipment during the first half of the twentieth century, well into the 1950s and 1960s. During this period, virtually every major U.S. corporation had data-processing rooms filled with keypunches, sorters, and tabulators, as well as drawer upon drawer of punched cards. In 1924, Hollerith's company changed its name to IBM, and it eventually evolved into the largest computing company in the world.

Charles Babbage (1791–1871) Ada Augusta Byron, Countess of Lovelace (1815–1852)

Charles Babbage, the son of a banker, was born into a life of wealth and comfort in eighteenth-century England. He attended Cambridge University and displayed an aptitude for mathematics and science. He was also an inventor and "tinkerer" who loved to build all sorts of devices. Among the devices he constructed were unpickable locks, skeleton keys, speedometers, and even the first cow catcher for trains. His first and greatest love, though, was mathematics, and he spent much of his life creating machines to do automatic computation. Babbage was enormously impressed by the work of Jacquard in France. (In fact, Babbage had on the wall of his home a woven portrait of Jacquard that was created using 24,000 punched cards.) He spent the last 30 to 40 years of his life trying to build a computing device, the Analytical Engine, based on Jacquard's ideas.

In that quest, he was helped by Countess Ada Augusta Byron, daughter of the famous English poet, Lord Byron. The countess was introduced to Babbage and was enormously impressed by his ideas about the Analytical Engine. As she put it, "We may say most aptly that the Analytical Engine weaves algebraic patterns just as the Jacquard Loom weaves flowers and leaves." Lady Lovelace worked closely with Babbage to specify how to organize instructions for the Analytical Engine to solve a particular mathematical problem. Because of that pioneering work, she is generally regarded as history's first computer programmer.

Babbage died in 1871 without realizing his dream. His work was generally forgotten until the twentieth century, when it became instrumental in moving the world into the computer age.

We have come a long way from the 1640s and the Pascaline, the early adding machine constructed by Pascal. We have seen the development of more powerful mechanical calculators (Leibnitz), automated programmable manufacturing devices (Jacquard), a design for the first computing device (Babbage), and the initial applications of information processing on a massive scale (Hollerith). However, we still have not yet entered the "computer age." That did not happen until around 1940, and it was motivated by an event that, unfortunately, has fueled many of the important technological advances in human history—the outbreak of war.

1.4.2 The Birth of Computers: 1940–1950

World War II created another, quite different set of information-based problems. Instead of inventory, sales, and payroll, the concerns became ballistics tables, troop deployment data, and secret codes. A number of research projects were started, funded largely by the military, to build automatic computing machines to perform these tasks and assist the Allies in the war effort.

Beginning in 1937, the U.S. Navy and IBM jointly funded a project under the direction of Professor Howard Aiken at Harvard University to build a computing device called Mark I. This was a general-purpose, electromechanical programmable computer that used a mix of relays, magnets, and gears to process and store data. The Mark I was the first computing device to use the base-2 binary numbering system, which we will discuss in Chapter 4. It used electromechanical switches and electric current to represent the two binary values, off for 0, on for 1. Until then, computing machines had used decimal representation, typically using a 10-toothed gear, each tooth representing one of the digits from 0 to 9. The Mark I was completed in 1944, about 110 years after Babbage's dream of the Analytical Engine, and is generally considered one of the first working general-purpose computers. The Mark I had a memory capacity of 72 numbers, and it could be programmed to perform a 23-digit multiplication in the lightning-like time of 4 seconds. Although laughably slow by modern standards, the Mark I was operational for almost 15 years, and it carried out a good deal of important mathematical work for the U.S. during the war.

At about the same time, a much more powerful machine was taking shape at the University of Pennsylvania in conjunction with the U.S. Army. During the early days of World War II, the Army was producing many new artillery pieces, but it found that it could not produce the firing tables equally as fast. These tables told the gunner how to aim the gun on the basis of such input as distance to the target and current temperature, wind, and elevation. Because of the enormous number of variables and the complexity of the computations (which use both trigonometry and calculus), these firing tables were taking more time to construct than the gun itself—a skilled person with a desk calculator required about 20 hours to analyze a single 60-second trajectory.

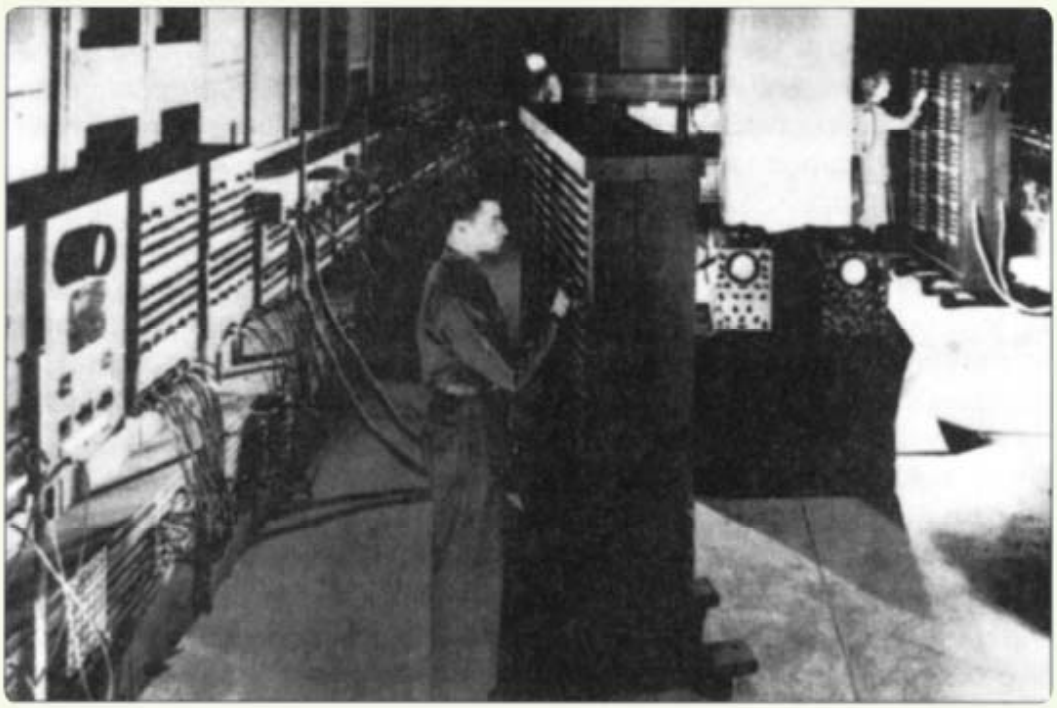
To help solve this problem, in 1943 the Army initiated a research project with J. Presper Eckert and John Mauchly of the University of Pennsylvania to build a completely electronic computing device. The machine, dubbed the ENIAC (Electronic Numerical Integrator and Calculator), was completed

in 1946 (too late to assist in the war effort) and was the first fully electronic general-purpose programmable computer. This pioneering machine is shown in Figure 1.7.

ENIAC contained 18,000 vacuum tubes and nearly filled a building; it was 100 feet long and 10 feet high and weighed 30 tons. Because it was fully electronic, it did not contain any of the slow mechanical components found in Mark I, and it executed instructions much more rapidly. The ENIAC could add two 10-digit numbers in about 1/5,000 of a second and could multiply two numbers in 1/300 of a second, one thousand times faster than the Mark I.

The Mark I and ENIAC are two well-known examples of early computers, but they are by no means the only ones of that era. For example, the ABC system (Atanasoff-Berry Computer), designed and built by Professor John Atanasoff and his graduate student Clifford Berry at Iowa State University, was actually the first electronic computer, constructed during the period 1939–1942. However, it never received equal recognition because it was useful for only one task, solving systems of simultaneous linear equations. In England, a computer called Colossus was built in 1943 under the direction of Alan Turing, a famous mathematician and computer scientist whom we will meet again in Chapter 12. This machine, one of the first computers built outside the United States, was used to crack the famous German

FIGURE 1.7



Photograph of the ENIAC computer

Source: From the Collections of the University of Pennsylvania Archives (U.S. Army photo)

Enigma code that the Nazis believed to be unbreakable. Colossus has also not received as much recognition as ENIAC because of the secrecy that shrouded the Enigma project. Its very existence was not widely known until the late 1970s, more than 30 years after the end of World War II.

At about the same time that Colossus was taking form in England, a German engineer named Konrad Zuse was working on a computing device for the German army. The machine, code named Z1, was similar in design to the ENIAC—a programmable, general-purpose, fully electronic computing device. Fortunately for the Allied forces, the Z1 project was not completed before the end of World War II.

Although the machines just described—ABC, Mark I, ENIAC, Colossus, and Z1—were computers in the fullest sense of the word (they had memory and were programmable), they did not yet look like modern computer systems. One more step was necessary, and that step was taken in 1946 by the one individual who was most instrumental in creating the computer as we know it today, John Von Neumann.

Von Neumann was not only one of the most brilliant mathematicians who ever lived, he was also a genius in many other areas as well, including experimental physics, chemistry, economics, and computer science. Von Neumann, who taught at Princeton University, had worked with Eckert and Mauchly on the ENIAC project at the University of Pennsylvania. Even though that project was successful, he recognized a number of fundamental shortcomings in ENIAC. In 1946, he proposed a radically different computer design based on a model called the **stored program computer**. Until then, all computers were programmed *erternally* using wires, connectors, and plugboards. The memory unit stored only data, not instructions. For each different problem, users had to rewire virtually the entire computer. For example, the plugboards on the ENIAC contained 6,000 separate switches, and reprogramming the ENIAC involved specifying the new settings for all these switches—not a trivial task.

Von Neumann proposed that the instructions that control the operation of the computer be encoded as binary values and stored internally in the memory unit along with the data. To solve a new problem, instead of rewiring the machine, you would rewrite the sequence of instructions—that is, create a new program. Von Neumann invented programming as it is known today.

The model of computing proposed by Von Neumann included many other important features found on all modern computing systems, and to honor him this model of computation has come to be known as the **Von Neumann architecture**. We will study this architecture in great detail in Chapters 4 and 5.

Von Neumann's research group at the University of Pennsylvania implemented his ideas, and they built one of the first stored program computers, called EDVAC, in 1949. At about the same time, a stored program computer called EDSAC was built at Cambridge University in England under the direction of Professor Maurice Wilkes. The appearance of these machines and others like them ushered in the modern computer age. Even though they were much slower, bulkier, and less powerful than our current machines, EDVAC and EDSAC executed programs in a fashion surprisingly similar to

John Von Neumann (1903–1957)

John Von Neumann was born in Budapest, Hungary. He was a child prodigy who could divide 8-digit numbers in his head by the age of 6. He was a genius in virtually every field that he studied, including physics, economics, engineering, and mathematics. At 18, he received an award as the best mathematician in Hungary, a country known for excellence in the field, and he received his Ph.D., summa cum laude, at 21. He came to the United States in 1930 to be a guest lecturer at Princeton University and taught there for 3 years. Then, in 1933 he became one of the founding members (along with Albert Einstein) of the Institute for Advanced Studies, where he worked for 20 years.

Source: Los Alamos National Laboratory

He was one of the most brilliant minds of the twentieth century, a true genius in every sense, both good and bad. He could do prodigious mental feats in his head, and his thought processes usually raced far ahead of "ordinary" mortals, who found him quite difficult to work with. One of his colleagues described him as possessing the most fearsome technical intellect of the century. Another joked that "Johnny wasn't really human, but after living among them for so long, he learned to do a remarkably good imitation of one."

Von Neumann was a brilliant theoretician who did pioneering work in pure mathematics, operations research, game theory, and theoretical physics. He was also an engineer, concerned about practicalities and real-world problems, and it was this interest in applied issues that led Von Neumann to design and construct the first stored program computer. One of the early computers built by the RAND Corp. in 1953 was affectionately called "Johnniac" in his honor, although Von Neumann detested that name. Following its shutdown, it was moved to the Computer History Museum in Mountain View, California.

the miniaturized and immensely more powerful computers of the twenty-first century. A commercial model of the EDVAC, called UNIVAC I—the first computer actually sold—was built by Eckert and Mauchly and delivered to the U.S. Bureau of the Census on March 31, 1951. (It ran for 12 years before it was retired, shut off for the last time, and moved to the Smithsonian Institution.) This date marks the true beginning of the "computer age."

The importance of Von Neumann's contributions to computer systems development cannot be overstated. Although his original proposals are about 70 years old, virtually every computer built today is a Von Neumann machine in its basic design. A lot has changed in computing, and a sleek new iPad and the bulky EDVAC would appear to have little in common. However, the basic principles on which these two very disparate machines

are constructed are virtually identical, and the same theoretical model underlies their operation. There is an old saying in computer science: "There is nothing new since Von Neumann!" This saying is certainly not true (much *has* happened), but it demonstrates the importance and amazing staying power of Von Neumann's original design.

1.4.3 The Modern Era: 1950 to the Present

The last 65 or so years of computer development have involved taking the Von Neumann architecture and improving it in terms of hardware and software. Since 1950, computer systems development has been primarily an *evolutionary*

And the Verdict Is . . .

Our discussion of what was happening in computing from 1939 to 1946 showed that many groups were involved in designing and building the first computers. Therefore, it would seem that no single individual can be credited with the title "Inventor of the Electronic Digital Computer."

Surprisingly, that is not true. In February 1964, the Sperry Rand Corp. (now UNISYS) was granted a U.S. patent on the ENIAC computer as the first fully electronic computing device, with J. Presper Eckert and John Mauchly listed as its designers and builders. However, in 1967 a suit was filed in U.S. District Court in Minneapolis, Minnesota, to overturn that patent. The suit, *Honeywell v. Sperry Rand*, was heard before U.S. Federal Judge Earl Larson, and on October 19, 1973, Judge Larson handed down his verdict. (This enormously important verdict was never given the media coverage it deserved because it happened in the middle of the Watergate hearings and on the very day that Vice President Spiro Agnew resigned in disgrace for tax fraud.) Judge Larson overturned the ENIAC patent on the basis that Eckert and Mauchly had been significantly influenced in their 1943–1944 work on ENIAC by earlier research and development work by John Atanasoff at Iowa State University. During the period 1939–1943, Mauchly had communicated extensively with Atanasoff and had even traveled to Iowa to see the ABC machine in person. In a sense, the verdict declared that Professor Atanasoff was the inventor of the electronic digital computer. This decision was never appealed. Therefore, the official honor of having designed and built the first computer, at least in U.S. District Court, goes to Professor John Vincent Atanasoff.

On November 13, 1990, in a formal ceremony at the White House, President George H.W. Bush awarded Professor Atanasoff the National Medal of Technology for his pioneering contributions to the development of the computer.

process, not a revolutionary one. The enormous number of changes in computers in recent decades has made them faster, smaller, cheaper, more reliable, and easier to use but has not drastically altered their basic underlying structure.

The period 1950–1957 (these dates are very rough approximations) is often called the *first generation* of computing. This era saw the appearance of UNIVAC I, the first computer built for sale, and the IBM 701, the first computer built by the company that would soon become a leader in this new field. These early systems were similar in design to EDVAC, and they were bulky, expensive, slow, and unreliable. They used vacuum tubes for processing and storage, and they were extremely difficult to maintain. The simple act of turning on the machine could blow out a dozen tubes! For this reason, first-generation machines were used only by trained personnel and only in specialized locations such as large corporations, government and university research labs, and military installations, which could provide this expensive support environment.

The *second generation* of computing, roughly 1957–1965, heralded a major change in the size and complexity of computers. In the late 1950s, the bulky vacuum tube was replaced by a single transistor only a few millimeters in size, and memory was now constructed using tiny magnetic cores only $1/50^{th}$ of an inch in diameter. (We will introduce and describe both devices in Chapter 4.) These technologies not only dramatically reduced the size of computers but also increased their reliability and reduced costs. Suddenly, buying and using a computer became a real possibility for some small and medium-sized businesses, colleges, and government agencies. This was also the era of the appearance of FORTRAN and COBOL, the first high-level (English-like) programming languages. (We will study this type of programming language in Chapters 9 and 10.) Now it was no longer necessary to be an electrical engineer to solve a problem on a computer. One simply needed to learn how to write commands in a high-level language. The occupation called *programmer* was born.

This miniaturization process continued into the *third generation* of computing, which lasted from about 1965 to 1975. This was the era of the *integrated circuit*. Rather than using discrete electronic components, integrated circuits with transistors, resistors, and capacitors were photographically etched onto a piece of silicon, which further reduced the size and cost of computers. From building-sized to room-sized, computers now became desk-sized, and this period saw the birth of the first minicomputer—the PDP-1 manufactured by the Digital Equipment Corp. It also saw the birth of the *software industry,* as companies sprang up to provide programs such as accounting packages and statistical programs to the ever-increasing numbers of computer users. By the mid-1970s, computers were no longer a rarity. They were being widely used throughout industry, government, the military, and education.

The *fourth generation,* roughly 1975–1985, saw the appearance of the first microcomputer. Integrated circuit technology had advanced to the point that a complete computer system could be contained on a single circuit board that you could hold in your hand. The desk-sized machine of the early 1970s now became a desktop machine, shrinking to the size of a typewriter. The Altair 8800, the world's first microcomputer, appeared in January 1975 (see the Special Interest Box on the next page).

The World's First Microcomputer

The Altair 8800, shown below, was the first microcomputer and made its debut on the cover of *Popular Electronics* in January 1975. Its developer, Ed Roberts, owned a tiny electronics store in Albuquerque, New Mexico. His company was in desperate financial shape when he read about a new microprocessor from Intel, the Intel 8080. Roberts reasoned that this new chip could be used to sell a complete personal computer in kit form. He bought these new chips from Intel at the bargain basement price of $75 each and packaged them in a kit called the Altair 8800 (named after a location in the TV series Star Trek), which he offered to hobbyists for $397. Roberts figured he might sell a few hundred kits a year, enough to keep his company afloat temporarily. He ended up selling hundreds of them per day! The Altair microcomputer kits were so popular that he could not keep them in stock, and legend has it that people even drove to New Mexico and camped out in the parking lot to buy their computers.

This is particularly amazing in view of the fact that the original Altair was difficult to assemble and had only 256 memory cells, no I/O devices, and no software support. To program it, the user had to enter binary machine language instructions directly from the console switches. But even though it could do very little, people loved it because it was a real computer, and it was theirs.

Source: University of Hawai'i at Hilo Graphics Services

The Intel 8080 chip did have the capability of running programs written in the language called BASIC that had been developed at Dartmouth in the early 1960s. A small software company located in Washington state wrote Ed Roberts a letter telling him that it had a BASIC compiler that could run on his Altair, making it much easier to use. That company was called Microsoft—and, as they say, the rest is history.

It soon became unusual *not* to see a computer on someone's desk. The software industry poured forth all types of new packages—spreadsheets, databases, word processors, and presentation graphics—to meet the needs of the burgeoning user population. This era saw the appearance of the first *computer networks*, as users realized that much of the power of computers lies in their facilitation of communication with other users. (We will look at networking in great detail in Chapter 7.) *Electronic mail* became an important application. Because so many users were computer novices, the concept of *user-friendly systems* emerged. This included new *graphical user interfaces* with pull-down menus, icons, and other visual aids to make computing easier and more fun. *Embedded systems*—devices that contain a computer to control their internal operation—first appeared during this generation. Computers were becoming small enough to be placed inside cars, thermostats, microwave ovens, and wristwatches.

The *fifth generation*, 1985–?, is where we are today. However, so much is changing so fast that the concept of distinct generations of computer development has outlived its usefulness. In computer science, change is now a constant companion. Some of the recent developments in computer systems include the following:

- Massively parallel processors capable of quadrillions (10^{15}) of computations per second
- Smartphones, tablets, and other types of handheld wireless devices
- High-resolution graphics for imaging, animation, movie making, video games, and virtual reality
- Powerful multimedia user interfaces incorporating sound, voice recognition, touch, photography, video, and television
- Integrated digital devices incorporating data, television, telephone, camera, fax, the Internet, and the web
- High-speed wireless communications
- Massive storage devices capable of holding 100 petabytes (10^{17}) of data
- Ubiquitous computing, in which miniature computers are embedded into cars, cameras, kitchen appliances, home heating systems, clothing, and even our bodies

In only a few decades, computers have progressed from the UNIVAC I, which cost millions of dollars, had a few thousand memory locations, and was capable of only a few thousand operations per second, to today's top-of-the-line graphics design workstations with a high-definition, flat panel monitor, trillions of memory cells, massive amounts of external storage, and enough processing power to execute tens of billions of instructions per second, all for about $1,000. Changes of this magnitude have never occurred so quickly in any other technology. If the same rate of change had occurred in the auto industry, beginning with the 1909 Model-T, today's cars would be capable of traveling at a speed of 20,000 miles per hour, would get about one million miles per gallon, and would cost about $1.00!

Figure 1.8 summarizes the major developments that occurred during each of the generations of computer development discussed in this section. And underlying all of these amazing improvements, the theoretical model describing the design and construction of computers has not changed significantly in the last 70 years.

FIGURE 1.8

Generation	Approximate Dates	Major Advances
First	1950–1957	First commercial computers
		First symbolic programming languages
		Use of binary arithmetic, vacuum tubes for storage
		Punched card input/output
Second	1957–1965	Transistors and core memories
		First disks for mass storage
		Size reduction, increased reliability, lower costs
		First high-level programming languages
		First operating systems
Third	1965–1975	Integrated circuits
		Further reduction in size and cost, increased reliability
		First minicomputers
		Time-shared operating systems
		Appearance of the software industry
		First set of computing standards for compatibility between systems
Fourth	1975–1985	Large-scale and very-large-scale integrated circuits
		Further reduction in size and cost, increased reliability
		First microcomputers
		Growth of new types of software and of the software industry
		Computer networks
		Graphical user interfaces
Fifth	1985–?	Ultra-large-scale integrated circuits
		Supercomputers and parallel processors
		Laptops, tablets, smartphones, and handheld wireless devices
		Mobile computing
		Massive external data storage devices
		Ubiquitous computing
		High-resolution graphics, visualization, virtual reality
		Worldwide networks and cloud computing
		Multimedia user interfaces
		Widespread use of digitized sound, images, and movies

Some of the major advancements in computing

However, many people feel that significant and important structural changes are on the way. At the end of Chapter 5, we will introduce models of computing that are fundamentally quite different from the Von Neumann architecture in use today. These totally new approaches (e.g., quantum computing) may be the models used in the twenty-second century and beyond.

1.5 Organization of the Text

This book is divided into six separate sections, called levels, each of which addresses one aspect of the definition of computer science that appears at the beginning of this chapter. Let's repeat the definition and see how it maps into the sequence of topics to be presented.

Computer science is the study of algorithms, including

1. *Their formal and mathematical properties.* Level 1 of the text (Chapters 2 and 3) is titled "The Algorithmic Foundations of Computer Science." It continues the discussion of algorithmic problem solving begun in Sections 1.2 and 1.3 by introducing important mathematical and logical properties of algorithms. Chapter 2 presents the development of several algorithms that solve important technical problems–certainly more "technical" than shampooing your hair. It also looks at concepts related to the problem-solving process, such as how we discover and create good algorithms, what notation we can use to express our solutions, and how we can check to see whether our proposed algorithm correctly solves the desired problem.

   Our brute force chess example illustrates that it is not enough simply to develop a correct algorithm; we also want a solution that is efficient and that produces the desired result in a reasonable amount of time. (Would you want to market a chess-playing program that takes 10^{25} years to make its first move?) Chapter 3 describes ways to compare the efficiency of different algorithms and select the best one to solve a given problem. The material in Level 1 provides the necessary foundation for a study of the discipline of computer science.

2. *Their hardware realizations.* Although our initial look at computer science investigated how an algorithm behaved when executed by some abstract "computing agent," we ultimately want to execute our algorithms on "real" machines to get "real" answers. Level 2 of the text (Chapters 4 and 5) is titled "The Hardware World," and it looks at how to design and construct computer systems. It approaches this topic from two quite different viewpoints.

   Chapter 4 presents a detailed discussion of the underlying hardware. It introduces the basic building blocks of computers–binary numbers, transistors, logic gates, and circuits–and shows how these elementary electronic devices can be used to construct components to perform arithmetic and logic functions such as addition, subtraction, comparison, and sequencing. Although it is both interesting and important, this perspective produces a rather low-level view of a computer system. It is difficult to

DEFINITION

Computer science: the study of algorithms, including

1. Their formal and mathematical properties
2. Their hardware realizations
3. Their linguistic realizations
4. Their applications

understand how a computer works by studying only these elementary components, just as it would be difficult to understand human behavior by investigating the behavior of individual cells. Therefore, Chapter 5 takes a higher-level view of computer hardware. It looks at computers not as a bunch of wires and circuits but as an integrated collection of subsystems called memory, processor, storage, input/output, and communications. It will explain in great detail the principles of the Von Neumann architecture introduced in Section 1.4.

A study of computer systems can be done at an even higher level. To understand how a computer works, we do not need to examine the functioning of every one of the thousands of components inside a machine. Instead, we need only be aware of a few critical pieces that are essential to our work. From the user's perspective, everything else is superfluous. This "user-oriented" view of a computer system and its resources is called a virtual machine or a virtual environment. A virtual machine is composed only of the resources that the user perceives rather than of all the hardware resources that actually exist.

This viewpoint is analogous to our level of understanding of what happens under the hood of a car. There may be thousands of mechanical components inside an automobile engine, but most of us concern ourselves only with the items reported on the dashboard—for example, oil pressure, fuel level, engine temperature. This is our "virtual engine," and that is all we need or want to know. We are all too happy to leave the remaining details about engine design to our friendly neighborhood mechanic.

Level 3 (Chapters 6, 7, and 8), titled "The Virtual Machine," describes how a virtual environment is created using a component called *system software*. Chapter 6 takes a look at the most important and widely used piece of system software on a modern computer system, the *operating system*, which controls the overall operation of a computer and makes it easier for users to access. Chapter 7 then goes on to describe how this virtual environment can extend beyond the boundaries of a single system as it examines how to interconnect individual machines into *computer networks* and *distributed systems* that provide users with access to a huge collection of computer systems and information as well as an enormous number of other users. It is the system software, and the virtual machine it creates, that makes computer hardware manageable and usable. Finally, Chapter 8 discusses a critically important component of a virtual machine—the *security system* that validates who you are and ensures that you are not attempting to carry out an improper, illegal, or unsafe operation. As computers become central to the management of such sensitive data as medical records, military information, and financial data, this aspect of system software is taking on even greater importance.

3. *Their linguistic realizations.* After studying hardware design, computer organization, and virtual machines, you will have a good idea of the techniques used to design and build computers. In the next section of the text, we ask the question, how can this hardware be used to solve important and interesting problems? Level 4, titled "The Software World"

(Chapters 9–12), takes a look at what is involved in designing and implementing computer software. It investigates the programs and instruction sequences executed by the hardware, rather than the hardware itself.

Chapter 9 compares several high-level programming languages and introduces fundamental concepts related to the topic of computer programming regardless of the particular language being studied. This single chapter is certainly not intended to make you a proficient programmer, and this book is not meant to be a programming text. Instead, its purpose is to illustrate some basic features of modern programming languages and give you an appreciation for the interesting and challenging task of the computer programmer. Rather than print a separate version of this text for each programming language, the textual material specific to each language can be found on the website for this text, and you can download the pages for the language specified by your instructor and used in your class. See the Preface of this text for instructions on accessing these webpages.

There are many programming languages, such as C++, Python, Java, and Perl, that can be used to encode algorithms. Chapter 10 provides an overview of a number of different languages and language models in current use, including the functional and parallel models. Chapter 11 describes how a program written in a high-level programming language can be translated into the low-level machine language codes first described in Chapter 5. Finally, Chapter 12 shows that, even when we marshal all the powerful hardware and software ideas described in the first 11 chapters, problems exist that cannot be solved algorithmically. Chapter 12 demonstrates that there are, indeed, limits to computing.

4. *Their applications*. Most people are concerned not with creating programs but with using programs, just as there are few automotive engineers but many, many drivers. Level 5, titled "Applications" (Chapters 13–16), moves from *how* to write a program to *what* these programs can do.

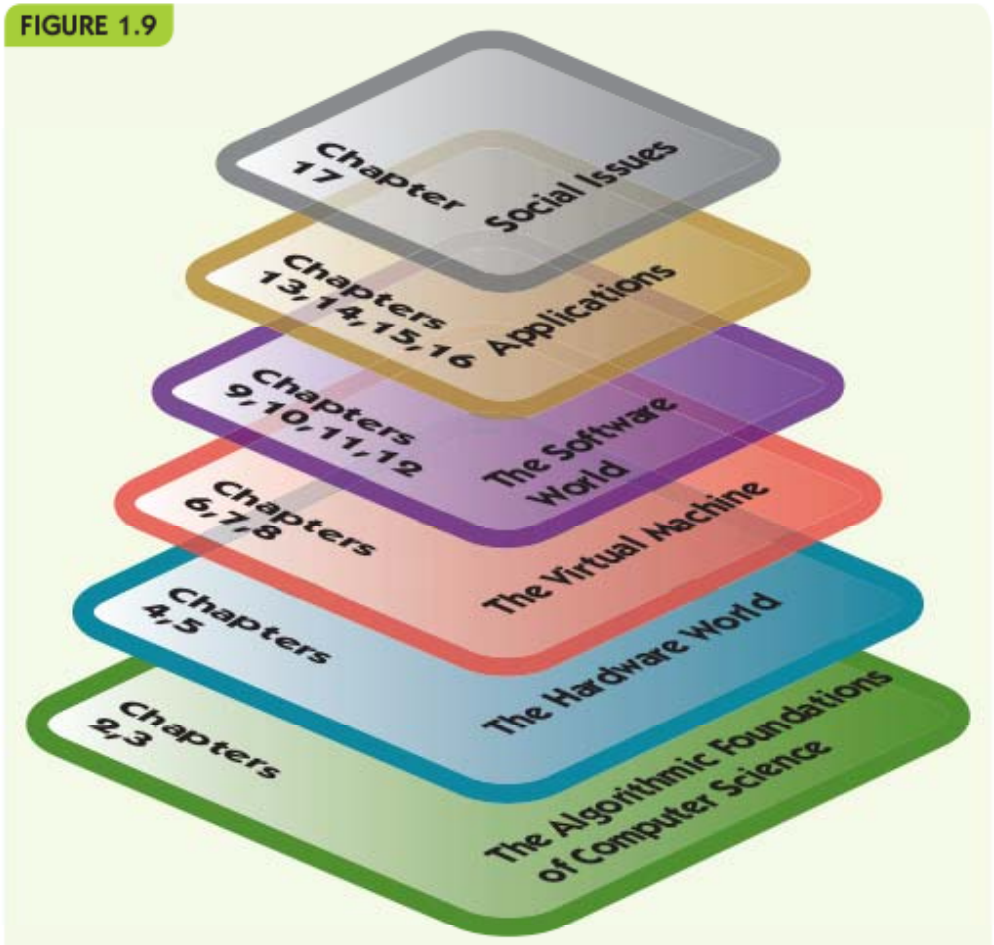
   Chapters 13 through 16 explore just a few of the many important and rapidly growing applications of computers, such as simulation, visualization, ecommerce, databases, artificial intelligence, computer graphics, and entertainment. Amazing as these applications seem, underneath their "shiny covers" they rely on the computer concepts of earlier chapters. Of course, we cannot possibly survey all the ways in which computers are being used today or will be used in the future. Indeed, there is hardly an area in our modern, complex society that has not been affected in some important way by information technology. (An excellent demonstration of this fact is that there are now more than one million apps available for downloading from the Apple App Store.) Readers interested in applications not discussed here should seek readings specific to their own areas of interest.

   Some computer science professionals are not concerned with building computers, creating programs, or using any of the applications just described. Instead, they are interested in the social and cultural impact—both positive and negative—of this ever-changing technology. The sixth level of this text addresses this important perspective on computer science. This is not part of the original definition of computer science but has become

an important area of study. In Level 6, titled "Social Issues" (Chapter 17), we move to the highest level of abstraction—the view furthest removed from the computer itself—to discuss social, ethical, legal, and professional issues related to computer and information technology. These issues are critically important because even individuals not directly involved in developing or using computers are deeply affected by them, just as society has been drastically and permanently altered by such technological developments as telephones, televisions, automobiles, and nuclear power. This last chapter takes a look at such thorny and difficult topics as computer crime, information privacy, and intellectual property. It also looks at one of the most important phenomena supported by this new technology, the creation of social networks such as Facebook, Twitter, LinkedIn, and Pinterest. Because it is impossible to resolve all the complex questions that arise in these areas, our intent is simply to raise your awareness and provide some decision-making tools to help you reach your own conclusions.

The overall six-layer hierarchy of this text is summarized in Figure 1.9. The organizational structure diagrammed in Figure 1.9 is one of the most

FIGURE 1.9



Organization of the text into a six-layer hierarchy

important aspects of this text. To describe a field of study, it is not enough to present a mass of facts and explanations. For learners to absorb, understand, and integrate this information, there must be a theme, a relationship, a thread that ties together the various parts of the narrative—in essence, a "big picture." Our big picture is Figure 1.9.

We first lay out the basic foundations of computer science (Level 1). We then proceed upward through five distinct layers of abstraction, from extremely low-level machine details such as electronic circuits and computer hardware (Level 2), through intermediate levels that address virtual machines (Level 3) and programming languages and software development (Level 4), to higher levels that investigate computer applications (Level 5) and address the use and misuse of information technology (Level 6). The material in each level provides a foundation to reveal the beauty and complexity of a higher and more abstract view of the discipline of computer science.

Laboratory Experience 1

Associated with this text is a laboratory manual that includes software packages and a collection of formal laboratory exercises. These Laboratory Experiences are designed to give you a chance to build on, modify, and experiment with the ideas discussed in the text. You are strongly encouraged to carry out these laboratories to gain a deeper understanding of the concepts presented in the chapters. Learning computer science involves not just reading and listening but also doing and trying. Our laboratory exercises will give you that chance. (In addition, we hope that you will find them fun.)

Laboratory Experience 1, titled "Building A Glossary," reviews the fundamental operations that you will need in all future labs—operations such as using menus, buttons, and windows. In addition, the lab provides a useful tool that you may use during your study of computer science and in other courses as well. You will learn how to build a glossary of important technical terms along with their definitions and locations in the text.

Please open Laboratory Experience 1 and try it now.

EXERCISES

1. Identify some algorithms, apart from DVR instructions and cooking recipes, that you encounter in your everyday life. Write them out in any convenient notation, and explain how they meet all of the criteria for algorithms presented in this chapter.

2. A concept related, but not identical, to an algorithm is the idea of a *heuristic*. Read about heuristics and identify differences between the two. Describe a heuristic for obtaining an approximate answer to the sum of two 3-digit numbers and show how this "addition heuristic" differs from the addition algorithm of Figure 1.2.

3. In the DVR instructions in Figure 1.1, Step 3 says, "Enter the channel number that you want to record and press the button labeled CHAN." Is that an unambiguous and well-defined operation? Explain why or why not.

4. Identify which type of algorithmic operation each one of the following steps belongs to:

   a. Get a value for x from the user.

   b. Test to determine if x is positive. If not, tell the user that he or she has made a mistake.

   c. Take the cube root of x.

   d. Do Steps 1.1, 1.2, and 1.3 x times.

5. Trace through the decimal addition algorithm of Figure 1.2 using the following input values:

$$m = 3 \quad a_2 = 1 \quad a_1 = 4 \quad a_0 = 9$$
$$b_2 = 0 \quad b_1 = 2 \quad b_0 = 9$$

   At each step, show the values for c_3, c_2, c_1, c_0, and *carry*.

6. Modify the decimal addition algorithm of Figure 1.2 so that it does not print out nonsignificant leading zeroes; that is, the answer to Exercise 5 would appear as 178 rather than 0178.

7. Modify the decimal addition algorithm of Figure 1.2 so that the two numbers being added need not have the same number of digits. That is, the algorithm should be able to add a value a containing m digits to a value b containing n digits, where m may or may not be equal to n.

8. Under what conditions would the well-known quadratic formula

$$Roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   not be effectively computable? (Assume that you are working with real numbers.)

9. Compare the two solutions to the shampooing algorithm shown in Figures 1.3 and 1.4. Which do you think is a better general-purpose solution? Why? (*Hint:* What if you wanted to wash your hair 1,000 times?)

10. The following is Euclid's 2,300-year-old algorithm for finding the greatest common divisor of two positive integers I and J.

Step	*Operation*
1	Get two positive integers as input; call the larger value I and the smaller value J
2	Divide I by J, and call the remainder R
3	If R is *not* 0, then reset I to the value of J, reset J to the value of R, and go back to Step 2
4	Print out the answer, which is the value of J
5	Stop

   a. Go through this algorithm using the input values 20 and 32. After each step of the algorithm is completed, give the values of I, J, and R. Determine the final output of the algorithm.

   b. Does the algorithm work correctly when the two inputs are 0 and 32? Describe exactly what happens, and modify the algorithm so that it gives an appropriate error message.

11. A salesperson wants to visit 25 cities while minimizing the total number of miles she must drive. Because she has studied computer science, she decides to design an algorithm to determine the optimal order in which to visit the cities to (1) keep her driving distance to a minimum, and (2) visit each city exactly once. The algorithm that she has devised is the following:

    > The computer first lists *all* possible ways to visit the 25 cities and then, for each one, determines the total mileage associated with that particular ordering. (Assume that the computer has access to data that gives the distances between all cities.) After determining the total mileage for each possible trip, the computer searches for the ordering with the minimum mileage and prints out the list of cities on that optimal route, that is, the order in which the salesperson should visit her destinations.

    If a computer could analyze 10,000,000 separate paths per second, how long would it take to determine the optimal route for visiting these 25 cities? On the basis of your answer, do you think this is a feasible algorithm? If it is not, can you think of a way to obtain a reasonable solution to this problem?

12. One way to do multiplication is by repeated addition. For example, 47 x 25 can be evaluated as 47 + 47 + 47 + ... + 47 (25 times). Sketch out an algorithm for multiplying two positive numbers *a* and *b* using this technique.

13. A student was asked to develop an algorithm to find and output the largest of three numerical values *x*, *y*, and *z* that are provided as input. Here is what was produced:

    Input: *x*, *y*, *z*

    Algorithm: Check if ($x > y$) and ($x > z$). If it is, then output the value of *x* and stop. Otherwise, continue to the next line.

    Check if ($y > x$) and ($y > z$). If it is, then output the value of *y* and stop. Otherwise, continue to the next line.

    Check if ($z > x$) and ($z > y$). If it is, then output the value of *z* and stop.

    Is this a correct solution to the problem? Explain why or why not. If it is incorrect, fix the algorithm so that it is a correct solution.

14. Read about one of the early pioneers mentioned in this chapter—Pascal, Liebnitz, Jacquard, Babbage, Lovelace, Hollerith, Eckert, Mauchly, Aiken, Zuse, Atanasoff, Turing, or Von Neumann. Write a paper describing in detail that person's contribution to computing and computer science.

15. Get the technical specifications of the computer on which you are working (either from a technical manual or from your computer center staff). Determine its cost, its processing speed (in MIPS, millions of instructions per second), its computational speed (in GFlops, billions of floating point operations per second), and the size of its primary memory.

    Compare those values with what was typically available on first-, second-, and third-generation computer systems, and calculate the percentage improvement between your computer and the first commercial machines of the early 1950s.

16. A new and growing area of computer science is *ubiquitous computing*, in which computers automatically provide services for a user without that user's knowledge or awareness. For example, a computer located in your car contacts the garage door opener and tells it to open the garage door when the car is close to home. Read about this new model of computing and write a paper describing some of its applications. What are some of the possible problems that could result?

17. Another important new area of computer science is *cloud computing*, which relies on a computer network, along with networking software, to provide transparent access to remote data and applications. Read about this new model of data and software access and write a paper describing some of the important uses, as well as potential risks, of this new information structure.

18. A standard computer DVD holds approximately 5 billion characters. Estimate how many linear feet of shelf space would be required to house 5 billion characters encoded as printed bound books rather than as electronic media. Assume there are 5 characters per word, 300 words per page, and 300 pages per inch of shelf.

CHALLENGE WORK

1. Assume we have a "computing agent" that knows how to do one-digit subtraction where the first digit is at least as large as the second (i.e., we do not end up with a negative number). Thus, our computing agent can do such operations as $7 - 3 = 4$, $9 - 1 = 8$, and $5 - 5 = 0$. It can also subtract a one-digit value from a two-digit value in the range 10–18 as long as the final result has only a single digit. This capability enables it to do such operations as $13 - 7 = 6$, $10 - 2 = 8$, and $18 - 9 = 9$.

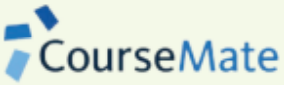
   Using these primitive capabilities, design an algorithm to do *decimal subtraction* on two m-digit numbers, where $m \geq 1$. You will be given two unsigned whole numbers ($a_{m-1}, a_{m-2} \ldots a_0$) and $b_{m-1}, b_{m-2} \ldots b_0$. Your algorithm must compute the value $c_{m-1}\, c_{m-2} \ldots c_0$, the difference of these two values.

$$\begin{array}{r} a_{m-1}a_{m-2}\ldots a_0 \\ -\,b_{m-1}b_{m-2}\ldots b_0 \\ \hline c_{m-1}c_{m-2}\ldots c_0 \end{array}$$

   You may assume that the top number ($a_{m-1}\, a_{m-2} \ldots a_0$) is greater than or equal to the bottom number ($b_{m-1}\, b_{m-2} \ldots b_0$) so that the result is not a negative value. However, do not assume that each individual digit a_i is greater than or equal to b_i. If the digit on the bottom is larger than the digit on the top, then you must implement a *borrowing scheme* to allow the subtraction to continue. (*Caution:* It may have been easy to learn subtraction as a first grader, but it is devilishly difficult to tell a computer how to do it!)

2. Our definition of the field of computer science is only one of many that have been proposed. Because it is so young, people working in the field are still debating how best to define exactly what they do. Review the literature of computer science (see the CourseMate for this text and chapter for some ideas) and browse the web to locate other definitions of computer science. Compare these definitions with the one presented in this chapter and discuss the differences among them. Discuss how different definitions may give you a vastly different perspective on the field and what people in this field do. [*Note:* A very well-known and widely used definition of computer science was presented in "Report of the ACM Task Force on the Core of Computer Science," reprinted in the journal *Communications of the ACM*, vol. 32, no. 1 (January 1989).]

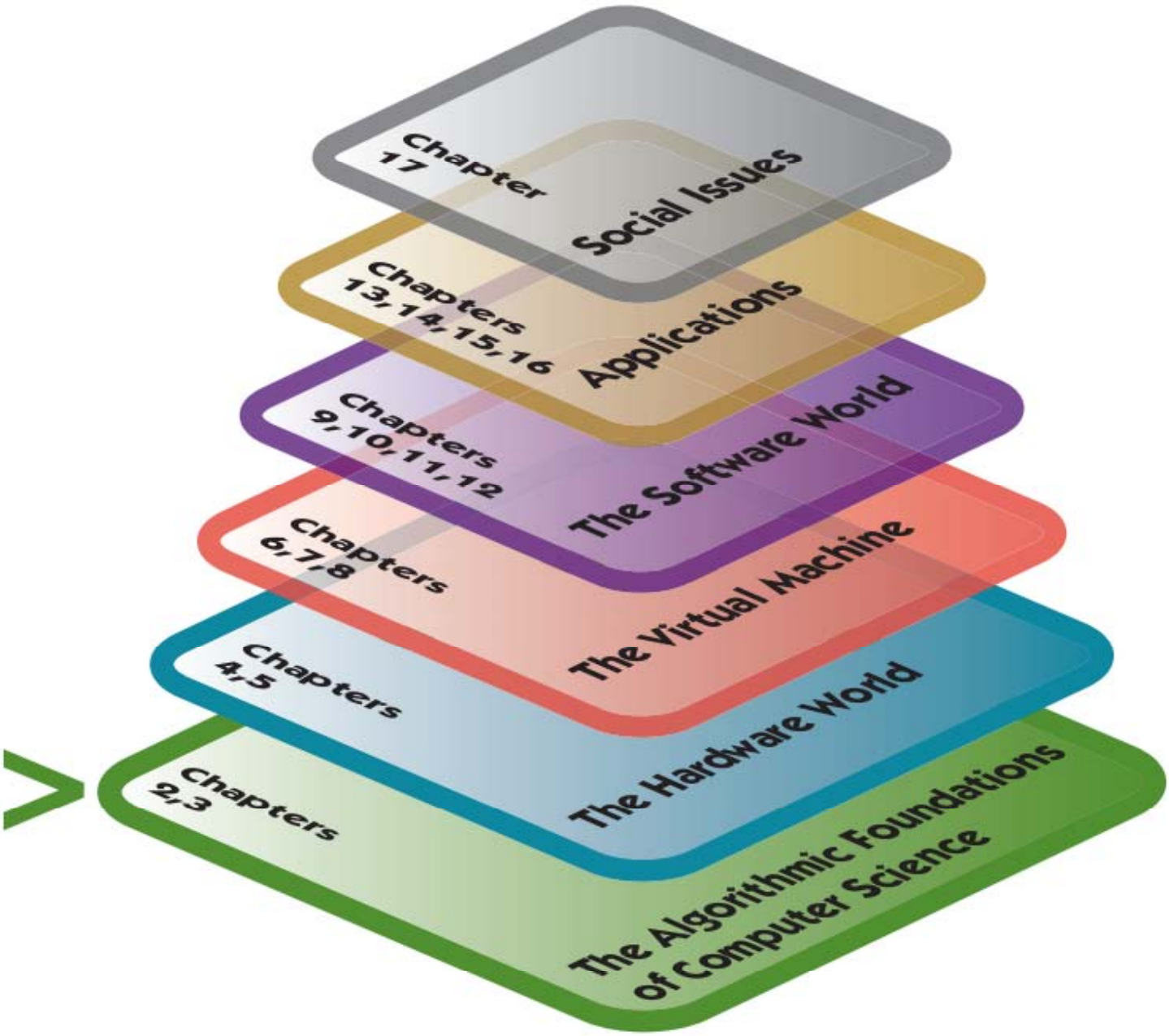
ADDITIONAL RESOURCES



For additional print and/or online resources relevant to this chapter, visit the CourseMate for this text at login.cengagebrain.com.

LEVEL

1 The Algorithmic Foundations of Computer Science



Level 1 of the text continues our exploration of algorithms and algorithmic problem solving—essential material for studying any branch of computer science. It first introduces methods for designing and representing algorithms. It then uses these ideas to develop solutions to some real-world problems, including an important application in medicine and biology.

When judging the quality of an essay or book report, we do not look only at sentence structure, spelling, and punctuation. Although grammatical issues are important, we also must evaluate the work's style, for it is a combination of correctness and expressiveness that produces a written document of high quality. So, too, for algorithms: Correctness is not the only measure of excellence. This section will present criteria for evaluating the quality and elegance of the algorithmic solutions that you develop.

CHAPTER 2

Algorithm Discovery and Design

CHAPTER TOPICS

2.1 Introduction

Chapter 1 introduced algorithms and algorithmic problem solving, two of the most fundamental concepts in computer science. Our introduction used examples drawn from everyday life, such as programming a DVR (Figure 1.1) and washing your hair (Figures 1.3 and 1.4). Although these are perfectly valid examples of algorithms, they are not of much interest to computer scientists. This chapter develops more fully the notions of algorithms and algorithmic problem solving and applies these ideas to problems that *are* of interest to computer scientists: searching lists, finding maxima and minima, and matching patterns.

2.2 Representing Algorithms

2.2.1 Pseudocode

Before presenting any algorithms, we must first make an important decision. How should we represent them? What notation should we use to express our algorithms so that they are clear, precise, and unambiguous?

One possibility is *natural language*, the language we speak and write in our everyday lives. (This could be English, Spanish, Arabic, Japanese, Swahili, or any language.) This is an obvious choice because it is the language with which we are most familiar. If we use natural language, then our algorithms would read much the same as a term paper or an essay. For example, when expressed in natural language, the addition algorithm in Figure 1.2 might look something like the paragraph shown in Figure 2.1.

Comparing Figure 1.2 with Figure 2.1 illustrates the problems of using natural language to represent algorithms. Natural language can be extremely verbose, causing the resulting algorithms to be rambling, unstructured, and hard to follow. (Imagine reading 5, 10, or even 100 pages of text like Figure 2.1.) An unstructured, "free-flowing" writing style might be wonderful for novels and essays, but it is horrible for algorithms. The lack of structure makes it difficult for the reader to locate specific sections of the algorithm because they are buried deep within the text. For example, about two-thirds of the way through Figure 2.1 is the phrase, "... and begin the

FIGURE 2.1

Initially, set the value of the variable *carry* to 0 and the value of the variable *i* to 0. When these initializations have been completed, begin looping as long as the value of the variable *i* is less than or equal to $(m - 1)$. First, add together the values of the two digits a_i and b_i and the current value of the carry digit to get the result called c_i. Now check the value of c_i to see whether it is greater than or equal to 10. If c_i is greater than or equal to 10, then reset the value of *carry* to 1 and reduce the value of c_i by 10; otherwise, set the value of *carry* to 0. When you are finished with that operation, add 1 to *i* and begin the loop all over again. When the loop has completed execution, set the leftmost digit of the result c_m to the value of *carry* and print out the final result, which consists of the digits $c_m\, c_{m-1} \ldots c_0$. After printing the result, the algorithm is finished, and it terminates.

The addition algorithm of Figure 1.2 expressed in natural language

loop all over again." To what part of the algorithm does this refer? Without any clues to guide us, such as indentation, line numbering, or highlighting, locating the beginning of that loop can be a daunting and time-consuming task. (For the record, the beginning of the loop corresponds to the sentence that starts, "When these initializations have been completed . . . ." It is certainly not easy to determine this from a casual reading of the text.)

A second problem is that natural language is too "rich" in interpretation and meaning. Natural language frequently relies on either context or a reader's experiences to give precise meaning to a word or phrase. This permits different readers to interpret the same sentence in totally different ways. This may be acceptable, even desirable, when writing poetry or fiction, but it is disastrous when creating algorithms that must always execute in the same way and produce identical results. We can see an example of this problem in the sentence of Figure 2.1 that starts with "When you are finished with that operation . . . ." When we are finished with *which* operation? It is not at all clear from the text, and individuals might interpret

the phrase *that operation* in different ways, producing radically different behavior. Similarly, the statement "Determine the shortest path between the source and destination" is ambiguous until we know the precise meaning of the phrase "shortest path." Does it mean shortest in terms of travel time, distance, or something else?

Because natural languages are not sufficiently precise to represent algorithms, we might be tempted to go to the other extreme. If we are ultimately going to execute our algorithm on a computer, why not immediately write it out as a computer program using a *high-level programming language* such as C++ or Java? If we adopt that approach, the addition algorithm of Figure 1.2 might start out looking like the program fragment shown in Figure 2.2.

As an algorithmic design language, this notation is also seriously flawed. During the initial phases of design, we should be thinking at a highly abstract level. However, using a formal programming language to express our design forces us to deal immediately with highly detailed language issues, such as punctuation, grammar, and syntax. For example, the algorithm in Figure 1.2 contains an operation that says, "Set the value of *carry* to 0." This is an easy statement to understand. However, when translated into a language like C++ or Java, that statement becomes

```
carry = 0;
```

FIGURE 2.2

```
{
Scanner inp = new Scanner(System.in);
int i, m, carry;
int[] a = new int[100];
int[] b = new int[100];
int[] c = new int[100];
m = inp.nextInt();
for (int j = 0;j <= m-1;j++) {
       a[j] = inp.nextInt();
       b[j] = inp.nextInt();
}
carry = 0;
i = 0;
while (i < m) {
    c[i] = a[i] + b[i] + carry;
    if (c[i] >= 10)
              .
              .
              .
```

The beginning of the addition algorithm of Figure 1.2 expressed in a high-level programming language

Is this operation setting *carry* to 0 or asking if *carry* is equal to 0? Why does a semicolon appear at the end of the line? Would the statement

```
Carry = 0;
```

mean the same thing? Similarly, what is meant by the utterly cryptic statement on Line 4 of Figure 2.2: `int[] a = new int[100];`? These technical details clutter our thoughts and at this point in the solution process are totally out of place. When creating algorithms, a programmer should no more worry about semicolons and capitalization than a novelist should worry about typography and cover design when writing the first draft!

If the two extremes of natural languages and high-level programming languages are both less than ideal, what notation should we use? What is the best way to represent the solutions shown in this chapter and the rest of the book?

Most computer scientists use a notation called **pseudocode** to design and represent algorithms. This is a set of English-language constructs designed to more or less resemble statements in a programming language but that do not actually run on a computer. Pseudocode represents a compromise between the two extremes of natural and formal languages. It is simple, highly readable, and has virtually no grammatical rules. (In fact, pseudocode is sometimes jokingly referred to as "a programming language without the details.") However, because it contains only statements that have a well-defined structure, it is easier to visualize the organization of a pseudocode algorithm than one represented as long, rambling natural-language paragraphs. In addition, because pseudocode closely resembles many popular programming languages, the subsequent translation of the algorithm into a computer program is relatively simple. The algorithms shown in Figures 1.1, 1.2, 1.3, and 1.4, and Exercise 10 of Chapter 1 are all written in pseudocode.

In the following sections, we will introduce a set of popular and easy-to-understand constructs for the three types of algorithmic operations introduced in Chapter 1: sequential, conditional, and iterative. Keep in mind, however, that pseudocode is *not* a formal language with rigidly standardized syntactic and semantic rules and regulations. On the contrary, it is an informal design notation used solely to express algorithms. If you do not like the constructs presented in the next two sections, feel free to modify them or select others that are more helpful to you. One of the nice features of pseudocode is that you can adapt it to your own personal way of thinking and problem solving.

2.2.2 Sequential Operations

Our pseudocode must include instructions to carry out the three basic *sequential operations* called computation, input, and output.

The instruction for performing a **computation** and saving the result looks like the following. (Words and phrases inside quotation marks represent specific elements that you must insert when writing an algorithm.)

```
Set the value of "variable" to "arithmetic expression"
```

This operation evaluates the "arithmetic expression," gets a result, and stores that result in the "variable." A variable is simply a named storage location that can hold a data value. A variable is often compared with a mailbox into which you can place a value and from which you can retrieve a value. Let's look at an example.

Set the value of *carry* to 0

First, evaluate the arithmetic expression, which in this case is the constant value 0. Then store that result in the variable called *carry*. If *carry* had a previous value, say 1, that value will be discarded and replaced by 0. If *carry* did not yet have a value, it now does—the value 0. You can visualize the result of this operation as follows:

Here is another example:

Set the value of *Area* to (πr^2)

Assuming that the variable *r* has been given a value by a previous instruction in the algorithm, this statement evaluates the arithmetic expression πr^2 to produce a numerical result. This result is then stored in the variable called *Area*. If *r* does not have a value, an error condition occurs because this instruction is not effectively computable, and it cannot be completed.

We can see additional examples of computational operations in Steps 4, 6, and 7 of the addition algorithm of Figure 1.2:

Step 4: Add the two digits a_i and b_i to the current value of *carry* to get c_i

Step 6: Add 1 to *i*, effectively moving one column to the left

Step 7: Set c_m to the value of *carry*

Note that these three steps are not written in exactly the format just described. If we had used that notation, they would have looked like this:

Step 4: Set the value of c_i to $(a_i + b_i + carry)$

Step 6: Set the value of *i* to $(i + 1)$

Step 7: Set the value of c_m to *carry*

However, in pseudocode it doesn't matter exactly how you choose to write your instructions as long as the intent is clear, effectively computable, and unambiguous. At this point in the design of a solution, we do not care about the minor linguistic differences between

Add *a* and *b* to get *c*

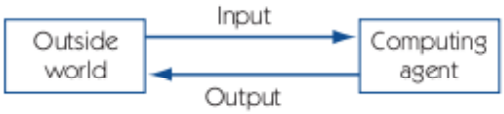
and

Set the value of *c* to $(a + b)$

Remember that pseudocode is not a precise set of notational rules to be memorized and rigidly followed. It is a flexible notation that can be adjusted to fit your own view about how best to express ideas and algorithms.

When writing arithmetic expressions, you may assume that the computing agent executing your algorithm has all the capabilities of a typical multifunction calculator. Therefore, it "knows" how to do all basic arithmetic operations such as $+$, $-$, $\times$, $\div$, $\sqrt{}$, absolute value, sine, cosine, and tangent. It also knows the value of important constants such as π.

The remaining two sequential operations enable our computing agent to communicate with "the outside world," which means everything other than the computing agent itself:



Input operations provide the computing agent with data values from the outside world that it may then use in later instructions. Output operations send results from the computing agent to the outside world. When the computing agent is a computer, communications with the outside world are done via the input/output equipment available on a typical computer, tablet, or smartphone, such as a physical keyboard, virtual keypad, screen, mouse, printer, hard drive, camera, or touch screen. However, when designing algorithms, we generally do not concern ourselves with the technical specifications of a particular device. At this point in the design process we care only that data is provided to us when we request it, and that results are issued for presentation.

Our pseudocode instructions for input and output are expressed as follows:

Input: Get values for "variable", "variable", ...

Output: Print the values of "variable", "variable", ...

For example,

Get a value for *r*, the radius of the circle

When the algorithm reaches this input operation, it waits until someone or something provides it with a value for the variable *r*. (In a computer, this may be done by entering a value at the keyboard.) When the algorithm has received and stored a value for *r*, it continues on to the next instruction.

Here is an example of an output operation:

Print the value of *Area*

Assuming that the algorithm has already computed the area of the circle, this instruction says to display that value to the outside world. This display may be viewed on a screen (computer, tablet, smartphone) or printed on paper by a printer.

Sometimes we use an output instruction to display a message in place of the desired results. If, for example, the computing agent cannot complete a computation because of an error condition, we might have it execute something like the following operation. (We will use 'single quotation marks' to enclose messages so as to distinguish them from such pseudocode constructs as "variable" and "arithmetic expression," which are enclosed in double quotation marks.)

Print the message 'Sorry, no answers could be computed.'

FIGURE 2.3

Step	Operation
1	Get values for *gallons used, starting mileage, ending mileage*
2	Set value of *distance driven* to (*ending mileage – starting mileage*)
3	Set value of *average miles per gallon* to (*distance driven ÷ gallons used*)
4	Print the value of *average miles per gallon*
5	Stop

Algorithm for computing average miles per gallon (version 1)

Using these three sequential operations—computation, input, and output—we can now write some simple but useful algorithms. Figure 2.3 presents an algorithm to compute the average miles per gallon on a trip, when given as input the number of gallons used and the starting and ending mileage readings on the odometer.

Practice Problems

Write pseudocode versions of the following:

1. An algorithm that gets three data values *x*, *y*, and *z* as input and outputs the average of those three values.
2. An algorithm that gets the radius *r* of a circle as input. Its output is both the circumference and the area of a circle of radius *r*.
3. An algorithm that gets the amount of electricity used in kilowatt-hours and the cost of electricity per kilowatt-hour. Its output is the total amount of the electric bill, including an 8% sales tax.
4. An algorithm that inputs your current credit card balance, the total dollar amount of new purchases, and the total dollar amount of all payments. The algorithm computes the new balance, which includes a 12% interest charge on any unpaid balance.
5. An algorithm that is given the length and width, in feet, of a rectangular carpet and determines its total cost given that the material cost is $23 per square yard.
6. An algorithm that is given three numbers corresponding to the number of times a race car driver has finished first, second, and third. The algorithm computes and displays how many points that driver has earned given 5 points for a first, 3 points for a second, and 1 point for a third place finish.

2.2.3 Conditional and Iterative Operations

The average miles per gallon algorithm in Figure 2.3 performs a set of operations once and then stops. It cannot select among alternative operations or perform a block of instructions more than once. A purely **sequential algorithm** of the type shown in Figure 2.3 is sometimes termed a *straight-line algorithm* because it executes its instructions in a straight line from top to bottom and then stops. Unfortunately, virtually all real-world problems are not straight-line in nature. They involve nonsequential operations such as branching and repetition.

To allow us to address these more interesting problems, our pseudocode needs two additional statements to implement conditional and iterative operations. Together, these two types of operations are called **control operations**; they allow us to alter the normal sequential flow of control in an algorithm. As we saw in Chapter 1, control operations are an essential part of all but the very simplest of algorithms.

Conditional statements are the "question-asking" operations of an algorithm. They allow an algorithm to ask a yes/no question and select the next operation to perform on the basis of the answer to that question. There are a number of ways to phrase a question, but the most common conditional statement is the *if/then/else* statement, which has the following format:

```
If "a true/false condition" is true then
    first set of algorithmic operations
Else (or otherwise)
    second set of algorithmic operations
```

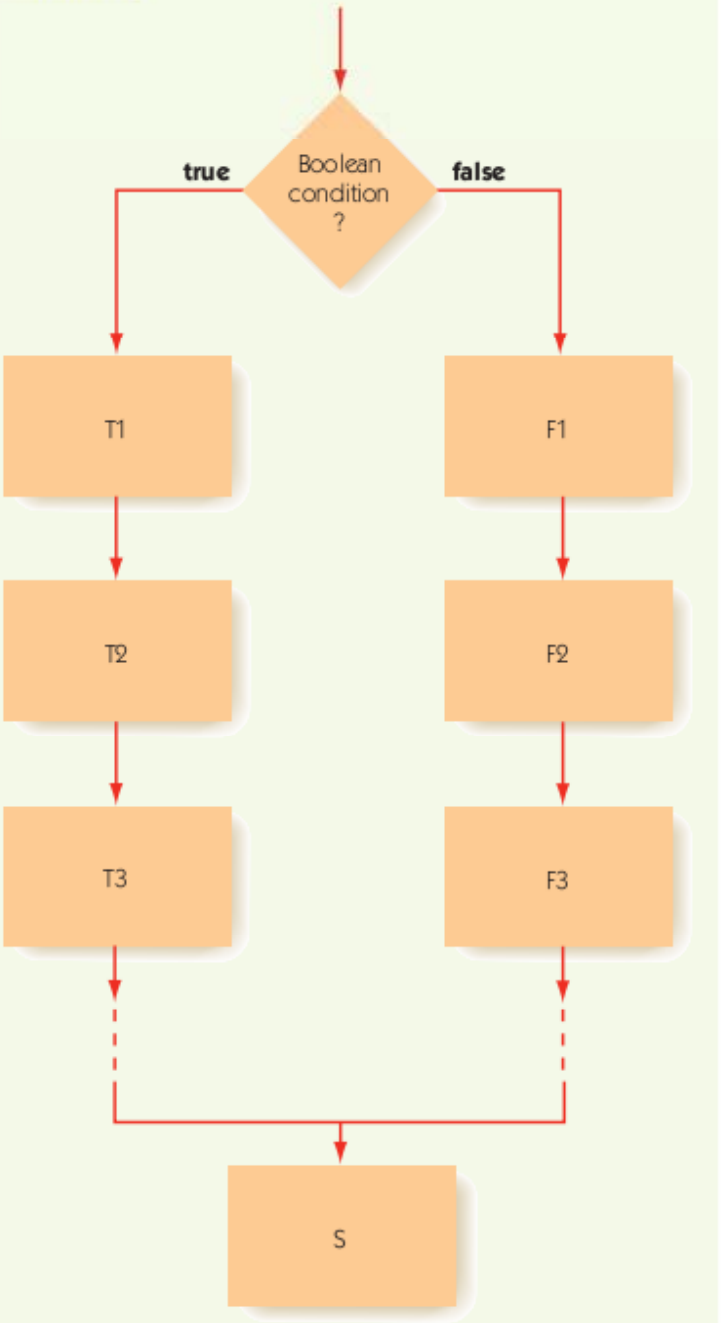
The meaning of this statement is as follows:

1. Evaluate the true/false condition on the first line to determine whether it is true or false.
2. If the condition is true, then do the first set of algorithmic operations and skip the second set entirely.
3. If the condition is false, then skip the first set of operations and do the second set.
4. Once the appropriate set of operations has been completed, continue executing the algorithm with the operation that follows the if/then/else instruction.

Figure 2.4 is a visual model of the execution of the if/then/else statement. We evaluate the condition shown in the diamond. If the condition is true, we execute the sequence of operations labeled T1, T2, T3, … . If the condition is false, we execute the sequence labeled F1, F2, F3, … . In either case, however, execution continues with statement S, which is the one that immediately follows the if/then/else statement.

Basically, the if/then/else statement allows you to select exactly one of two alternatives—either/or, this or that. We saw an example of this statement in Step 5 of the addition algorithm shown in Figure 1.2. (The statement has been reformatted slightly to highlight the two alternatives clearly, but it has not been changed.)

FIGURE 2.4



The if/then/else pseudocode statement

If ($c_j \geq 10$) then
    Set the value of c_j to ($c_j - 10$)
    Set the value of *carry* to 1
Else
    Set the value of *carry* to 0

The condition ($c_i \geq 10$) can be only true or false. If it is true, then there is a carry into the next column, and we must do the first set of instructions—subtracting 10 from c_i and setting *carry* to 1. If the condition is false, then there is no carry—we skip over these two operations and perform the second block of operations, which simply sets the value of *carry* to 0.

Figure 2.5 shows another example of the if/then/else statement. It extends the miles per gallon algorithm of Figure 2.3 to include a second line of output stating whether you are getting good gas mileage. Good gas mileage is defined as a value for average miles per gallon strictly greater than 25.0 mpg.

The last algorithmic statement to be introduced allows us to implement a *loop*—the repetition of a block of instructions. The real power of a computer comes not from doing a calculation once but from doing it many, many times. If, for example, we need to compute a single value of average miles per gallon, it would be foolish to convert an algorithm like Figure 2.5 into a computer program and execute it on a computer—it would be far faster to use a calculator, which could complete the job in a few seconds. However, if we need to do the same computation 1 million times, the power of a computer to repetitively execute a block of statements becomes quite apparent. If each computation of average miles per gallon takes 5 seconds on a hand calculator, then 1 million of them would require about 2 months, not allowing for such luxuries as sleeping and eating. Once the algorithm is developed and the program written, a computer could carry out that same task in a fraction of a second!

The first algorithmic statement that we will use to express the idea of iteration, also called *looping*, is the *while* statement:

FIGURE 2.5

Step	Operation
1	Get values for *gallons used, starting mileage, ending mileage*
2	Set value of *distance driven* to (*ending mileage – starting mileage*)
3	Set value of *average miles per gallon* to (*distance driven ÷ gallons used*)
4	Print the value of *average miles per gallon*
5	If *average miles per gallon* is >25.0 then
6	Print the message 'You are getting good gas mileage'
	Else
7	Print the message 'You are NOT getting good gas mileage'
8	Stop

Second version of the average miles per gallon algorithm

While ("a true/false condition") do Step *i* to Step *j*

Step *i*: operation

Step *i* + 1: operation

.

.

.

Step *j*: operation

This instruction initially evaluates the "true/false condition"—called the **continuation condition**—to determine if it is true or false. If the condition is true, all operations from Step *i* to Step *j*, inclusive, are executed. This block of operations is called the **loop body**. (Operations within the loop body should be indented so that it is clear to the reader of the algorithm which operations belong inside the loop.) When the entire loop body has finished executing, the algorithm again evaluates the continuation condition. If it is still true, then the algorithm executes the entire loop body, statements *i* through *j*, again. This looping process continues until the continuation condition evaluates to false, at which point execution of the loop body terminates and the algorithm proceeds to the statement immediately following the loop—Step *j* + 1 in the above pseudocode. If for some reason the continuation condition never becomes false, then we have violated one of the fundamental properties of an algorithm, and we have the error, first mentioned in Chapter 1, called an *infinite loop*.

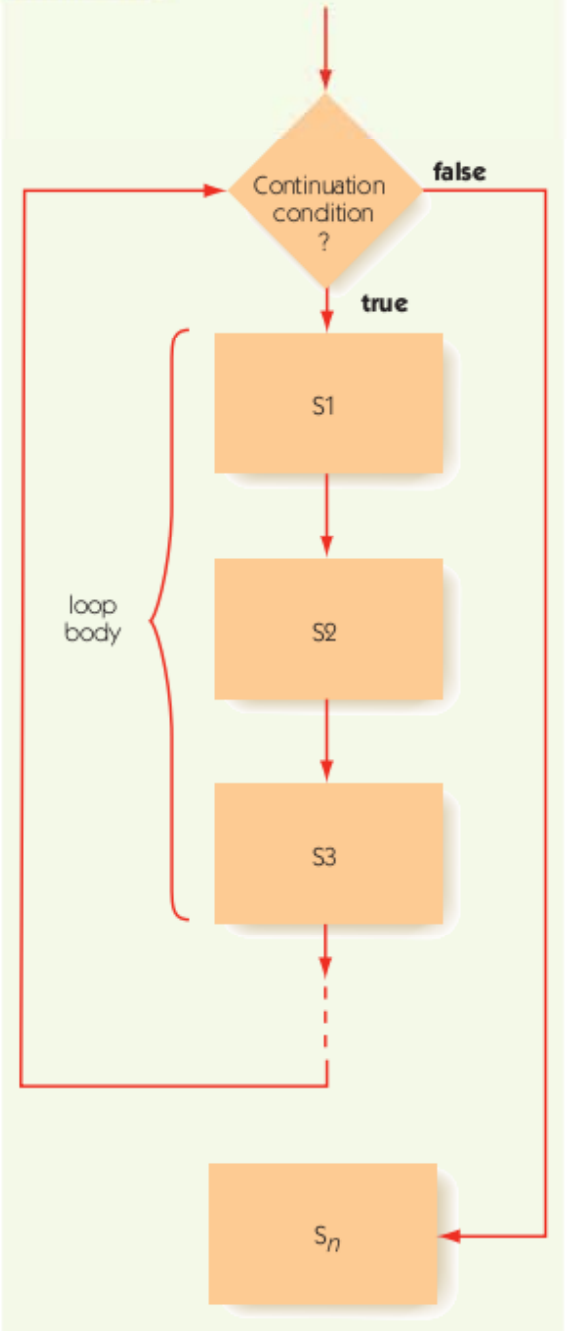
Figure 2.6 is a visual model of the execution of a while loop. The algorithm first evaluates the continuation condition inside the diamond-shaped symbol. If it is true, then it executes the sequence of operations labeled S1, S2, S3, . . . , which are the operations of the loop body. Then the algorithm returns to the top of the loop and reevaluates the condition. If the condition is false, then the loop has ended, and the algorithm continues executing with the statement after the loop, the one labeled S_n in Figure 2.6.

Here is a simple example of a loop:

Step	***Operation***
1	Set the value of *count* to 1
2	While (*count* ≤ 100) do Step 3 through Step 5
3	Set *square* to (*count* × *count*)
4	Print the values of *count* and *square*
5	Add 1 to *count*

Step 1 initializes *count* to 1, the next operation determines that (*count* ≤ 100), and then the loop body is executed, which in this case includes the three statements in Steps 3, 4, and 5. Those statements compute the value of *count* squared (Step 3) and print the value of both *count* and *square* (Step 4). The last operation inside the loop body (Step 5) adds 1 to *count* so that it now has the value 2. At the end of the loop, the algorithm must determine

FIGURE 2.6



Execution of the while loop

whether it should be executed again. Because *count* is 2, the continuation condition (*count* $\leq$ 100) is still true, and the algorithm must perform the loop body again. Looking at the entire loop, we can see that it will execute 100 times, producing the following output, which is a table of numbers and their squares from 1 to 100.

1	1
2	4
3	9
.	
.	
.	
100	10,000

At the end of the 100th pass through the loop, the value of *count* is incremented in Step 5 to 101. When the continuation condition is evaluated, it is false (because 101 is not less than or equal to 100), and the loop terminates.

We can see additional examples of loop structures in Steps 3 through 6 of Figure 1.2 and in Steps 3 through 6 of Figure 1.3. Another example is shown in Figure 2.7, which is yet another variation of the average miles per gallon algorithm of Figures 2.3 and 2.5. In this modification, after finishing one computation, the algorithm asks the user whether to repeat this calculation again. It waits until it gets a Yes or No response and repeats the entire loop body until the response provided by the user is No. (Note that the algorithm must initialize the value of *response* to Yes because the very first thing that the loop does is test the value of this quantity.)

There are many variations of this particular looping construct in addition to the while statement just described. For example, it is common to omit

FIGURE 2.7

Step	Operation
1	*response* = Yes
2	While (*response* = Yes) do Steps 3 through 11
3	Get values for *gallons used*, *starting mileage*, *ending mileage*
4	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
5	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
6	Print the value of *average miles per gallon*
7	If average miles per gallon > 25.0 then
8	Print the message 'You are getting good gas mileage'
	Else
9	Print the message 'You are NOT getting good gas mileage'
10	Print the message 'Do you want to do this again? Enter Yes or No'
11	Get a new value for *response* from the user
12	Stop

Third version of the average miles per gallon algorithm

the line numbers from algorithms and simply execute them in order, from top to bottom. In that case, we could use an "End of the loop" construct (or something similar) to mark the end of the loop rather than explicitly stating which steps are contained in the loop body. Using this approach, our loops would be written something like this:

```
While ("a true/false condition") do
        operation
        .
        .
        .
        operation
End of the loop
```

In this case, the loop body is delimited not by explicit step numbers but by the two lines that read, "While ..." and "End of the loop".

The type of loop just described is called a *pretest loop* because the continuation condition is tested at the *beginning* of each pass through the loop, and therefore it is possible for the loop body never to be executed. (This would happen if the continuation condition were *initially* false.) Sometimes this can be inconvenient, as we see in Figure 2.7. In that algorithm, the value of the variable called *response* is tested for the first time long before we ask the user if he or she wants to solve the problem again. Therefore, we had to give *response* a "dummy" value of Yes so that the test would be meaningful when the loop was initially entered.

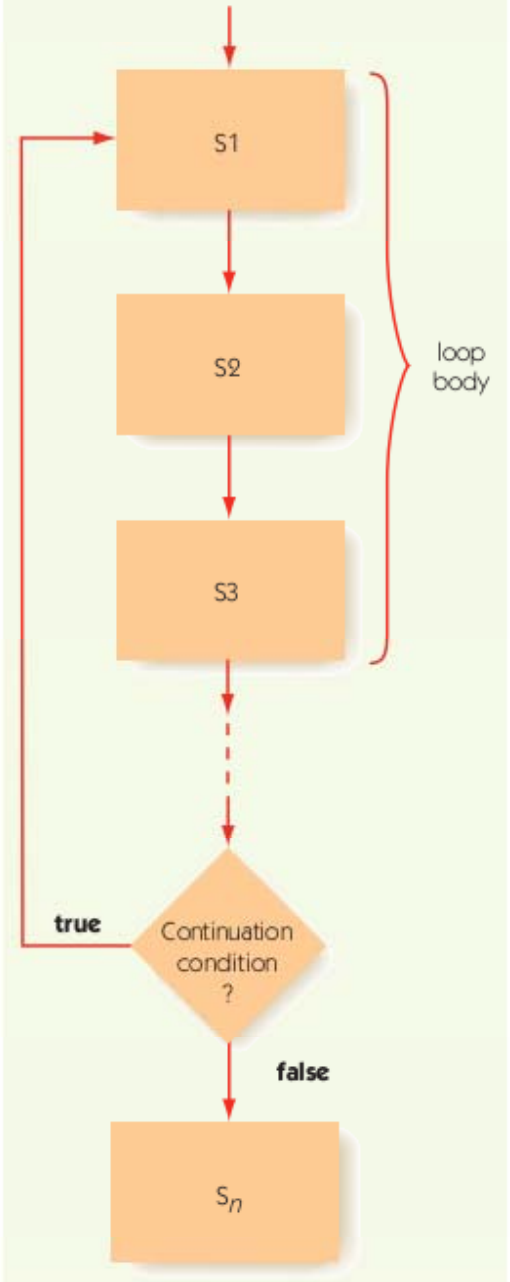
A useful variation of the looping structure is called a *posttest loop*, which also uses a true/false continuation condition to control execution of the loop. However, now the test is done at the *end* of the loop body, not the beginning. The loop is typically expressed using the *do/while* statement, which is usually written as follows:

```
Do
   operation
   operation
   .
   .
   .
   operation
While ("a true/false condition")
```

This type of iteration performs all the algorithmic operations contained in the loop body before it evaluates the true/false condition specified at the end of the loop. If this condition is false, the loop is terminated and

execution continues with the operation following the loop. If the condition is true, then the entire loop body is executed again. Note that in the do/while variation, the loop body is always executed at least once, whereas the while loop can execute 0, 1, or more times. Figure 2.8 diagrams the execution of the posttest do/while looping structure.

FIGURE 2.8



Execution of the do/while posttest loop

Figure 2.9 summarizes the algorithmic operations introduced in this section. These represent the primitive operations of our computing agent. These are the instructions that we assume our computing agent understands and is capable of executing without further explanation or simplification. In the next section, we will use these operations to design algorithms that solve some interesting and important problems.

FIGURE 2.9

Computation:

Set the value of "variable" to "arithmetic expression"

Input/Output:

Get a value for "variable", "variable"...
Print the value of "variable", "variable", ...
Print the message 'message'

Conditional:

If "a true/false condition" is true then
    first set of algorithmic operations
Else
    second set of algorithmic operations

Iterative:

While ("a true/false condition") do Step *i* through Step *j*
    Step *i*: operation
    .
    .
    .
    Step *j*: operation

While ("a true/false condition") do
    operation
    .
    .
    .
    operation
End of the loop

Do
    operation
    operation
    .
    .
    .
    operation
While ("a true/false condition")

Summary of pseudocode language instructions

From Little Primitives Mighty Algorithms Grow

Although the set of algorithmic primitives shown in Figure 2.9 might seem quite puny, it is anything but! In fact, an important theorem in theoretical computer science proves that the operations shown in Figure 2.9 are sufficient to represent *any* valid algorithm. No matter how complicated it might be, if a problem can be solved algorithmically, it can be expressed using only the sequential, conditional, and iterative operations just discussed. This includes not only the simple addition algorithm of Figure 1.2 but also the exceedingly complex algorithms needed to operate the International Space Station, manage billions of Facebook accounts, and implement all the Internal Revenue Service's tax rules and regulations.

In many ways, building algorithms is akin to constructing essays or novels using only the 26 letters of the English alphabet, plus a few punctuation symbols. Expressive power does not always come from having a huge set of primitives. It can also arise from a small number of simple building blocks combined in interesting and complex ways. This is the real secret of building algorithms.

2.3 Examples of Algorithmic Problem Solving

2.3.1 Example 1: Go Forth and Multiply

Our first example of algorithmic problem solving addresses a problem originally posed in Chapter 1 (Exercise 12). That problem asked you to implement an algorithm to multiply two numbers using repeated addition. This problem can be formally expressed as follows:

> Given two nonnegative integer values, $a \geq 0$, $b \geq 0$, compute and output the product $(a \times b)$ using the technique of repeated addition. That is, determine the value of the sum $a + a + a + \ldots + a$ (b times).

Obviously, we need to create a loop that executes exactly b times, with each execution of the loop adding the value of a to a running total. These operations will not make any sense (that is, they will not be effectively

Practice Problems

1. Write an if/then/else statement that sets the variable *y* to the value 1 if $x \geq 0$. If $x < 0$, then the statement should set *y* to the value 2. (Assume *x* already has a value.)
2. Write an algorithm that gets as input three data values *x*, *y*, and *z* and outputs the average of these values if the value of *x* is positive. If the value of *x* is either 0 or negative, your algorithm should not compute the average but should print the error message 'Bad Data' instead.
3. Write an algorithm that gets as input your current credit card balance, the total dollar amount of new purchases, and the total dollar amount of all payments. The algorithm computes the new balance, which this time includes an 8% interest charge on any unpaid balance below $100, 12% interest on any unpaid balance between $100 and $500, inclusive, and 16% on any unpaid balance above $500.
4. Write an algorithm that gets as input a single nonzero data value *x* and outputs the three values x^2, sin *x*, and 1/*x*. This process is repeated until the input value for *x* is equal to 999, at which time the algorithm terminates.
5. Write an algorithm that inputs the length and width, in feet, of a rectangular carpet and the price of the carpet in dollars per square yard. It then determines if you can afford to purchase this carpet, given that your total budget for carpeting is $500.
6. Add the following feature to the algorithm created in the previous Practice Problem: If the cost of the carpet is less than or equal to $250, output a message that this is a particularly good deal.
7. Add the following feature to the algorithm created in Practice Problem 4 above. Assume the input for the data value *x* can be any value, including 0. Since the value 1/*x* cannot be computed if $x = 0$, you will have to check for this condition and output an error message saying that you are unable to compute the value 1/*x*.

computable) until we have explicit values for *a* and *b*. So one of the first operations in our algorithm must be to input these two values.

Get values for *a* and *b*

To create a loop that executes exactly *b* times, we create a counter, let's call it *count*, initialized to 0 and incremented by (increased by) 1 after each pass through the loop. This means that when we have completed the loop once, the value of *count* is 1; when we have completed the loop twice, the value of *count* is 2; and so forth. Because we want to stop when we have completed the loop *b* times, we want to stop when (*count* = *b*). Therefore, the condition

for continuing execution of the loop is ($count < b$). Putting all these pieces together produces the following algorithmic structure, which is a loop that executes exactly b times as the variable *count* ranges from 0 up to ($b - 1$).

Get values for *a* and *b*

Set the value of *count* to 0

While (*count* < *b*) do

    ... the rest of the loop body will go here ...

    Set the value of *count* to (*count* + 1)

End of the loop

The purpose of the loop body is to add the value of a to a running total, which we will call *product*. We express that operation in the following manner:

Set the value of *product* to (*product* + *a*)

This statement says the new value of *product* is to be reset to the current value of *product* plus a.

What is the current value of *product* the first time this operation is encountered? Unless we initialize it, it has no value, and this operation is not effectively computable. Before starting the loop, we must be sure to include the following step:

Set the value of *product* to 0

Now our solution is starting to take shape. Here is what we have developed so far:

Get values for *a* and *b*

Set the value of *count* to 0

Set the value of *product* to 0

While (*count* < *b*) do

        Set the value of *product* to (*product* + *a*)

        Set the value of *count* to (*count* + 1)

End of the loop

There are only a few minor "tweaks" left to make this a correct solution to our problem.

When the while loop completes, we have computed the desired result, namely ($a \times b$), and it is stored in *product*. However, we have not displayed that result, and as it stands, this algorithm produces no output. Remember

from Chapter 1 that one of the fundamental characteristics of an algorithm is that it produces an observable result. In this case, the desired result is the final value of *product*, which we can display using our output primitive:

Print the value of *product*

The original statement of the problem said that the two inputs a and b must satisfy the following conditions: $a \geq 0$ and $b \geq 0$. The previous algorithm works for positive values of a and b, but what happens when either $a = 0$ or $b = 0$? Does it still function correctly?

If $b = 0$, there is no problem. If you look at the while loop, you see that it continues executing so long as ($count < b$). The variable *count* is initialized to 0. If the input variable b also has the value 0, then the test ($0 < 0$) is initially false, and the loop is *never* executed. The variable *product* keeps its initial value of 0, and that is the output that is displayed, which is the correct answer.

Now let's look at what happens when $a = 0$ and b is any nonzero value, say 5,386. Of course we know immediately that the correct answer to $0 \times 5{,}386$ is 0, but our algorithm does not. Instead, the loop will execute 5,386 times, the value of b, each time adding the value of a, which is 0, to *product*. Because adding 0 to anything has no effect, *product* remains at 0, and that is the output that will be displayed. In this case, we do get the right answer, and our algorithm does work correctly. However, it gets that correct answer only after doing 5,386 unnecessary and time-wasting repetitions of the loop.

In Chapter 1, we stated that it is not only algorithmic correctness we are after but efficiency and elegance as well. The algorithms designed and implemented by computer scientists are intended to solve important real-world problems, and they must accomplish that task in a correct and reasonably efficient manner. Otherwise they are not of much use to their intended audience.

In this case, we can eliminate all of those needless repetitions of the loop by using our if/then/else conditional primitive. Right at the start of the algorithm, we ask if either a or b is equal to 0. If the answer is yes, we can immediately set the final result to 0 without requiring any further computations:

If (either $a = 0$ or $b = 0$) then

    Set the value of *product* to 0

Else

    … solve the problem as described above …

We will have much more to say about the critically important concepts of algorithmic efficiency and elegance in Chapter 3.

This completes the development of our multiplication algorithm, and the finished solution is shown in Figure 2.10.

FIGURE 2.10

```
Get values for a and b
If (either a = 0 or b = 0) then
    Set the value of product to 0
Else
    Set the value of count to 0
    Set the value of product to 0
    While (count < b) do
        Set the value of product to (product + a)
        Set the value of count to (count + 1)
    End of loop
Print the value of product
Stop
```

Algorithm for multiplication of nonnegative values via repeated addition

Practice Problems

1. Manually work through the algorithm in Figure 2.10 using the input values $a = 2$, $b = 4$. After each completed pass through the loop, write down the current value of the four variables *a*, *b*, *count*, and *product*.
2. Trace the execution of the algorithm in Figure 2.10 using the "special" input values $a = 0$ and $b = 0$. Does the algorithm produce the result you expect?
3. Describe exactly what would be output by the algorithm in Figure 2.10 for each of the following two cases, and state whether that output is or is not correct. (*Note*: Because one of the two inputs is negative, these values violate the basic conditions of the problem.)

   case 1: $a = -2$, $b = 4$ case 2: $a = 2$, $b = -4$

4. If the algorithm of Figure 2.10 produced the wrong answer for either case 1 or case 2 of Practice Problem 3, explain exactly how you could fix the algorithm so it works correctly and produces the correct answer.
5. Explain why the multiplication algorithm shown in Figure 2.10 is or is not an efficient way to do multiplication. Justify and explain your answer.
6. Modify the algorithm in Figure 2.10 so it examines the two inputs a and b and if either one is greater than 10,000, it displays a message saying that for large numbers like this we should use a different, and more efficient, multiplication algorithm. It then terminates without computing a result.

This first example needed only two integer values, a and b, as input. That is a bit unrealistic, as most interesting computational problems deal not with a few numbers but with huge collections of data, such as long lists of names or large sets of experimental data. In the following sections, we will show examples of the types of processing—searching, reordering, comparing—often done on huge collections of information.

2.3.2 Example 2: Looking, Looking, Looking

Finding a solution to a given problem is called **algorithm discovery**, and it is the most challenging and creative part of the problem-solving process. We developed an algorithm for a fairly simple problem (multiplication by repeated addition) in Example 1. Discovering a correct and efficient algorithm to solve a complicated problem can be difficult and can involve equal parts of intelligence, hard work, past experience, technical skill, and plain good luck. In the remaining examples, we will develop solutions to a range of problems to give you more experience in working with algorithms. Studying these examples, together with lots of practice, is by far the best way to learn creative problem solving, just as experience and practice are the best ways to learn how to write essays, hit a golf ball, or repair cars.

The next problem we address involves finding a person's name given his or her telephone number, an application often referred to as *reverse telephone lookup*. This is the type of important but rather menial repetitive task so well suited to computerization. (Apple has a half-dozen reverse telephone lookup apps in its App Store.)

This algorithm could be used, for example, to implement *Caller ID*, in which the caller's name (if it is found) is shown on a screen so you can decide whether or not to answer the phone. It can also be used with missed calls to determine whether the caller is someone with whom you actually wish to speak. Since there are approximately 350 million listed phone numbers in the United States, reverse telephone lookup can only be implemented using computer-based search techniques.

Assume that we have a list of 10,000 telephone numbers (rather than 350 million) that we represent symbolically as T_1, T_2, T_3, … , $T_{10,000}$, along with the 10,000 names of the individuals associated with each specific phone number, denoted as N_1, N_2, N_3, … , $N_{10,000}$. That is, N_1 is the name of the person who "owns" the telephone with the number T_1, and so forth. On this first attempt to build an algorithm, let's assume that the 10,000 telephone numbers are not necessarily in numerical order. (In Chapter 3 we will modify the problem to search a list of phone numbers sorted into ascending order.)

Essentially what we have described is randomly ordered pairs of number/name lists with the following structure:

Telephone Number	*Name*	
T_1	N_1	10,000 (phone number, name) pairs
T_2	N_2	
T_3	N_3	
.	.	
.	.	
.	.	
$T_{10,000}$	$N_{10,000}$	

Our algorithm should allow us to input a specific telephone number, which we will symbolically denote as *NUMBER*. The algorithm will then search to see if *NUMBER* matches any of the 10,000 numbers contained in our reverse directory. If *NUMBER* matches T_j, where j is some value between 1 and 10,000, then the output of our algorithm will be the name of the person associated with that number: the value N_j. If *NUMBER* is not in our reverse directory, then the output of our algorithm will be the message 'I am sorry but this number is not in our directory.' This type of lookup algorithm has many additional uses. For example, it could be used to locate the zip code of a particular city, the seat number of a specific airline passenger, or the room number of a hotel guest.

Because the numbers in our directory are not in numerical order, there is no clever way to speed up the search. With a randomly ordered collection, there is no method more efficient than starting at the beginning and looking at each number in the list, one at a time, until we either find the one we are looking for or we come to the end of the list. This rather simple and straightforward technique is called *sequential search*, and it is the standard algorithm for searching an *unordered* list of values. For example, this is how we would search a bookshelf for a book with a particular title if the books were sorted by the author's name instead of by title. It is also the way that we would search a shuffled deck of cards trying to locate one particular card. A first attempt at designing a sequential search algorithm to solve our search problem might look something like Figure 2.11.

The solution shown in Figure 2.11 is extremely long. At 66 lines per page, it would require about 150 pages to write out the 10,002 steps in the completed solution. It would also be unnecessarily slow. If we are lucky enough to find *NUMBER* in the very first position of the list, T_1, then we get the answer N_1 almost immediately. However, the algorithm does not stop at that point. Even though it has already found the correct answer, it foolishly asks 9,999 more questions looking for *NUMBER* in positions T_2, …, T_{10000}. Of course, humans have enough common sense to know that when they find the answer they are searching for, they can stop. However, we cannot assume common sense in a computer system. On the contrary, a computer will mechanically execute the entire algorithm from the first step to the last.

FIGURE 2.11

Step	Operation
1	Get values for *NUMBER*, $T_1, \ldots, T_{10,000}$, and $N_1, \ldots, N_{10,000}$
2	If *NUMBER* = T_1 then print the value of N_1
3	If *NUMBER* = T_2 then print the value of N_2
4	If *NUMBER* = T_3 then print the value of N_3
.	.
.	.
.	.
10,000	If *NUMBER* = $T_{9,999}$ then print the value of $N_{9,999}$
10,001	If *NUMBER* = $T_{10,000}$ then print the value of $N_{10,000}$
10,002	Stop

First attempt at designing a sequential search algorithm

Not only is the algorithm excessively long and highly inefficient, it is also wrong. If the desired *NUMBER* is not in the list, this algorithm simply stops (at Step 10,002) rather than providing the desired result, a message that the number you requested could not be found. An algorithm is deemed correct only when it produces the correct result for *all* possible cases.

The problem with this first attempt is that it does not use the powerful algorithmic concept of *iteration*. Instead of writing an instruction 10,000 separate times, it is far better to write it only once and indicate that it is to be repetitively *executed* 10,000 times, or however many times it takes to obtain the answer. As you learned in the previous section, much of the power of a computer comes from being able to perform a *loop*—the repetitive execution of a block of statements a large number of times. Virtually every algorithm developed in this text contains at least one loop and most contain many. (This is the difference between the two shampooing algorithms shown in Figures 1.3 and 1.4. The algorithm in the former contains a loop; that in the latter does not.)

The algorithm in Figure 2.12 shows how we might write a loop to implement the sequential search technique. It uses a variable called *i* as an *index*, or *pointer*, into the list of all numbers. That is, T_i refers to the *i*th number in the list. The algorithm then repeatedly executes a group of statements using different values of *i*. The variable *i* can be thought of as a "moving finger" scanning the list of telephone numbers and pointing to the one on which the algorithm is currently working.

The first time through the loop, the value of the index *i* is 1, so the algorithm checks (in Step 4) to see whether *NUMBER* is equal to T_1, the first one on the list. If it is, then the algorithm writes out the result and sets the variable *Found* to YES, which causes the loop in Steps 4 through 7 to terminate. If T_1 is not the desired *NUMBER*, then *i* is incremented by 1 (in Step 7) so that it now has the value 2, and the loop is executed again. The algorithm

FIGURE 2.12

Step	Operation
1	Get values for *NUMBER*, $T_1, \ldots, T_{10,000}$, and $N_1, \ldots, N_{10,000}$
2	Set the value of *i* to 1 and set the value of *Found* to NO
3	While (*Found* = NO) do Steps 4 through 7
4	If *NUMBER* is equal to the *i*th number on the list, T_i, then
5	Print the name of the corresponding person, N_i
6	Set the value of *Found* to YES
	Else (*NUMBER* is not equal to T_i)
7	Add 1 to the value of *i*
8	Stop

Second attempt at designing a sequential search algorithm

now checks to see whether *NUMBER* is equal to T_2, the second number on the list. In this way, the algorithm uses the single conditional statement "If *NUMBER* is equal to the *i*th number on the list ..." to check up to 10,000 different values. It executes that one line over and over, each time with a different value of *i*. This is the advantage of using iteration.

However, the attempt shown in Figure 2.12 is not yet a complete and correct algorithm because it still does not work correctly when the desired *NUMBER* does not appear anywhere in our reverse directory. This final problem can be solved by terminating the loop either when the desired phone number is found or when we reach the end of the list. The algorithm can determine exactly what happened by checking the value of *Found* when the loop terminates. If the value of *Found* is NO, then the loop terminated because the index *i* exceeded 10,000, and we searched the entire list without finding the desired *NUMBER*. The algorithm should then produce an appropriate message.

An iterative solution to the sequential search algorithm that incorporates this feature is shown in Figure 2.13. The sequential search algorithm shown in Figure 2.13 is a correct solution to our reverse telephone lookup problem. It meets all the requirements listed in Section 1.3.1: It is well ordered, each of the operations is clearly defined and effectively computable, and it is certain to halt with the desired result after a finite number of operations. (In Exercise 12 at the end of this chapter, you will develop a formal argument that proves that this algorithm will always halt.) Furthermore, this algorithm requires only 10 steps to write out fully, rather than the 10,002 steps of the first attempt in Figure 2.11. As you can see, not all algorithms are created equal.

Looking at the algorithm in Figure 2.13, our first thought might be that this is not at all how people would manually search a list of telephone numbers looking for one specific value. Humans would never turn to page 1, column 1, and start scanning all numbers beginning with (000) 000-0001

FIGURE 2.13

Step	Operation
1	Get values for *NUMBER*, $T_1, \ldots, T_{10,000}$, and $N_1, \ldots, N_{10,000}$
2	Set the value of *i* to 1 and set the value of *Found* to NO
3	While both (*Found* = NO) and ($i \leq 10{,}000$) do Steps 4 through 7
4	If NUMBER is equal to the ith number on the list, T_i then
5	Print the name of the corresponding person, N_i
6	Set the value of *Found* to YES
	Else (*NUMBER* is not equal to T_i)
7	Add 1 to the value of *i*
8	If (*Found* = NO) then
9	Print the message 'Sorry, this number is not in the directory'
10	Stop

The sequential search algorithm

(which is not actually a valid telephone number according to the North American Numbering Plan that covers the United States, Canada, and many Caribbean islands). In all likelihood, a communications company located in New York City would not be satisfied with the performance of the sequential search algorithm of Figure 2.13 when applied to the 20 million or so phones in its city.

But because our reverse directory was not sorted into numerical order, we really had no choice in the design of our search algorithm. However, in real life we can do much better than sequential search, because these types of directories *are* sorted numerically, and we can exploit this fact during the search process. For example, we know that the digit 5 is about halfway through the set of decimal digits 0–9. So when looking for the owner of phone number (555) 123-4567 in a sorted reverse directory, we could start our search somewhere in the middle rather than on the first page. We then see exactly where we are by looking at the first digit of the phone numbers on the current page and then move forward or backward toward numbers beginning with 5. This approach allows us to find the desired telephone number much more quickly than searching the numbers sequentially from the beginning of the list.

This use of different search techniques points out a very important concept in the design of algorithms:

The selection of an algorithm to solve a problem is greatly influenced by the way the input data for that problem is organized.

An algorithm is a method for processing some data to produce a result, and the way the data is organized has an enormous influence both on the algorithm we select and on how speedily that algorithm can produce the desired result.

Laboratory Experience 2

Computer science is an empirical discipline as well as a theoretical one. Learning comes not just from reading about concepts like algorithms but also from manipulating and observing them. The laboratory manual for this text includes laboratory exercises that enable you to engage the ideas and concepts presented on these pages. Laboratory Experience 2 introduces the concept of *algorithm animation*, in which you can observe an algorithm being executed and watch as data values are dynamically transformed into final results.

Bringing an algorithm to life in this way can help you understand what the algorithm does and how it works. The first animation that you will work with is the sequential search algorithm shown in Figure 2.13. The laboratory software allows you to create a list of data values and to watch as the algorithm searches this list to determine whether a special target value occurs.

We strongly encourage you to work through these Laboratory Experiences to deepen your understanding of the ideas presented in this and the following chapters.

In Chapter 3, we will expand on the concept of the efficiency and quality of algorithms, and we will present an algorithm for searching *sorted* reverse directories that is far superior to the one shown in Figure 2.13.

2.3.3 Example 3: Big, Bigger, Biggest

The third algorithm we will develop is similar to the sequential search in Figure 2.13 in that it also searches a list of values. However, this time the algorithm will search not for a particular value supplied by the user but for the numerically largest value in a list of numbers. This type of "find largest" algorithm could be used to answer a number of important questions. (With only a single trivial change, the same algorithm also finds the smallest value, so a better name for it might be "find extreme values.") For example, given a list of examinations, which student received the highest (or lowest) score? Given a list of annual salaries, which employee earns the most (or least) money? Given a list of grocery prices from different stores, where should I shop to find the lowest price? All these questions could be answered by executing this type of algorithm.

In addition to being important in its own right, such an algorithm can also be used as a "building block" for the construction of solutions to other problems. For example, the Find Largest algorithm that we will develop

could be used to implement a *sorting algorithm* that puts an unordered list of numbers into ascending order. (Find and remove the largest item in list A and move it to the last position of list B. Now repeat these operations, each time moving the largest remaining number in list A to the last unfilled slot of list B. We will develop and write this algorithm in Chapter 3.)

The use of a "building-block" component is a very important concept in computer science. The examples in this chapter might lead you to believe that every algorithm you write must be built from only the most elementary and basic of primitives—the sequential, conditional, and iterative operations shown in Figure 2.9. However, once an algorithm has been developed, it may itself be used in the construction of other, more complex algorithms, just as we will use Find Largest in the design of a sorting algorithm. This is similar to what a builder does when constructing a home from prefabricated units rather than bricks and boards. Our problem-solving task need not always begin at the beginning but can instead build on ideas and results that have come before. Every algorithm that we create becomes, in a sense, a primitive operation of our computing agent and can be used as part of the solution to other problems. That is why a collection of useful, prewritten algorithms, called a library, is such an important tool in the design and development of algorithms.

Formally, the problem we will be solving in this section is defined as follows:

> Given a value $n \geq 1$ and a list containing exactly n unique numbers called A_1, A_2, ... , A_n, find and print both the largest value in the list and the position in the list where that largest value occurred.

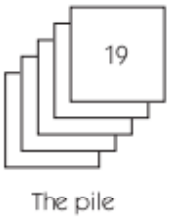
For example, if our list contained the five values

19, 41, 12, 63, 22 ($n = 5$)

then our algorithm should locate the largest value, 63, and print that value together with the fact that it occurred in the fourth position of the list. (*Note*: Our definition of the problem states that all numbers in the list are unique, so there can be only a single occurrence of the largest number. Exercise 15 at the end of the chapter asks how our algorithm would behave if the numbers in the list were not unique and the largest number could occur two or more times.)

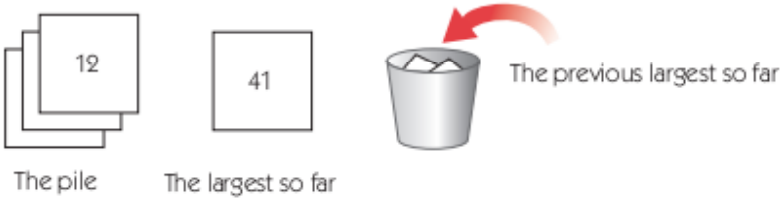
When faced with a problem statement like the one just given, how do we go about creating a solution? What strategies can we employ to discover a correct and efficient answer to the problem? One way to begin is to ask ourselves how the same problem might be solved by hand. If we can understand and explain how we would approach the problem manually, we might be able to express that manual solution as a formal algorithm.

For example, suppose we were given a pile of papers, each of which contains a single number, and were asked to locate the largest number in the pile. (The following diagrams assume the papers contain the five values 19, 41, 12, 63, and 22.)

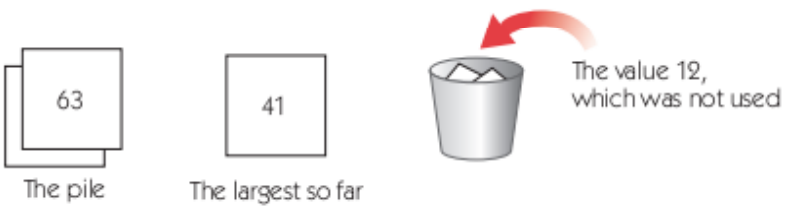


We might start off by saying that the first number in the pile (the top one) is the largest one that we have seen so far, and then putting it to the side where we are keeping the largest value.

Now we compare the top number in the pile with the one that we have called the largest one so far. In this case, the top number in the pile, 41, is larger than our current largest, 19, so we make it the new largest. To do this, we throw the value 19 into the wastebasket (or, better, into the recycle bin) and put the number 41 to the side because it is the largest value encountered so far.



We now repeat this comparison operation, asking whether the number on top of the pile is larger than the largest value seen so far, now 41. This time the value on top of the pile, 12, is not larger, so we do not want to save it. We simply throw it away and move on to the next number in the pile.



This compare-and-save-or-discard process continues until our original pile of numbers is empty, at which time the largest so far is the largest value in the entire list.

Let's see how we can convert this informal, pictorial solution into a formal algorithm that is built from the primitive operations shown in Figure 2.9.

We certainly cannot begin to search a list for a largest value until we have a list to search. Therefore, our first operation must be to get a value for n, the size of the list, followed by values for the n-element list $A_1, A_2, \ldots, A_n$. This can be done using our input primitive:

Get a value for *n*, the size of the list

Get values for $A_1, A_2, \ldots, A_n$, the list to be searched

Now that we have the data, we can begin to implement a solution.

Our informal description of the algorithm stated that we should begin by calling the first item in the list, A_1, the largest value so far. (We know that this operation is meaningful because we stated that the list must always have at least one element.) We can express this formally as

Set the value of *largest so far* to A_1

Our solution must also determine where that largest value occurs. To remember this value, let's create a variable called *location* to keep track of the position in the list where the largest value occurs. Because we have initialized *largest so far* to the first element in the list, we should initialize *location* to 1.

Set the value of *location* to 1

We are now ready to begin looking through the remaining items in list A to find the largest one. However, if we write something like the following instruction:

If the second item in the list is greater than *largest so far* then ...

we will have made exactly the same mistake that occurred in the initial version of the sequential search algorithm shown in Figure 2.11. This instruction explicitly checks only the second item of the list. We would need to rewrite that statement to check the third item, the fourth item, and so on. Again, we are failing to use the idea of *iteration*, where we repetitively execute a loop as many times as it takes to produce the desired result.

To solve this problem, let's use the same technique used in the sequential search algorithm. Let's not talk about the second, third, fourth, ... item in the list but about the *i*th item in the list, where *i* is a variable that takes on different values during the execution of the algorithm. Using this idea, a statement such as

If $A_i >$ *largest so far* then ...

can be executed with different values for *i*. This allows us to check all *n* values in the list with a single statement. Initially, *i* should be given the value 2 because the first item in the list was automatically set to the largest value. Therefore, we want to begin our search with the second item in the list.

.
.
.

Set the value of *i* to 2

.
.
.

If A_i > *largest so far* then ...

What operations should appear after the word *then*? A check of our earlier discussion shows that the algorithm must reset the values of both *largest so far* and *location*.

If A_i > *largest so far* then
    Set *largest so far* to A_i
    Set *location* to *i*

If A_i is not larger than *largest so far*, then we do not want the algorithm to do anything. To indicate this, the if/then instruction can include an else clause that looks something like

Else
    Don't do anything at all to *largest so far* and *location*

This is certainly correct, but instructions that tell us not to do anything are usually omitted from an algorithm because they do not carry any meaningful information.

Regardless of whether the algorithm resets the values of *largest so far* and *location*, it needs to move on to the next item in the list. Our algorithm refers to A_i, the *i*th item in the list, so it can move to the next item by simply adding 1 to the value of *i* and repeating the if/then statement. The outline of this iteration can be sketched as follows:

If A_i > *largest so far* then
    Set *largest so far* to A_i
    Set *location* to *i*
Add 1 to the value of *i*
.
.
.

However, we do not want the loop to repeat forever. (Remember that one of the properties of an algorithm is that it must eventually halt.) What stops

this iterative process? When does the algorithm display an answer and terminate execution?

The conditional operation "If $A_i >$ *largest so far* then ..." is meaningful only if A_i represents an actual element of list A. Because A contains n elements numbered 1 to n, the value of i must be in the range 1 to n. If $i > n$, then the loop has searched the entire list, and it is finished. Therefore, our continuation condition should be expressed as ($i \leq n$). When this condition becomes false, the algorithm can stop looping and print the values of both *largest so far* and *location*. Using our looping primitive, we can describe this iteration as follows:

```
While (i ≤ n) do
    If A_i > largest so far then
        Set largest so far to A_i
        Set location to i
    Add 1 to the value of i
End of the loop
```

We have now developed all the pieces of the algorithm and can finally put them together. Figure 2.14 shows the completed Find Largest algorithm. Note that the steps are not numbered. This omission is quite common, especially as algorithms get larger and more complex.

FIGURE 2.14

```
Get a value for n, the size of the list
Get values for A_1, A_2, ... , A_n, the list to be searched
Set the value of largest so far to A_1
Set the value of location to 1
Set the value of i to 2
While (i ≤ n) do
    If A_i > largest so far then
        Set largest so far to A_i
        Set location to i
    Add 1 to the value of i
End of the loop
Print out the values of largest so far and location
Stop
```

Algorithm to find the largest value in a list

Practice Problems

1. What part(s) of the sequential search algorithm of Figure 2.13 would need to be changed if our phone book contained 1 million numbers rather than 10,000?
2. Rewrite the sequential search algorithm to use the do/while looping structure shown in Figure 2.9 in place of the while structure.
3. Modify the algorithm of Figure 2.14 so that it finds the smallest value in a list rather than the largest. Describe exactly what changes were necessary.
4. Describe exactly what would happen to the algorithm in Figure 2.14 if you tried to apply it to an empty list of length $n = 0$. Describe exactly how you could fix this problem.
5. Describe exactly what happens to the algorithm in Figure 2.14 when it is presented with a list with exactly one item, i.e., $n = 1$. Determine whether this algorithm will or will not work correctly on this one-item list.
6. Would the Find Largest algorithm of Figure 2.14 still work correctly if the test on Line 7 were written as ($A_i \geq$ *largest so far*)? Explain why or why not.

Laboratory Experience 3

Like Laboratory Experience 2, this laboratory also uses animation to help you better understand the concept of algorithm design and execution. It presents an animation of the Find Largest algorithm discussed in the text and shown in Figure 2.14.

This Laboratory Experience allows you to create a list of data and watch as the algorithm attempts to determine the largest numerical value contained in that list. You will be able to observe dynamic changes to the variables *index*, *location*, and *maximum*, and will be able to see how values are set and discarded as the algorithm executes. Like the previous Laboratory Experience, it is intended to give you a deeper understanding of how this algorithm works by allowing you to observe its behavior.

2.3.4 Example 4: Meeting Your Match

The last algorithm we develop in this chapter solves a common problem in computer science called *pattern matching*. For example, imagine that you have a large collection of Civil War data files that you want to use as resource material for an article on Abraham Lincoln. Your first step would probably be to search these files to locate every occurrence of the text patterns "Abraham Lincoln," "A. Lincoln," and "Lincoln." The process of searching for a special pattern of symbols within a larger collection of information is called *pattern matching*. Most good word processors provide this service as a menu item called *Find* or something similar. Furthermore, most web search engines try to match your search terms to the keywords that appear on a webpage.

Pattern matching can be applied to almost any kind of information, including graphics, sound, and photographs. For example, an important medical application of pattern matching is to input an X-ray or CT scan image into a computer and then have the computer search for special patterns, such as dark spots, which represent conditions that should be brought to the attention of a physician. This can help speed up the interpretation of X-rays and avoid the problem of human error caused by fatigue or oversight. (Computers do not get tired or bored!)

One of the most interesting and exciting applications of pattern matching is to assist microbiologists and geneticists studying and analyzing the *human genome*, the basis for all human life. The Human Genome Project was started in 1990 with the goal of determining the sequence of chemical base pairs that comprise the human genome. The mammoth project was completed in April 2003, and today this genetic information is available via online databases to biological and medical researchers worldwide.

The human genome is composed of approximately 3.5 billion *nucleotides*, each of which can be one of only four different chemical compounds. These compounds (adenine, cytosine, thymine, and guanine) are usually referred to by the first letter of their chemical names: A, C, T, and G. Thus, the basis for our existence can be conceptually viewed as a very large "text file" written in a four-letter alphabet.

... T C G G A C T A A C A T C G G G A T C G A G A T G ...

(If you were to write out the entire sequence of nucleotides, at 12 characters per inch, it would stretch from New York City to Moscow.) Sequences of these nucleotides are called *genes*. There are about 25,000 genes in the human genome, and they determine virtually all of our physical characteristics—sex, race, eye color, hair color, and height, to name just a few. Genes are also an important factor in the occurrence of certain diseases. A missing or flawed nucleotide can result in one of a number of serious genetic disorders, such as Down syndrome or Tay-Sachs disease. To help find a cure

for these diseases, researchers attempt to locate individual genes that, when exhibiting a certain defect, cause a specific malady. The process of locating the starting and ending boundaries of an individual gene within the 3+ billion nucleotides is called *genome annotation*, and because of the scale of the problem it is done using automated algorithms that search DNA sequences in genomic databases—not unlike the algorithm we will develop in this section.

A gene is typically composed of thousands of nucleotides, and researchers generally do not know the entire sequence. However, they may know what a small portion of the gene—say, a few hundred nucleotides—looks like. Therefore, to search for one particular gene, they must match the sequence of nucleotides that they do know, called a *probe*, against the entire 3.5 billion-element genome to locate every occurrence of that probe. From this matching information, researchers hope to locate and isolate specific genes. For example,

When a match is found, researchers examine the nucleotides located before and after the probe to see whether they have located the desired gene and, if so, whether the gene is defective. Physicians hope someday to be able to "clip out" a bad sequence and insert in its place a correct sequence.

This application of pattern matching dispels any notion that the algorithms discussed here—sequential search (Figure 2.13), Find Largest (Figure 2.14), and pattern matching—are nothing more than academic exercises that serve as examples for introductory classes but have absolutely no role in solving real-world problems. The algorithms that we have presented (or will present) *are* important, either in their own right or as building blocks for algorithms used by physical scientists, mathematicians, engineers, and social scientists.

Let's formally define the pattern-matching problem as follows:

> You will be given some *text* composed of n characters that will be referred to as $T_1\ T_2\ \ldots\ T_n$. (*Note*: n may be very, very large.) You will also be given a *pattern* of m characters, $m \leq n$, that will be represented as $P_1\ P_2\ \ldots\ P_m$. (*Note*: m will usually be much, much smaller than n.) The algorithm must locate every occurrence of the given pattern within the text. The output of the algorithm is the location in the text where each match occurred. For this problem, the location of a match is defined to be the index position in the text where the match begins.

For example, if our text is the phrase "to be or not to be, that is the question" and the pattern for which we are searching is the word *to*, then our algorithm produces the following output:

Text: **to** be or not to be, that is the question
Pattern: **to**
Output: Match starting at position 1.
Text: to be or not **to** be, that is the question
Pattern: **to**
Output: Match starting at position 14. (The t is in position 14, including blanks.)

The pattern-matching algorithm that we will implement is composed of two parts. In the first part, the pattern is aligned under a specific position of the text, and the algorithm determines whether there is a match at that given position. The second part of the algorithm "slides" the entire pattern ahead one character position. Assuming that we have not gone beyond the end of the text, the algorithm returns to the first part to check for a match at this new position. Pictorially, this algorithm can be represented as follows:

Repeat the following two steps.

Step 1: The matching process: $T_1\ T_2\ T_3\ T_4\ T_5 \ldots$
$P_1\ P_2\ P_3$

Step 2: The slide forward: $T_1\ T_2\ T_3\ T_4\ T_5 \ldots$
one-character slide → $P_1\ P_2\ P_3$

The algorithm involves repetition of these two steps beginning at position 1 of the text and continuing until the pattern has slid off the right-hand end of the text.

A first draft of an algorithm that implements these ideas is shown in Figure 2.15, in which not all of the operations are expressed in terms of the basic algorithmic primitives of Figure 2.9. Although statements like "Set *k*, the starting location for the attempted match, to 1" and "Print the value of *k*, the starting location of the match" are just fine, the instructions "Attempt to match every character in the pattern beginning at position *k* of the text" and "Keep going until we have fallen off the end of the text" are certainly not primitives. On the contrary, they are high-level operations that, if written out using only the operations in Figure 2.9, would expand into many instructions.

FIGURE 2.15

```
Get values for n and m, the size of the text and the pattern, respectively
Get values for both the text T1 T2 ... Tn and the pattern P1 P2 ... Pm
Set k, the starting location for the attempted match, to 1
Keep going until we have fallen off the end of the text
    Attempt to match every character in the pattern beginning at
        position k of the text (this is Step 1 from the previous page)
    If there was a match then
        Print the value of k, the starting location of the match
    Add 1 to k, which slides the pattern forward one position (this is Step 2)
End of the loop
Stop
```

First draft of the pattern-matching algorithm

Is it okay to use high-level statements like this in our algorithm? Wouldn't their use violate the requirement stated in Chapter 1 that algorithms be constructed only from unambiguous operations that can be directly executed by our computing agent?

In fact, it is perfectly acceptable, and quite useful, to use high-level statements like this during the *initial phase* of the algorithm design process. When starting to design an algorithm, we might not want to think only in terms of elementary operations such as input, computation, output, conditional, and iteration. Instead, we might want to express our proposed solution in terms of high-level and broadly defined operations that represent dozens or even hundreds of primitive instructions. Here are some examples of these higher-level constructs:

- Sort the entire list into ascending order.
- Attempt to match the entire pattern against the text.
- Find a root of the equation.

Using instructions like these in an algorithm allows us to postpone worrying about how to implement that operation and lets us focus instead on other aspects of the problem. This is equivalent to inserting the phrase "put a brief historical introduction right here" into a story that you are writing. It functions as a reminder of something that you must add but that can be postponed until a later time. With our algorithm, we will come back to these high-level constructs and either express them in terms of our available primitives or use existing building-block algorithms taken from a program library. However, we can do this at our convenience.

The use of high-level instructions during the design process is an example of one of the most important intellectual tools in computer science—**abstraction**. Abstraction refers to the separation of the high-level view of

an entity or an operation from the low-level details of its implementation. It is abstraction that allows us to understand and intellectually manage any large, complex system, whether it is a mammoth corporation, a complex piece of machinery, or an intricate and very detailed algorithm. For example, the president of General Motors views the company in terms of its major corporate divisions and high-level policy issues, not in terms of every worker, every supplier, every car, or every bolt. Attempting to manage the company at that level of detail would drown the president in a sea of detail.

In computer science, we frequently use abstraction because of the complexity of hardware and software. For example, abstraction allows us to view the hardware component called "memory" as a single, indivisible high-level entity without paying heed to the billions of electronic devices that go into constructing a memory unit. (Chapter 4 examines how computer memories are built, and it makes extensive use of abstraction.) In algorithm design and software development, we use abstraction whenever we think of an operation at a high level and temporarily ignore how we might actually implement that operation. This allows us to decide which details to address now and which to postpone until later. Viewing an operation at a high level of abstraction and fleshing out the details of its implementation at a later time constitute an important computer science problem-solving strategy called **top-down design**.

Ultimately, however, we do have to describe how each of these high-level abstractions can be represented using the available algorithmic primitives. The fifth line of the first draft of the pattern-matching algorithm shown in Figure 2.15 reads:

> Attempt to match every character in the pattern beginning at position *k* of the text

When this statement is reached, the pattern is aligned under the text beginning with the *k*th character. Pictorially, we are in the following situation:

Text: $T_1\ T_2\ T_3 \ldots \quad T_k\ T_{k+1}\ T_{k+2} \ldots T_{k+(m-1)} \ldots$

Pattern: $\quad\quad\quad\quad\quad P_1 P_2\ \ P_3 \ldots\ \ P_m$

The algorithm must now perform the following comparisons:

Compare P_1 to T_k

Compare P_2 to T_{k+1}

Compare P_3 to T_{k+2}

.

.

.

Compare P_m to $T_{k+(m-1)}$

If every single one of these pairs is equal, then there is a match starting at position *k*. However, if even one pair is not equal, then there is no match, and the

algorithm can immediately cease making comparisons at this location. Thus, we must construct a loop that executes until one of two things happens—it has either completed m successful comparisons (i.e., we have matched the entire pattern) or it has detected a mismatch. When either of these conditions occurs, the loop stops; however, if neither condition has occurred, the loop must keep going. Algorithmically, this iteration can be expressed in the following way. (Remember that k is the starting location in the text.)

```
Set the value of i to 1
Set the value of Mismatch to NO
While both (i ≤ m) and (Mismatch = NO)
    If P_i ≠ T_{k+(i−1)} then
        Set Mismatch to YES
    Else
        Increment i by 1 (to move to the next character)
End of the loop
```

When the loop has finished, we can determine whether there was a match by examining the current value of the variable *Mismatch*. If *Mismatch* is YES, then there was not a match because at least one of the characters was out of place. If *Mismatch* is NO, then every character in the pattern matched its corresponding character in the text, and there is a match starting at position k.

```
If Mismatch = NO then
    Print the message 'There is a match at position '
    Print the value of k
```

Regardless of whether there was a match at position k, we now must add 1 to k to begin searching for a match at the next position. This is the "sliding forward" step diagrammed earlier.

The final high-level statement in Figure 2.15 that needs to be expanded is the loop on Line 4.

```
Keep going until we have fallen off the end of the text
```

What does it mean to "fall off the end of the text"? Where is the last possible place that a match can occur? To answer these questions, let's draw a diagram in which the last character of the pattern, P_m, lines up directly under T_n, the last character of the text.

Text:	T_1 T_2 T_3 …	T_{n-m+1} …	T_{n-2}	T_{n-1}	T_n
Pattern:		P_1 …	P_{m-2}	P_{m-1}	P_m

This diagram illustrates that the last possible place a match could occur is when the first character of the pattern is aligned under the character at

position T_{n-m+1} of the text because P_m is aligned under T_n, P_{m-1} is under T_{n-1}, P_{m-2} is aligned under T_{n-2}, and so on. Thus, P_1, which can be written as $P_{m-(m-1)}$, is aligned under $T_{n-(m-1)}$, which is T_{n-m+1}. If we tried to slide the pattern forward any further, we would truly "fall off" the right-hand end of the text. Therefore, our loop must terminate when k, the starting point for the match, strictly exceeds the value of $n - m + 1$. We can express this as follows:

While $(k \leq (n - m + 1))$ do

Now we have all the pieces of our algorithm in place. We have expressed every statement in Figure 2.15 in terms of our basic algorithmic primitives and are ready to put it all together. The final draft of the pattern-matching algorithm is shown in Figure 2.16.

2.4 Conclusion

You have now had a chance to see the step-by-step design and development of some interesting, nontrivial algorithms. You have also been introduced to a number of fundamental concepts related to problem solving, including algorithm design and discovery, pseudocode, control statements, iteration, libraries, abstraction, and top-down design. However, this by no means marks the end of our discussion of algorithms. The development of a

FIGURE 2.16

```
Get values for n and m, the size of the text and the pattern, respectively
Get values for both the text T1 T2 ... Tn and the pattern P1 P2 ... Pm
Set k, the starting location for the attempted match, to 1
While (k ≤ (n − m + 1)) do
    Set the value of i to 1
    Set the value of Mismatch to NO
    While both (i ≤ m) and (Mismatch = NO) do
        If Pi ≠ T(k+(i−1)) then
            Set Mismatch to YES
        Else
            Increment i by 1 (to move to the next character)
    End of the loop
    If Mismatch = NO then
        Print the message 'There is a match at position'
        Print the value of k
    Increment k by 1
End of the loop
Stop
```

Final draft of the pattern-matching algorithm

correct solution to a problem is only the first step in creating a useful piece of software.

Designing a technically correct algorithm to solve a given problem is only part of what computer scientists do. They must also ensure that they have created an *efficient* algorithm that generates results quickly enough for its intended users. Chapter 1 described a brute force chess algorithm that would, at least theoretically, play perfect chess but that would be unusable

Practice Problems

1. Consider the following "reverse telephone directory."

Number	Name
(648) 555-1285	Smith
(247) 834-6543	Jones
(771) 921-5281	Adams
(356) 327-8900	Doe

   Trace the sequential search algorithm of Figure 2.13 using each of the following *NUMBER*s and show the output produced.

   a. (771) 921-5281

   b. (488) 351-1673

2. Consider the following list of seven data values.

   22, 18, 23, 17, 25, 30, 2

   Trace the Find Largest algorithm of Figure 2.14 and show the output produced.

3. Consider the following text.

   Text: A man and a woman

   Trace the pattern-matching algorithm of Figure 2.16 using the two-character pattern *an* and show the output produced.

4. Explain exactly what would happen to the algorithm of Figure 2.16 if *m*, the length of the pattern, were greater than *n*, the length of the text.

5. Explain exactly what would happen to the algorithm of Figure 2.16 if Line 5 had been incorrectly written as "set the value of *i* to 0".

6. Determine whether the pattern-matching algorithm of Figure 2.16 will work if *n*, the length of the text, is exactly the same size as *m*, the length of the pattern.

because it would take millions of centuries to make its first move. Similarly, a reverse directory lookup program that takes 30 minutes to locate a person's name would be of little or no use. The caller would surely have hung up long before we learned who it was. This practical concern for efficiency and usefulness, in addition to correctness, is one of the hallmarks of computer science.

Therefore, after developing a correct algorithm, we must analyze it thoroughly and study its efficiency properties and operating characteristics. We must ask ourselves how quickly it will give us the desired results and whether it is better than other algorithms that solve the same problem. This analysis, which is the central topic of Chapter 3, enables us to create algorithms that are not only correct but elegant, efficient, and useful as well.

EXERCISES

1. Write pseudocode instructions to carry out each of the following computational operations:
   a. Determine the area of a triangle given values for the base *b* and the height *h*.
   b. Compute the interest earned in 1 year given the starting account balance *B* and the annual interest rate *I* and assuming simple interest, that is, no compounding. Also determine the final balance at the end of the year.
   c. Determine the flying time between two cities given the mileage *M* between them and the average speed of the airplane.
2. Using only the sequential operations described in Section 2.2.2, write an algorithm that gets values for the starting account balance *B*, annual interest rate *I*, and annual service charge *S*. Your algorithm should output the amount of interest earned during the year and the final account balance at the end of the year. Assume that interest is compounded monthly and the service charge is deducted once, at the end of the year.
3. Using only the sequential operations described in Section 2.2.2, write an algorithm that gets four numbers corresponding to scores received on three semester tests and a final examination. Your algorithm should compute and display the average of all four tests, weighting the final exam twice as heavily as a regular test.
4. Write an algorithm that gets the price for item A plus the quantity purchased. The algorithm prints the total cost, including a 6% sales tax.
5. Write an if/then/else primitive to do each of the following operations:
   a. Compute and display the value $x \div y$ if the value of *y* is not 0. If *y* does have the value 0, then display the message 'Unable to perform the division'.
   b. Compute the area and circumference of a circle given the radius *r* if the radius is greater than or equal to 1.0; otherwise, you should compute only the circumference.

6. Modify the algorithm of Exercise 2 to include the annual service charge only if the starting account balance at the beginning of the year is less than $1,000. If it is greater than or equal to $1,000, then there is no annual service charge.

7. Write an algorithm that uses a loop (1) to input 10 pairs of numbers, where each pair represents the score of a football game with the Computer State University (CSU) score listed first, and (2) for each pair of numbers, to determine whether CSU won or lost. After reading in these 10 pairs of values, print the won/lost/tie record of CSU. In addition, if this record is a perfect 10-0, then print the message 'Congratulations on your undefeated season'.

8. Modify the test-averaging algorithm of Exercise 3 so that it reads in 15 test scores rather than 4. There are 14 regular tests and a final examination, which counts twice as much as a regular test. Use a loop to input and sum the scores.

9. Modify the sales computation algorithm of Exercise 4 so that after finishing the computation for one item, it starts on the computation for the next. This iterative process is repeated until the total cost exceeds $1,000.

10. Write an algorithm that is given your electric meter readings (in kilowatt-hours) at the beginning and end of each month of the year. The algorithm determines your annual cost of electricity on the basis of a charge of 6 cents per kilowatt-hour for the first 1,000 kilowatt-hours of each month and 8 cents per kilowatt-hour beyond 1,000. After printing out your total annual charge, the algorithm also determines whether you used less than 500 kilowatt-hours for the entire year and, if so, prints out a message thanking you for conserving electricity.

11. Develop an algorithm to compute gross pay. The inputs to your algorithm are the hours worked per week and the hourly pay rate. The rule for determining gross pay is to pay the regular pay rate for all hours worked up to 40, time-and-a-half for all hours over 40 up to 54, and double time for all hours over 54. Compute and display the value for gross pay using this rule. After displaying one value, ask the user whether he or she wants to do another computation. Repeat the entire set of operations until the user says no.

12. Develop a formal argument that "proves" that the sequential search algorithm shown in Figure 2.13 cannot have an infinite loop; that is, prove that it will always stop after a finite number of operations.

13. Modify the sequential search algorithm of Figure 2.13 so that it works correctly even if the numbers in the reverse directory are not unique, that is, if the desired number occurs more than once. (For example, a single number may be listed separately under the husband's name and the wife's name.) Your modified algorithm should find *every* occurrence of *NUMBER* in the directory and print the name corresponding to every match. In addition, after all the names have been displayed, your algorithm should print how many occurrences were located. For example, if *NUMBER* occurred twice, the output of the algorithm might look something like this:

    Susan Doe

    John Doe

    A total of two occurrences were located.

14. Use the Find Largest algorithm of Figure 2.14 to help you develop an algorithm to find the median value in a list containing *N* unique numbers. The median of *N* numbers is defined as the value in the list in which approximately half the values are larger than it and half the values are smaller than it. For example, consider the following list of seven numbers.

    26, 50, 83, 44, 91, 20, 55

    The median value is 50 because three values (20, 26, and 44) are smaller and three values (55, 83, and 91) are larger. If *N* is an even value, then the

number of values larger than the median will be one greater than the number of values smaller than the median.

15. With regard to the Find Largest algorithm of Figure 2.14, if the numbers in our list were not unique and therefore the largest number could occur more than once, would the algorithm find the first occurrence? The last occurrence? Every occurrence? Explain precisely how this algorithm would behave when presented with this new condition.

16. On the sixth line of the Find Largest algorithm of Figure 2.14, there is an instruction that reads,

    While ($i \leq n$) do

    Explain exactly what would happen if we changed that instruction to read as follows:

    a. While ($i \geq n$) do

    b. While ($i < n$) do

    c. While ($i = n$) do

17. On the seventh line of the Find Largest algorithm of Figure 2.14, there is an instruction that reads,

    If $A_i >$ *largest so far* then ...

    Explain exactly what would happen if we changed that instruction to read as follows:

    a. If $A_i \geq$ *largest so far* then ...

    b. If $A_i <$ *largest so far* then ...

    Looking back at your answers in Exercises 16 and 17, what do they say about the importance of using the correct *relational operation* ($<$, $=$, $>$, $\geq$, $\leq$, $\neq$) when writing out either an iterative or conditional algorithmic primitive?

18. Refer to the pattern-matching algorithm in Figure 2.16.

    a. What is the output of the algorithm as it currently stands if our text is

       Text: We must band together and handle adversity

    and we search for the pattern "and"?

    b. How could we modify the algorithm so that it finds only the complete word *and* rather than the occurrence of the character sequence *a*, *n*, and *d* that is contained within another word, such as *band*?

19. Refer to the pattern-matching algorithm in Figure 2.16. Explain how the algorithm would behave if we accidentally omitted the statement on Line 16 that says,

    Increment *k* by 1

20. Design an algorithm that is given a positive integer *N* and determines whether *N* is a prime number, that is, not evenly divisible by any value other than 1 and itself. The output of your algorithm is either the message 'not prime', along with a factor of *N*, or the message 'prime'.

21. Write an algorithm that generates a Caesar cipher—a secret message in which each letter is replaced by the one that is *k* letters ahead of it in the alphabet, in a circular fashion. For example, if $k = 5$, then the letter *a* would be replaced by the letter *f*, and the letter *x* would be replaced by the letter *c*. (We'll talk more about the Caesar cipher and other encryption algorithms in Chapter 8.) The input to your algorithm is the text to be encoded, ending with the special symbol "$", and the value *k*. (You may assume that, except for the special ending character, the text contains only the 26 letters *a* ... *z*.) The output of your algorithm is the encoded text.

22. Design and implement an algorithm that is given as input an integer value $k \geq 0$ and a list of *k* numbers $N_1, N_2, \ldots, N_k$. Your algorithm should reverse the order of the numbers in the list. That is, if the original list contained:

    $N_1 = 5, N_2 = 13, N_3 = 8, N_4 = 27, N_5 = 10 (k = 5)$

    then when your algorithm has completed, the values stored in the list will be:

    $N_1 = 10, N_2 = 27, N_3 = 8, N_4 = 13, N_5 = 5$

23. Design and implement an algorithm that gets as input a list of k integer values $N_1, N_2, \ldots, N_k$ as well as a special value *SUM*. Your algorithm must locate a pair of values in the list N that sum to the value *SUM*. For example, if your list of values is 3, 8, 13, 2, 17, 18, 10, and the value of *SUM* is 20, then your algorithm would output either of the two values (2, 18) or (3, 17). If your algorithm cannot find any pair of values that sum to the value *SUM*, then it should print the message 'Sorry, there is no such pair of values'.

24. Instead of reading in an entire list $N_1, N_2, \ldots$ all at once, some algorithms (depending on the task to be done) read in only one element at a time and process that single element completely before inputting the next one. This can be a useful technique when the list is very big (e.g., billions of elements) and there might not be enough memory in the computer to store it in its entirety. Write an algorithm that reads in a sequence of values $V \geq 0$, one at a time, and computes the average of all the numbers. You should stop the computation when you input a value of $V = -1$. Do *not* include this negative value in your computations; it is not a piece of data but only a marker to identify the end of the list.

25. Write an algorithm to read in a sequence of values $V \geq 0$, one at a time, and determine if the list contains at least one adjacent pair of values that are identical. The end of the entire list is marked by the special value $V = -1$. For example, if you were given the following input:

    14, 3, 7, 7, 9, 1, 804, 22, −1

    the output of your algorithm should be a 'Yes' because there is at least one pair of adjacent numbers that are equal (the 7s). However, given the following input:

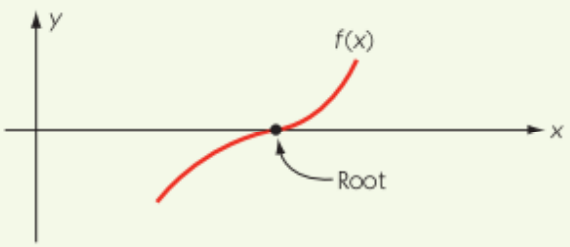
    14, 3, 7, 77, 9, 1, 804, 22, −1

    the output of your algorithm should be a 'No' because there are no adjacent pairs that are equal. You may assume in your solution that there are at least two numbers in the list.

26. Modify the algorithm that you developed in Exercise 25 so that if there is a pair of identical adjacent values, your algorithm also outputs, in addition to the phrase 'Yes', the value of the identical numbers. So, given the first list of numbers shown in Exercise 25, the output of your algorithm would be something like 'Yes, the numbers 7 and 7 are adjacent to each other in the list'.

CHALLENGE WORK

1. Design an algorithm to find the *root* of a function $f(x)$, where the root is defined as a point x such that $f(x) = 0$. Pictorially, the root of a function is the point where the graph of that function crosses the x-axis.

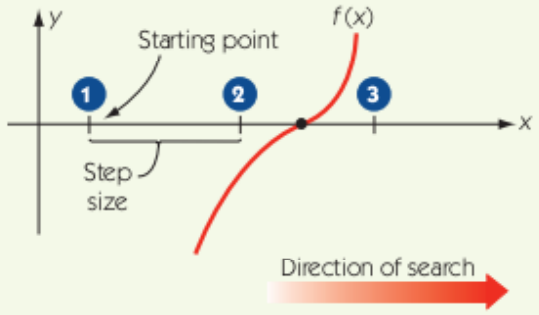


Your algorithm should operate as follows. Initially it will be given three values:

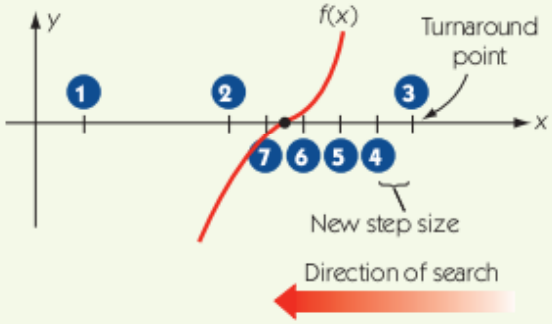
1. A starting point for the search
2. A step size
3. The accuracy desired

Your algorithm should begin at the specified starting point and begin to "walk up" the x-axis in units of step size. After taking a step, it should ask the question "Have I passed a root?" It can determine the answer to this question by seeing

whether the sign of the function has changed from the previous point to the current point. (Note that below the axis, the sign of $f(x)$ is negative; above the axis, it is positive. If it crosses the x-axis, it must change its sign.) If the algorithm has not passed a root, it should keep walking up the x-axis until it does. This is expressed pictorially as:



When the algorithm passes a root, it must do two things. First, it changes the sign of step size so that it starts walking in the reverse direction because it is now past the root. Second, it multiplies step size by 0.1, so our steps are 1/10 as big as they were before. We now repeat the operation described previously, walking down the x-axis until we pass the root.



Again, the algorithm changes the sign of step size to reverse direction and reduces it to 1/10 its previous size. As the diagrams show, we are slowly zeroing in on the root—going past it, turning around, going past it, turning around, and so forth. This iterative process stops when the algorithm passes a root and the step size is smaller than the desired accuracy. It has now bracketed the root within an interval that is smaller than the accuracy we want. At this point, it should print the midpoint of the interval and terminate.

There are many special cases that this algorithm must deal with, but in your solution you may disregard them. Assume that you will always encounter a root in your "travels" along the x-axis. After creating a solution, you might want to look at some of these special cases, such as a function that has no real roots, a starting point that is to the right of all the roots, and two roots so close together that they fall within the same step.

2. One of the most important and widely used classes of algorithms in computer science is *sorting*, the process of putting a list of elements into a predefined order, usually numeric or alphabetic. There are many different sorting algorithms, and we will look at some of them in Chapter 3. One of the simplest sorting algorithms is called *selection sort*, and it can be implemented using the tools that you have learned in this chapter. It is also one of the easiest to understand as it mimics how we often sort collections of values when we must do so by hand.

   Assume that we are given a list named A, containing eight values that we want to sort into ascending order, from smallest to largest:

A:	23	18	66	9	21	90	32	4
Position:	1	2	3	4	5	6	7	8

We first look for the largest value contained in positions 1 to 8 of list A. We can do this using something like the Find Largest algorithm that appears in Figure 2.14. In this case, the largest value is 90, and it appears in position 6. Because this is the largest value in list A, we swap it with the value in position 8 so that it is in its correct place at the back of the list. The list is now partially sorted from position 8 to position 8:

A:	23	18	66	9	21	4	32	90
Position:	1	2	3	4	5	6	7	8

We now search the array for the second largest value. Because we know that the largest value is contained in position 8, we need to search only positions 1 to 7 of list A to find the second largest value. In this case, the second largest value is 66, and it appears in position 3. We now swap the value in position 3 with the value in position 7 to get the second largest value into its correct location. This produces the following:

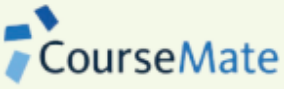
A:	23	18	32	9	21	4	66	90
Position:	1	2	3	4	5	6	7	8

The list is now partially sorted from position 7 to position 8, with those two locations holding the two largest values. The next search goes from position 1 to position 6 of list A, this time trying to locate the third largest value, and we swap that value with the number in position 6. After repeating this process seven times, the list is completely sorted. (That is because if the last seven items are in their correct place, the item in position 1 must also be in its correct place.)

Using the Find Largest algorithm shown in Figure 2.14 (which may have to be slightly modified) and the primitive pseudocode operations listed in Figure 2.9, implement the selection sort algorithm that we have just described. Assume that *n*, the size of the list, and the *n*-element list A are input to your algorithm. The output of your algorithm should be the sorted list.

3. Most people are familiar with the work of the great mathematicians of ancient Greece and Rome, such as Archimedes, Euclid, Pythagoras, and Plato. However, a great deal of important work in arithmetic, geometry, algebra, number theory, and logic was carried out by scholars working in Egypt, Persia, India, and China. For example, the concept of zero was first developed in India, and positional numbering systems (like our own decimal system) were developed and used in China, India, and the Middle East long before they made their way to Europe. Read about the work of some mathematician (such as Al-Khowarizmi) from one of these places or another place, and write a paper describing his or her contributions to mathematics, logic, and (ultimately) computer science.

ADDITIONAL RESOURCES



For additional print and/or online resources relevant to this chapter, visit the CourseMate for this text at login.cengagebrain.com.

CHAPTER

3 The Efficiency of Algorithms

CHAPTER TOPICS

3.1 Introduction

Finding algorithms to solve problems of interest is an important part of computer science. Any algorithm that is developed to solve a specific problem has, by definition, certain required characteristics (see the formal definition in Chapter 1, Section 1.3.1), but are some algorithms better than others? That is, are there other desirable but nonessential characteristics of algorithms?

Consider the automobile: There are certain features that are part of the "definition" of a car, such as four wheels and an engine. These are the basics. However, when purchasing a car, we almost certainly take into account other things, such as ease of handling, style, and fuel efficiency. This analogy is not as superficial as it seems—the properties that make better algorithms are in fact very similar.

3.2 Attributes of Algorithms

First and foremost, we expect *correctness* from our algorithms. An algorithm intended to solve a problem must, again by formal definition, give a result and then halt. But this is not enough; we also want the result to be a correct solution to the problem. You could consider this an inherent property of the definition of an algorithm (like a car being capable of transporting us where we want to go), but it bears emphasizing. An elegant and efficient algorithm that gives wrong results for the problem at hand is worse than useless. It can lead to mistakes that are enormously expensive or even fatal.

Determining that an algorithm gives correct results might not be as straightforward as it seems. For one thing, our algorithm might indeed be providing correct results—but to the wrong problem. This can happen when we design an algorithm without a thorough understanding of the real problem we are trying to solve, and it is one of the most common causes of "incorrect" algorithms. Also, once we understand the problem, the algorithm must provide correct results for all possible input values, not just for those values that are the most likely to occur. Do we know what all those

correct results are? Probably not, or we would not be writing an algorithm to solve this problem. But there may be a certain standard against which we can check the result for reasonableness, thus giving us a way to determine when a result is obviously incorrect. In some cases, as noted in Chapter 1, the correct result may be an error message saying that there is no correct answer. There may also be an issue of the accuracy of the result we are willing to accept as correct. If the "real" answer is π, for example, then we can only approximate its decimal value. Is 3.14159 close enough to "correct"? Is 3.1416 close enough? What about 3.14? Computer scientists often summarize these two views of correctness by asking, "Are we solving the right problem? Are we solving the problem right?"

If an algorithm to solve a problem exists and we determine, after taking into account all the considerations of the previous paragraph, that it gives correct results, what more can we ask? To many mathematicians, this would be the end of the matter. After all, once a solution has been obtained and shown to be correct, it is no longer of interest (except possibly for use in obtaining solutions to other problems). This is where computer science differs significantly from theoretical disciplines such as pure mathematics and begins to take on an "applied" character more closely related to engineering or applied mathematics. The algorithms developed by computer scientists are not merely of academic interest. They are also intended to be *used*.

Suppose, for example, that a road to the top of a mountain is to be built. An algorithmic solution exists that gives a correct answer for this problem in the sense that a road is produced: Just build the road straight up the mountain. Problem solved. But the highway engineer knows that the road must be usable by real traffic and that this constraint limits the grade of the road. The existence and correctness of the algorithm is not enough; there are practical considerations as well.

The practical considerations for computer science arise because the algorithms developed are executed in the form of computer programs running on real computers to solve problems of interest to real people. Let's consider the "people aspect" first. A computer program is seldom written to be used only once to solve a single instance of a problem. It is written to solve many instances of that problem with many different input values, just as the sequential search algorithm of Chapter 2 would be used many times with different lists of telephone numbers and associated names, and different target *NUMBER* values. Furthermore, the problem itself does not usually "stand still." If the program is successful, people will want to use it for slightly different versions of the problem, which means they will want the program slightly enhanced to do more things. Therefore, after a program is written, it needs to be maintained, both to fix any errors that are uncovered through repeated usage with different input values and to extend the program to meet new requirements. A great deal of time and money are devoted to *program maintenance*. The person who has to modify a program, either to correct errors or to expand its functionality, often is not the person who wrote the original program. To make program maintenance as easy as possible, the algorithm the program uses should be easy to understand. *Ease of understanding*, clarity, "ease of handling"—whatever you want to call it—is a highly desirable characteristic of an algorithm.

On the other hand, there is a certain satisfaction in having an "elegant" solution to a problem. *Elegance* is the algorithmic equivalent of style. The classic example, in mathematical folklore, is the story of the German mathematician Karl Frederick Gauss (1777–1855), who was asked as a schoolchild to add up the numbers from 1 to 100. The straightforward algorithm of adding 1 + 2 + 3 + 4 + ... + 100 by adding one number at a time can be expressed in pseudocode as follows:

1. Set the value of *sum* to 0
2. Set the value of *x* to 1
3. While *x* is less than or equal to 100, do Steps 4 and 5
4. Add *x* to *sum*
5. Add 1 to the value of *x*
6. Print the value of *sum*
7. Stop

This algorithm can be executed to find that the sum has the value 5,050. It is fairly easy to read this pseudocode and understand how the algorithm works. It is also fairly clear that if we want to change this algorithm to one that adds the numbers from 1 to 1,000, we only have to change the loop condition to

3. While *x* is less than or equal to 1,000, do Steps 4 and 5

However, Gauss noticed that the numbers from 1 to 100 could be grouped into 50 pairs of the form

$$1 + 100 = 101$$
$$2 + 99 = 101$$
$$\vdots$$
$$50 + 51 = 101$$

so that the sum equals $50 \times 101 = 5{,}050$. This is certainly an elegant and clever solution, but is it easy to understand? If a computer program just said to multiply

$$\left(\frac{100}{2}\right)101$$

with no further explanation, we might guess how to modify the program to add up the first 1,000 numbers, but would we really grasp what was happening enough to be sure the modification would work? (The Practice Problems at the end of this section discuss this.) Sometimes elegance and ease of understanding work at cross-purposes; the more elegant the solution, the more difficult it may be to understand. Do we win or lose if we have to trade ease of understanding for elegance? Of course, if an algorithm has both characteristics—ease of understanding and elegance—that's a plus.

Now let's consider the real computers on which programs run. Although these computers can execute instructions very rapidly and have some memory in which to store information, time and space are not unlimited resources. The computer scientist must be conscious of the resources consumed by a given algorithm, and if there is a choice between two (correct) algorithms that perform the same task, the one that uses fewer resources is preferable. The term used to describe an algorithm's careful use of resources is **efficiency**. Efficiency, in addition to *correctness*, *ease of understanding*, and *elegance*, is an extremely desirable attribute of an algorithm.

Because of the rapid advances in computer technology, today's computers have much more memory capacity and execute instructions much more quickly than computers of just a few years ago. Efficiency in algorithms might seem to be a moot point; we can just wait for the next generation of technology and it won't matter how much time or space is required. There is some truth to this, but as computer memory capacity and processing speed increase, people find more complex problems to be solved, so the boundaries of the computer's resources continue to be pushed. Furthermore, we will see in this chapter that there are algorithms that consume so many resources that they will never be practical, no matter what advances in computer technology occur.

How should we measure the time and space consumed by an algorithm to determine whether it is efficient? Space efficiency can be judged by the amount of information the algorithm must store in the computer's memory to do its job, in addition to the initial data on which the algorithm is operating. If it uses only a few extra memory locations while processing the input data, the algorithm is relatively space efficient. If the algorithm

requires almost as much additional storage as the input data itself takes up, or even more, then it is relatively space inefficient.

How can we measure the time efficiency of an algorithm? Consider the sequential search algorithm shown in Figure 2.13, which finds a person's name given his or her telephone number in a reverse telephone directory in which the telephone numbers are not arranged in numerical order. How about running the algorithm on a real computer and timing it to see how many seconds (or maybe what small fraction of a second) it takes to find a name or announce that the phone number is not present? The difficulty with this approach is that there are three factors involved, each of which can affect the answer to such a degree as to make whatever number of seconds we come up with rather meaningless.

1. On which computer will we run the algorithm? Should we use a smartphone, a modest laptop, or a supercomputer capable of doing many trillions of calculations per second?
2. Which reverse telephone book (list of numbers) will we use: one limited to a relatively small geographic area or one that covers all listed numbers in the North American Numbering Plan?
3. Which number will we search for? What if we pick a number that happens to be first in the list? What if it happens to be last in the list?

Simply timing the running of an algorithm is more likely to reflect machine speed or variations in input data than the efficiency (or lack thereof) of the algorithm itself.

This is not to say that you can't obtain meaningful information by timing an algorithm. Using the same input data (for example, searching for the same number in the same reverse directory) and timing the algorithm on different machines gives a comparison of machine speeds because the task is identical. Using the same machine and the same reverse directory, but searching for different numbers, gives an indication of how the choice of *NUMBER* affects the algorithm's running time on that particular machine. This type of comparative timing is called **benchmarking**. Benchmarks are useful for rating one machine against another and for rating how sensitive a particular algorithm is with respect to variations in input on one particular machine.

However, what we mean by an algorithm's time efficiency is an indication of the amount of "work" required by the algorithm itself. It is a measure of the inherent efficiency of the method, independent of the speed of the machine on which it executes or the specific input data being processed. Is the amount of work an algorithm does the same as the number of instructions it executes? Not all instructions do the same things, so perhaps they should not all be "counted" equally. Some instructions are carrying out work that is fundamental to the way the algorithm operates, whereas other instructions are carrying out peripheral tasks that must be done in support of the fundamental work. To measure time efficiency, we identify the fundamental unit (or units) of work of an algorithm and count how many times the work unit is executed. Later in this chapter, we will see why we can ignore peripheral tasks.

Practice Problems

1. Use Gauss's approach to find a formula for the sum of the numbers from 1 to *n*,

   $1 + 2 + 3 + \ldots + n$

   where *n* is an even number. Your formula will be an expression involving *n*.
2. Test your formula from Practice Problem 1 for the following sums:
   a. $1 + 2$
   b. $1 + 2 + \ldots + 6$
   c. $1 + 2 + \ldots + 10$
   d. $1 + 2 + \ldots + 100$
   e. $1 + 2 + \ldots + 1000$
3. Now see if the same formula from Practice Problem 1 works when *n* is odd; try it on the following sums:
   a. $1 + 2 + 3$
   b. $1 + 2 + \ldots + 5$
   c. $1 + 2 + \ldots + 9$

3.3 Measuring Efficiency

The study of the efficiency of algorithms is called the **analysis of algorithms**, and it is an important part of computer science. As a first example of the analysis of an algorithm, we'll look at the sequential search algorithm that we created to solve the reverse telephone lookup problem.

3.3.1 Sequential Search

The pseudocode description of the **sequential search algorithm** from Chapter 2 appears in Figure 3.1, where we have assumed that the list contains *n* entries instead of 10,000 entries.

The central unit of work is the comparison of the *NUMBER* being searched for against a single phone number in the list. The essence of the algorithm is the repetition of this task against successive numbers in the list until *NUMBER* is found or the list is exhausted. The comparison takes place at Step 4, within the loop body composed of Steps 4 through 7. Peripheral tasks include setting the initial value of the index *i* and the initial value of

FIGURE 3.1

1. Get values for *NUMBER*, n, $T_1, \ldots, T_n$ and $N_1, \ldots, N_n$
2. Set the value of i to 1 and set the value of *Found* to NO
3. While (*Found* = NO) and ($i \leq n$) do Steps 4 through 7
4. If *NUMBER* is equal to the ith number on the list, T_i, then
5. Print the name of the corresponding person, N_i
6. Set the value of *Found* to YES
   Else (*NUMBER* is not equal to T_i)
7. Add 1 to the value of i
8. If (*Found* = NO) then
9. Print the message 'Sorry, this number is not in our directory'
10. Stop

Sequential search algorithm

Found, writing the output, adjusting *Found*, and moving the index forward in the list of numbers. Why are these considered peripheral tasks?

Setting the initial value of the index and the initial value of *Found* requires executing a single instruction, done at Step 2. Writing output requires executing a single instruction, either at Step 5 if *NUMBER* is in the list or at Step 9 if *NUMBER* is not in the list. Note that instruction 5, although it is part of the loop, writes output at most once (if *NUMBER* equals T_i). Similarly, setting *Found* to YES occurs at most once (if *NUMBER* equals T_i) at Step 6. We can ignore the small contribution of these single-instruction executions to the total work done by the algorithm.

Moving the index forward is done once for each comparison, at Step 7. We can get a good idea of the total amount of work the algorithm does by simply counting the number of comparisons and then multiplying by some constant factor to account for the index-moving task. The constant factor could be 2 because we do one index move for each comparison, so we would double the work. It could be less because it is less work to add 1 to i than it is to compare *NUMBER* digit by digit against T_i. As we will see later, the precise value of this constant factor is not very important.

So again, the basic unit of work in this algorithm is the comparison of *NUMBER* against a list element. One comparison is done at each pass through the loop in Steps 4 through 7, so we must ask how many times the loop is executed. Of course, this depends on when, or if, we find *NUMBER* in the list.

The minimum amount of work is done if *NUMBER* is the very first number in the list. This requires only one comparison because *NUMBER* has then been found and the algorithm exits the loop after only one pass. This is the *best case*, requiring the least work. The *worst case*, requiring the maximum amount of work, occurs if *NUMBER* is the very last number in the list or is absent. In either of these situations, *NUMBER* must be compared against all n numbers in the list before the loop terminates because *FOUND* gets set to

YES (if *NUMBER* is the last number in the list) or because the value of the index i exceeds n (if *NUMBER* is not in the list).

When *NUMBER* occurs somewhere in the middle of the list, it requires somewhere between 1 (the best case) and n (the worst case) comparisons. If we were to run the sequential search algorithm many times with random *NUMBERs* occurring at various places in the list and count the number of comparisons done each time, we would find that the average number of comparisons done is about $n/2$. (The exact average is actually slightly higher than $n/2$; see Exercise 5 at the end of the chapter.) It is not hard to explain why an average of approximately $n/2$ comparisons are done (or the loop is executed approximately $n/2$ times) when *NUMBER* is in the list. If *NUMBER* occurs halfway down the list, then roughly $n/2$ comparisons are required; random *NUMBERs* in the list occur before the halfway point about half the time and after the halfway point about half the time, and these cases of less work and more work balance out.

This means that the average number of comparisons needed to find a *NUMBER* that occurs in a 10-element list is about 5, in a 100-element list about 50, and in a 1,000-element list about 500. With small values of n—say, a few hundred or a few thousand numbers—the values of $n/2$ (the average case) or n (the worst case) are small enough that a computer could execute the algorithm quickly and get the desired answer in a fraction of a second. However, computers are generally used to solve not tiny problems but very large ones. Therefore, we are usually interested in the behavior of an algorithm as the size of a problem (n) gets very, very large. In our reverse directory example, the total number of publicly listed telephone numbers may be as large as 350,000,000. (Cell phone numbers are usually unlisted.) If the sequential search algorithm were executed on a computer that could do 50,000 comparisons per second, it would require on the average about

$$\frac{350{,}000{,}000}{2} \text{ comparisons} \times \frac{1}{50{,}000} \text{ seconds/comparison} = 3{,}500 \text{ seconds}$$

or nearly an hour just to do the comparisons necessary to locate a specific number. Including the constant factor for advancing the index, the actual time needed would be even greater. It would require almost 2 hours to do the comparisons required to determine that a number is not in the reverse directory! Sequential search is not sufficiently time efficient for large values of n to be useful as a reverse directory lookup algorithm.

Information about the number of comparisons required to perform the sequential search algorithm on a list of n numbers is summarized in Figure 3.2. Note that the values for both the worst case and the average case depend on n, the number of numbers in the list. The bigger the list, the more work must be done to search it. Few algorithms do the same amount of work on large inputs as on small inputs, simply because most algorithms process the input data, and more data to process means more work. The work an algorithm does can usually be expressed in terms of a formula that depends on the size of the problem input. In the case of searching a list of numbers, the input size is the length of the list.

Let's say a word about the space efficiency of sequential search. The algorithm stores the list of numbers/names and the target *NUMBER* as part of the

FIGURE 3.2

Best Case	Worst Case	Average Case
1	n	$n/2$

Number of comparisons to find NUMBER in a list of n numbers using sequential search

input. The only additional memory required is storage for the index value i and the *Found* indicator. Two single additional memory locations are insignificant compared with the size of the list, just as executing a single instruction to initialize the value of i and *Found* is insignificant beside the repetitive comparison task. Therefore, sequential search uses essentially no more memory storage than the original input requires, so it is very space efficient.

While we have presented sequential search in the context of searching for a particular telephone number from an unsorted list of numbers, the algorithm applies to searching any unordered list of items for one particular "target" item. And again, although it does not seem to be a time-efficient algorithm for a large value of n, it is the best approach available to search an unsorted list.

3.3.2 Order of Magnitude—Order *n*

When we analyzed the time efficiency of the sequential search algorithm, we glossed over the contribution of the constant factor for the peripheral work. To see why this constant factor doesn't particularly matter, we need to understand a concept called *order of magnitude*.

The worst-case behavior of the sequential search algorithm on a list of n items requires n comparisons, and if c is a constant factor representing the peripheral work, it requires cn total work. Suppose that c has the value 2. Then the values of n and $2n$ are

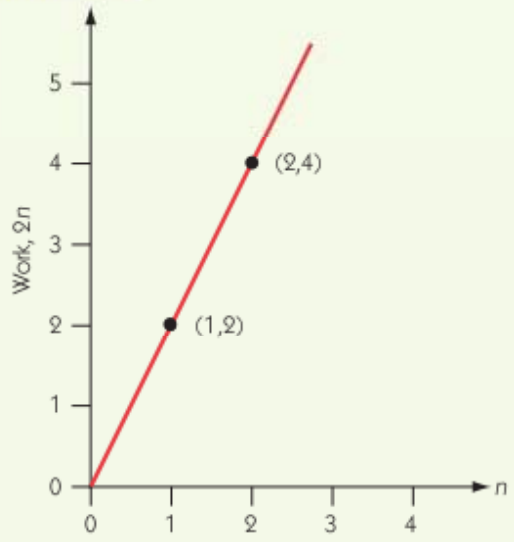
n	$2n$
1	2
2	4
3	6

and so on

These values are shown in Figure 3.3, which illustrates how the value of $2n$, which is the total work, changes as n changes. We can add to this graph to show how the value of cn changes as n changes, where $c = 1$ or $c = 1/2$ as well as $c = 2$ (see Figure 3.4; these values of c are completely arbitrary). Figure 3.5 presents a different view of the growth rate of cn as n changes for these three values of c.

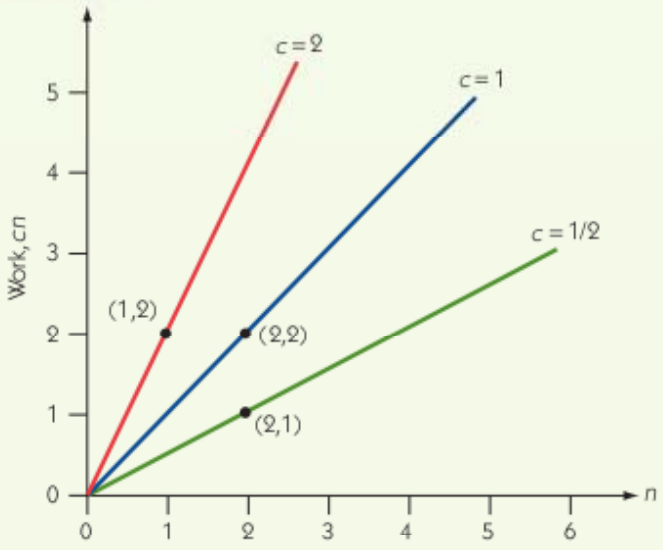
Both Figure 3.4 and Figure 3.5 show that the amount of work cn increases as n increases, but at different rates. The work grows at the same rate as n when $c = 1$, at twice the rate of n when $c = 2$, and at half the rate of n

FIGURE 3.3



Work = 2*n*

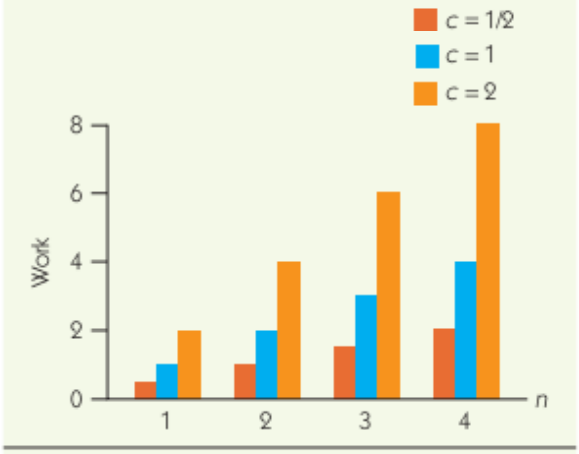
FIGURE 3.4



Work = *cn* for various values of *c*

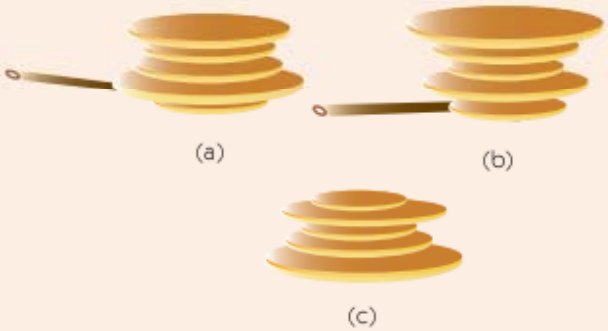
when $c = 1/2$. However, Figure 3.4 also shows that all of these graphs follow the same basic straight-line shape of n. Anything that varies as a constant times n (and whose graph follows the basic shape of n) is said to be of order of magnitude n, written $\Theta(n)$ and pronounced "order n." We will classify algorithms according to the order of magnitude of their time efficiency. Sequential search is therefore an $\Theta(n)$ algorithm (an order-n algorithm) in both the worst case and the average case.

FIGURE 3.5



Growth of work = *cn* for various values of *c*

Flipping Pancakes



A problem posed in the *American Mathematical Monthly* in 1975 by Jacob Goodman concerned a waiter in a café where the cook produced a stack of pancakes of varying sizes. The waiter, on the way to delivering the stack to the customer, attempted to arrange the pancakes in order by size, with the largest on the bottom. The only action available was to stick a spatula into the stack at some point and flip the entire stack above that point. The question is: What is the maximum number of flips ever needed for any stack of n pancakes? This number, P_n, is known as the *nth pancake number.*

Here's a fairly simple algorithm to arrange the pancakes. Put the spatula under the largest pancake, as shown in (a) in the figure, and flip. This puts the largest pancake on top [(b) in the figure]. Put the spatula at the bottom of the unordered section (in this case at the bottom) and flip. This puts the largest pancake on the bottom [(c) in the figure], where it belongs. Repeat with the rest of the pancakes. Each pancake therefore requires two flips, which would give a total of $2n$ flips required. But the last two pancakes require at most one flip; if they are already in order, no flips are needed, and if they are out of order, only one flip is needed. So this algorithm requires at most $2(n - 2) + 1 = 2n - 3$ flips in the worst case, which means that $P_n \leq 2n - 3$. Are there other algorithms that require fewer flips in the worst case?

A faculty member at Harvard University posed this question to his class; several days later, a sophomore from the class came to his office with a better algorithm. This algorithm, which requires at most $(5n + 5)/3$ flips, was published in the journal *Discrete Mathematics* in 1979. The authors were William Gates (the student) and Christos Papadimitriou.

Yes, that William Gates!

3.3.3 Selection Sort

The sequential search algorithm solves a very common problem: **searching** a list of items to locate a particular item. Another very common problem is that of **sorting** a list of items into order—either alphabetical or numerical order. The registrar at your institution sorts students in a class by name, a mail-order business sorts its customer list by name, and the IRS sorts tax records by Social Security number. In this section, we'll examine a sorting algorithm and analyze its efficiency.

Practice Problem

Using the information in Figure 3.2, fill in the following table for the number of comparisons required in the sequential search algorithm.

n	Best Case	Worst Case	Average Case
10			
50			
100			
1,000			
10,000			
100,000			

Suppose we have a list of numbers to sort into ascending order—for example, 5, 7, 2, 8, 3. The result of sorting this list is the new list 2, 3, 5, 7, 8. The selection sort algorithm performs this task. The selection sort "grows" a sorted subsection of the list from the back to the front. We can look at "snapshots" of the progress of the algorithm on our sample list, using a vertical line as the marker between the unsorted section at the front of the list and the sorted section at the back of the list in each case. At first the sorted subsection is empty; that is, the entire list is unsorted. This is how the list looks when the algorithm begins.

5, 7, 2, 8, 3 |

Unsorted subsection (entire list) Sorted subsection (empty)

Later, the sorted subsection of the list has grown from the back so that some of the list members are in the right place.

5, 3, 2, | 7, 8

Unsorted subsection Sorted subsection

Finally, the sorted subsection of the list contains the entire list; there are no unsorted numbers, and the algorithm stops.

| 2, 3, 5, 7, 8

Unsorted subsection (empty) Sorted subsection (entire list)

At any point, then, there is both a sorted and an unsorted section of the list. A pseudocode version of the algorithm is shown in Figure 3.6.

Before we illustrate this algorithm at work, take a look at Step 4, which finds the largest number in some list of numbers. We developed an algorithm

FIGURE 3.6

1. Get values for *n* and the *n* list items
2. Set the marker for the unsorted section at the end of the list
3. While the unsorted section of the list is not empty, do Steps 4 through 6
4. Select the largest number in the unsorted section of the list
5. Exchange this number with the last number in the unsorted section of the list
6. Move the marker for the unsorted section left one position
7. Stop

Selection sort algorithm

for this task in Chapter 2 (Figure 2.14). A detailed version of the selection sort algorithm would replace Step 4 with the instructions from this existing algorithm. New algorithms can be built up from "parts" consisting of previous algorithms, just as a recipe for pumpkin pie might begin with the instruction, "Prepare crust for a one-crust pie." The recipe for pie crust is a previous algorithm that is now being used as one of the steps in the pumpkin pie algorithm.

Let's follow the selection sort algorithm. Initially, the unsorted section is the entire list, so Step 2 sets the marker at the end of the list.

5, 7, 2, 8, 3 |

Step 4 says to select the largest number in the unsorted section—that is, in the entire list. This number is 8. Step 5 says to exchange 8 with the last number in the unsorted section (the whole list). To accomplish this exchange, the algorithm must determine not only that 8 is the largest value but also the location in the list where 8 occurs. The Find Largest algorithm from Chapter 2 provides both these pieces of information. The exchange to be done is

5, 7, 2, 8, 3 |

After this exchange and after the marker is moved left as instructed in Step 6, the list looks like

5, 7, 2, 3 | 8

The number 8 is now in its correct position at the end of the list. It becomes the sorted section of the list, and the first four numbers are the unsorted section.

The unsorted section is not empty, so the algorithm repeats Step 4 (find the largest number in the unsorted section); it is 7. Step 5 exchanges 7 with the last number in the unsorted section, which is 3.

5, 7, 2, 3 | 8

After the marker is moved, the result is

5, 3, 2 | 7, 8

The sorted section is now 7, 8 and the unsorted section is 5, 3, 2.

Repeating the loop of Steps 4 through 6 again, the algorithm determines that the largest number in the unsorted section is 5, and exchanges it with 2, the last number in the unsorted section.

5, 3, 2 | 7, 8

After the marker is moved, we get

2, 3 | 5, 7, 8

Now the unsorted section (as far as the algorithm knows) is 2, 3. The largest number here is 3. Exchanging 3 with the last number of the unsorted section (that is, with itself) produces no change in the list ordering. The marker is moved, giving

2 | 3, 5, 7, 8

When the only part of the list that is unsorted is the single number 2, there is also no change in the list ordering produced by carrying out the exchange. The marker is moved, giving

| 2, 3, 5, 7, 8

The unsorted section of the list is empty, and the algorithm terminates.

To analyze the amount of work the selection sort algorithm does, we must first decide on the unit of work to count. When we analyzed sequential search, the unit of work that we measured was the comparison between the item being searched for and the items in the list. At first glance, there seem to be no comparisons of any kind going on in the selection sort. Remember, however, that there is a subtask within the selection sort: the task of finding the largest number in a list. The algorithm from Chapter 2 for finding the largest value in a list begins by taking the first number in the list as the largest so far. The largest-so-far value is compared against successive numbers in the list; if a larger value is found, it becomes the largest so far.

When the selection sort algorithm begins, the largest-so-far value, initially the first number, must be compared with all the other numbers in the list. If there are n numbers in the list, $n - 1$ comparisons must be done. The next time through the loop, the last number is already in its proper place, so it is never again involved in a comparison. The largest-so-far value, again initially the first number, must be compared with all the other numbers in the unsorted part of the list, which will require $n - 2$ comparisons. The number of comparisons keeps decreasing as the length of the unsorted section of the list gets smaller, until finally only one comparison is needed. The total number of comparisons is

$$(n - 1) + (n - 2) + (n - 3) + \ldots + 3 + 2 + 1$$

Reviewing our sample problem, we can see that the following comparisons are done:

- To put 8 in place in the list 5, 7, 2, 8, 3 |
  Compare 5 (largest so far) to 7
  7 becomes largest so far
  Compare 7 (largest so far) to 2
  Compare 7 (largest so far) to 8
  8 becomes largest so far
  Compare 8 to 3
  8 is the largest
  Total number of comparisons: 4 (which is 5 – 1)
- To put 7 in place in the list 5, 7, 2, 3 | 8
  Compare 5 (largest so far) to 7
  7 becomes largest so far
  Compare 7 to 2
  Compare 7 to 3
  7 is the largest
  Total number of comparisons: 3 (which is 5 – 2)
- To put 5 in place in the list 5, 3, 2 | 7, 8
  Compare 5 (largest so far) to 3
  Compare 5 to 2
  5 is the largest
  Total number of comparisons: 2 (which is 5 – 3)
- To put 3 in place in the list 2, 3 | 5, 7, 8
  Compare 2 (largest so far) to 3
  3 is the largest
  Total number of comparisons: 1 (which is 5 – 4)

To put 2 in place requires no comparisons; there is only one number in the unsorted section of the list, so it is of course the largest number. It gets exchanged with itself, which produces no effect. The total number of comparisons is $4 + 3 + 2 + 1 = 10$.

The sum

$$(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

turns out to be equal to

$$\left(\frac{n-1}{2}\right)n = \frac{1}{2}n^2 - \frac{1}{2}n$$

(Recall from earlier in this chapter how Gauss computed a similar sum.) For our example with five numbers, this formula says that the total number of comparisons is (using the first version of the formula):

$$\left(\frac{5-1}{2}\right)5=\left(\frac{4}{2}\right)5=(2)5=10$$

which is the number of comparisons we had counted.

Figure 3.7 uses this same formula

$$\frac{1}{2}n^2-\frac{1}{2}n$$

to compute the comparisons required for larger values of n. Remember that n is the size of the list we are sorting. If the list becomes 10 times longer, the work increases by much more than a factor of 10; it increases by a factor closer to 100, which is 10^2.

The selection sort algorithm not only does comparisons, it also does exchanges. Even if the largest number in the unsorted section of the list is already at the end of the unsorted section, the algorithm exchanges this number with itself. Therefore, the algorithm does n exchanges, one for each position in the list to put the correct value in that position. With every exchange, the marker gets moved. However, the work contributed by exchanges and marker moving is so much less than the amount contributed by comparisons that it can be ignored.

We haven't talked here about a best case, a worst case, or an average case for the selection sort. This algorithm does the same amount of work no matter how the numbers are initially arranged. It has no way to recognize, for example, that the list might already be sorted at the outset.

A word about the space efficiency of the selection sort: The original list occupies n memory locations, and this is the major space requirement. Some storage is needed for the marker between the unsorted and sorted sections and for keeping track of the largest-so-far value and its location in the list, used in Step 4. Surprisingly, the process of exchanging two values at Step 5 also requires an extra storage location. Here's why. If the two numbers to be

FIGURE 3.7

Length n of List to Sort	n^2	Number of Comparisons Required
10	100	45
100	10,000	4,950
1,000	1,000,000	499,500

Comparisons required by selection sort

exchanged are at position X and position Y in the list, we might think the following two steps will exchange these values:

1. Copy the current value at position Y into position X
2. Copy the current value at position X into position Y

The problem is that after Step 1, the value at position X is the same as that at position Y. Step 2 does not put the original value of X into position Y. In fact, we don't even have the original value of position X anymore. In Figure 3.8(a), we see the original X and Y values. At Figure 3.8(b), after execution of Step 1, the current value of position Y has been copied into position X, writing over what was there originally. At Figure 3.8(c), after execution of Step 2, the current value at position X (which is the original Y value) has been copied into position Y, but the picture looks the same as Figure 3.8(b).

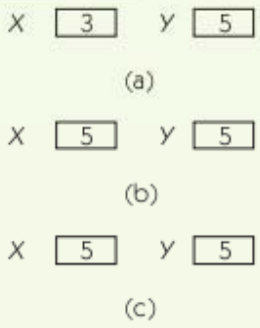
Here's the correct algorithm, which makes use of one extra temporary storage location that we'll call T.

1. Copy the current value at position X into location T
2. Copy the current value at position Y into position X
3. Copy the current value at location T into position Y

Figure 3.9 illustrates that this algorithm does the job. In Figure 3.9(a), the temporary location contains an unknown value. After execution of Step 1 (Figure 3.9(b)), it holds the current value of X. When Y's current value is put into X at Step 2 (Figure 3.9(c)), T still holds the original X value. After Step 3 (Figure 3.9(d)), the current value of T goes into position Y, and the original values of X and Y have been exchanged. (Step 5 of the selection sort algorithm is thus performed by another algorithm, just as Step 4 is.)

All in all, the extra storage required for the selection sort, over and above that required to store the original list, is slight. Selection sort is space efficient.

FIGURE 3.8



An attempt to exchange the values at X and Y

FIGURE 3.9

X [3] Y [5] T [ ]

(a)

X [3] Y [5] T [3]

(b)

X [5] Y [5] T [3]

(c)

X [5] Y [3] T [3]

(d)

Exchanging the values at *X* and *Y*

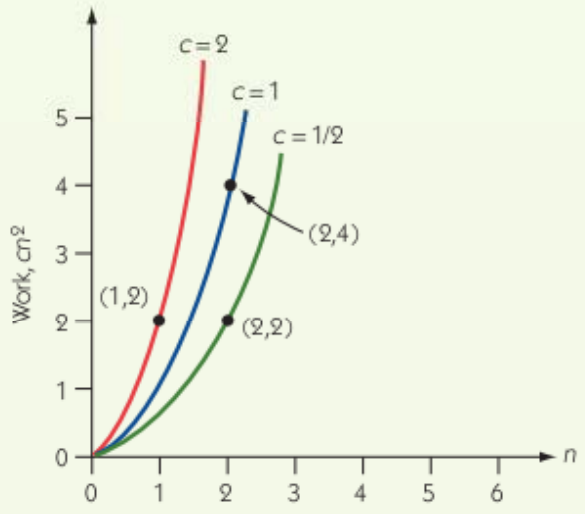
Practice Problems

1. For each of the following lists, perform a selection sort and show the list after each exchange that has an effect on the list ordering:
   a. 4, 8, 2, 6
   b. 12, 3, 6, 8, 2, 5, 7
   c. D, B, G, F, A, C, E, H
   d. 3, 7, 12, 16, 21
2. How many comparisons are required to sort each of the four lists shown in Practice Problem 1? How many exchanges?

3.3.4 Order of Magnitude—Order n^2

We saw that the number of comparisons done by the selection sort algorithm does not grow at the same rate as the problem size n; it grows at approximately the *square* of that rate. An algorithm that does cn^2 work for any constant c is order of magnitude n^2, or $\Theta(n^2)$. Figure 3.10 shows how cn^2 changes as n changes, where $c = 1$, 2, and 1/2. The work grows at the same rate as n^2 when $c = 1$, at twice that rate when $c = 2$, and at half that rate when $c = 1/2$. But all three graphs in Figure 3.10 follow the basic shape of n^2, which is different from all of the straight-line graphs that are of $\Theta(n)$. Thus,

FIGURE 3.10

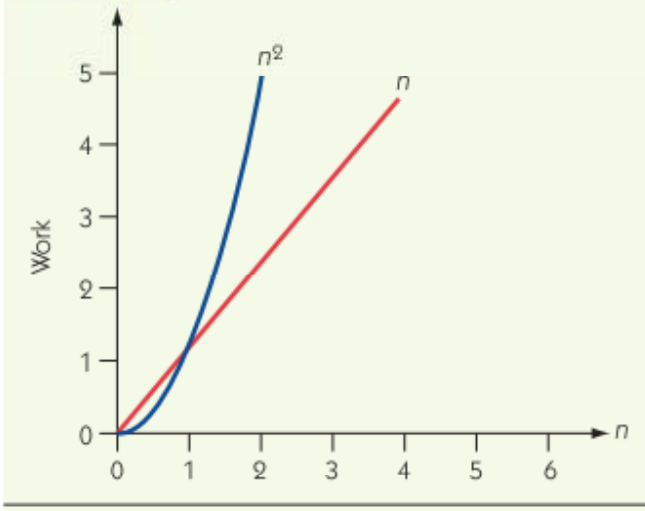


Work = cn^2 for various values of c

we have come up with two different "shape classifications": one including all graphs that are $\Theta(n)$ and the other including all graphs that are $\Theta(n^2)$.

If it is not important to distinguish among the various graphs that make up a given order of magnitude, why is it important to distinguish between the two different orders of magnitude n and n^2? We can find the answer by comparing the two basic shapes n and n^2, as is done in Figure 3.11.

FIGURE 3.11

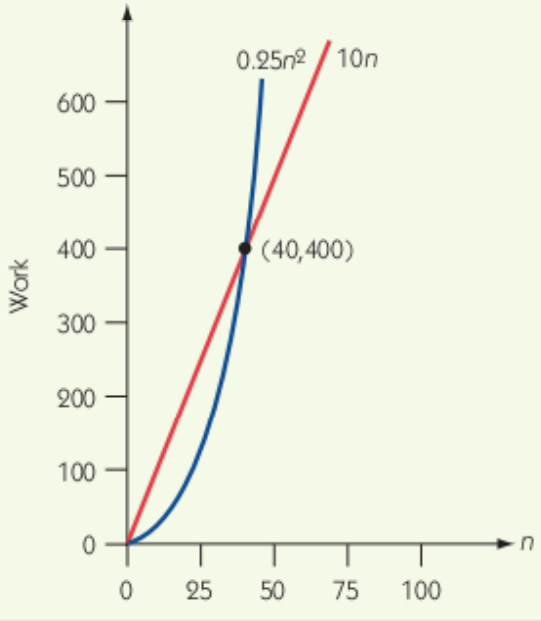


A comparison of n and n^2

Figure 3.11 illustrates that n^2 grows at a much faster rate than n. The two curves cross at the point (1,1), and for any value of n larger than 1, n^2 has a value increasingly greater than n. Furthermore, anything that is order of magnitude n^2 eventually has larger values than anything that is of order n, no matter what the constant factors are. For example, Figure 3.12 shows that if we choose a graph that is $\Theta(n^2)$ but has a small constant factor (to keep the values low), say $0.25n^2$, and a graph that is $\Theta(n)$ but has a larger constant factor (to pump the values up), say $10n$, it is still true that the $\Theta(n^2)$ graph eventually has larger values. (Note that the vertical scale and the horizontal scale are different.)

Selection sort is an $\Theta(n^2)$ algorithm (in all cases) and sequential search is an $\Theta(n)$ algorithm (in the worst case), so these two algorithms are different orders of magnitude. Because these algorithms solve two different problems, this is somewhat like comparing apples and oranges—what does it mean? But suppose we have two different algorithms that solve the same problem and count the same units of work but have different orders of magnitude. Suppose that algorithm A does $0.0001n^2$ units of work to solve a problem with input size n and that algorithm B does $100n$ of the same units of work to solve the same problem. Here algorithm B's factor of 100 is *1 million times larger* than algorithm A's factor of 0.0001. Nonetheless, when the problem gets large enough, the inherent inefficiency of algorithm A causes it to do more work than algorithm B. Figure 3.13 shows that the "crossover" point occurs at a value of 1,000,000 for n. At this point, the two algorithms do the same amount of work and therefore take the same amount of time

FIGURE 3.12



For large enough n, $0.25n^2$ has larger values than $10n$

FIGURE 3.13

	Number of Work Units Required	
	Algorithm A	Algorithm B
n	$0.0001n^2$	$100n$
1,000	100	100,000
10,000	10,000	1,000,000
100,000	1,000,000	10,000,000
1,000,000	100,000,000	100,000,000
10,000,000	10,000,000,000	1,000,000,000

A comparison of two extreme $\Theta(n^2)$ and $\Theta(n)$ algorithms

to run. For larger values of n, the order-n^2 algorithm A runs increasingly slower than the order-n algorithm B. (Input sizes in the millions are not that uncommon; the list of worldwide active Amazon.com customer accounts contained 244 million people as of mid-2014.)

As we have seen, if an $\Theta(n^2)$ algorithm and an $\Theta(n)$ algorithm exist for the same task, then for large enough n, the $\Theta(n^2)$ algorithm does more work and takes longer to execute, regardless of the constant factors for peripheral work. *This is the rationale for ignoring constant factors and concentrating on the basic order of magnitude of algorithms.*

As an analogy, the two shape classifications $\Theta(n^2)$ and $\Theta(n)$ may be thought of as two different classes of transportation, the "walking" class and the "driving" class, respectively. The walking class is fundamentally more time consuming than the driving class. Walking can include jogging, running, and leisurely strolling (which correspond to different values for c), but compared with any form of driving, these all proceed at roughly the same speed. The driving class can include driving a MINI Cooper and driving a Ferrari (which correspond to different values for c), but compared with any form of walking, these proceed at roughly the same speed. In other words, varying c can make modest changes within a class, but changing to a different class is a quantum leap.

Given two algorithms for the same task, we should usually choose the algorithm of the lesser order of magnitude because for large enough n, it always "wins out." It is for large values of n that we need to be concerned about the time resources being used and, as we noted earlier, it is often for large values of n that we are seeking a computerized solution in the first place.

Note, however, that for smaller values of n, the size of the constant factor is significant. In Figure 3.12, the $10n$ line stayed above the $0.25n^2$ curve up to the crossover point of $n = 40$ because it had a large constant factor relative to the factor for n^2. Varying the factors changes the crossover point. If $10n$ and $0.25n^2$ represented the work of two different algorithms for the same task, and if we are sure that the size of the input is never going to

The Tortoise and the Hare

One way to compare performance among different makes of computers is to give the number of arithmetic operations, such as additions or subtractions of real numbers, that each one can do in 1 second. These operations are called *floating-point operations*, and computers are often compared in terms of the number of **flops** (floating-point operations per second) they can crank out.

In April 2014, AMD (Advanced Micro Devices) announced a new graphics card processor on a single circuit board with a speed of 11.6 *teraflops* (11.6 trillion floating-point operations per second). Such a card would be useful for a PC targeted toward video-game playing, although such a machine could perform general-purpose computing as well. In June 2014, the Tianhe-2 ("Milky Way-2") at Sun Yat-sen University, Guangzhou, China, was declared the world's top-speed supercomputer. It is a parallel processor system with 3,120,000 core processors. It performed at the rate of nearly 34 *petaflops* (34 quadrillion floating-point operations per second) during testing, and theoretically will be able to attain a rate of nearly 55 petaflops. The supercomputer (at its testing rate) is almost 3,000 times faster than the gaming computer. In some sense this is a meaningless comparison because the two machines are optimized for entirely different purposes. Nonetheless, in terms of raw speed, the stage is set for the race between the tortoise and the hare.

Not fair, you say? We'll see. Let's suppose the gaming machine is assigned to run an $\Theta(n)$ algorithm, whereas the supercomputer gets an $\Theta(n^2)$ algorithm for the same task. The work units are floating-point operations, and for simplicity, we'll take the constant factor to be 1 in each case. Here are the timing results:

n	Desktop	Supercomputer
1,000	0.0000000000862 sec	0.0000000000294 sec
100,000,000	0.00000862 sec	0.294 sec
10,000,000,000,000	0.862 sec	2,941,176,471 sec = 93 years

Out of the gate—that is, for relatively small values of n such as 1,000—the supercomputer has the advantage and takes slightly less time. When n reaches 100,000,000, however, the supercomputer is falling behind. And for the largest value of n, the desktop leaves the supercomputer in the dust. The difference in order of magnitude between the algorithms was enough to slow down the mighty supercomputer and let the desktop pull ahead, chugging along doing its more efficient $\Theta(n)$ algorithm. Where would one need to perform 10,000,000,000,000 operations? Complex problems involving weather simulations, biomedical research, and economic modeling might utilize such number-crunching applications.

(Continued)

The point of this little tale is not to say that supercomputers will be readily replaced by desktop gaming computers! It is to demonstrate that the order of magnitude of the algorithm being executed can play a more important role than the raw speed of the computer.

exceed 40, then the $0.25n^2$ algorithm is preferable in terms of time resources used. (To continue the transportation analogy, for traveling short distances—say, to the end of the driveway—walking is faster than driving because of the overhead of getting the car started, and so on. But for longer distances, driving is faster.)

However, making assumptions about the size of the input on which an algorithm will run can be dangerous. A program that runs quickly on small input size may at some point be selected, perhaps because it seems efficient, to solve instances of the problem with large input size, at which point the efficiency may go down the drain! (Customer management software that served Amazon when it was a startup company with 1,000 registered customers may not translate satisfactorily to managing 244 million customers.) Part of the job of program documentation is to make clear any assumptions or restrictions about the input size the program was designed to handle.

Comparing algorithm efficiency only makes sense if there is a choice of algorithms for the task at hand. Are there any tasks for which a choice of algorithms exists? Yes; because sorting a list is such a common task, a lot of research has gone into finding good sorting algorithms. Selection sort is one sorting algorithm, but there are many others, including the bubble sort, described in Exercises 11–14 at the end of this chapter. You might wonder

Laboratory Experience 4

This Laboratory Experience allows you to step through animations of various sorting algorithms to understand how they work. The sorting algorithms available in the laboratory software include selection sort and bubble sort—which are described in this text—as well as insertion sort and quicksort, which are described in the laboratory manual. You'll be able to see values being switched around according to the various algorithms and see how lists eventually settle into sorted order.

You'll also do some experiments to measure the amount of work the various algorithms perform.

Practice Problem

An algorithm does $14n^2 + 5n + 1$ units of work on input of size n. Explain why this is considered an $\Theta(n^2)$ algorithm even though there is a term that involves just n.

why people don't simply use the one "best" sorting algorithm. It's not that simple. Some algorithms (unlike the selection sort) are sensitive to what the original input looks like. One algorithm might work well if the input is already close to being sorted, whereas another algorithm might work better if the input is random. An algorithm such as selection sort has the advantage of being relatively easy to understand. If the size of the list, n, is fairly small, then an easy-to-understand algorithm might be preferable to one that is more efficient but more obscure.

3.4 Analysis of Algorithms

3.4.1 Data Cleanup Algorithms

In this section, we'll look at three different algorithms that solve the same problem—the *data cleanup problem*—and then do an analysis of each. Suppose a survey includes a question about the age of the person filling out the survey, and that some people choose not to answer this question. When data from the survey are entered into the computer, an entry of 0 is used to denote "no response" because a legitimate value for age would have to be a positive number. For example, assume that the age data from 10 people who completed the survey are stored in the computer as the following 10-entry list, where the positions in the list range from 1 (far left) to 10 (far right).

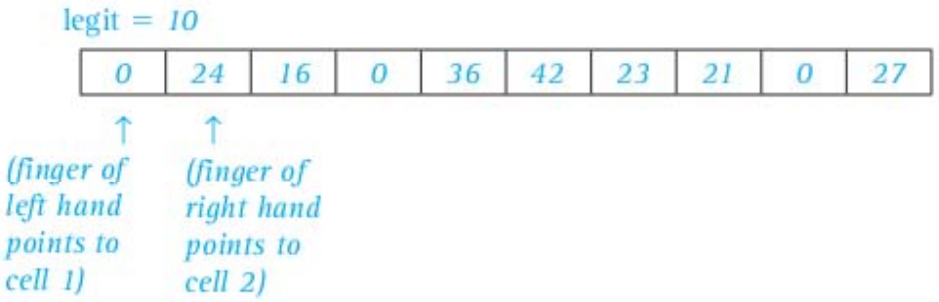
0	24	16	0	36	42	23	21	0	27
1	2	3	4	5	6	7	8	9	10

Eventually, the average age of the survey respondents is to be computed. Because the 0 values are not legitimate data—including them in the average would produce too low a value—we want to perform a "data cleanup" and remove them from the list before the average is computed. In our example, the cleaned data could consist of a 10-element list, where the seven legitimate elements are the first seven entries of the list, and some quantity—let's

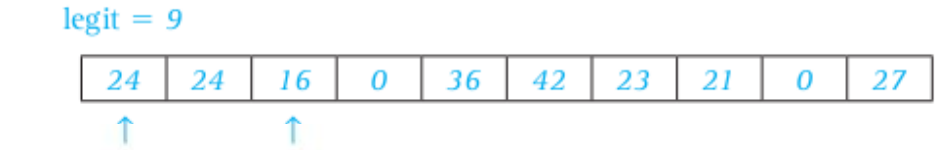
call it *legit*—has the value 7 to indicate that only the first seven entries are legitimate. An alternative acceptable result would be a seven-element list consisting of the seven legitimate data items, in which case there is no need for a *legit* quantity.

The Shuffle-Left Algorithm. Algorithm 1 to solve the data cleanup problem works in the way we might solve this problem using a pencil and paper (and an eraser) to modify the list. We proceed through the list from left to right, pointing with a finger on the left hand to keep our place, and passing over nonzero values. Every time we encounter a 0 value, we squeeze it out of the list by copying each remaining data item in the list one cell to the left. We could use a finger on the right hand to move along the list and point at what to copy next. The value of *legit*, originally set to the length of the list, is reduced by 1 every time a 0 is encountered. (Sounds complicated, but you'll see that it is easy.)

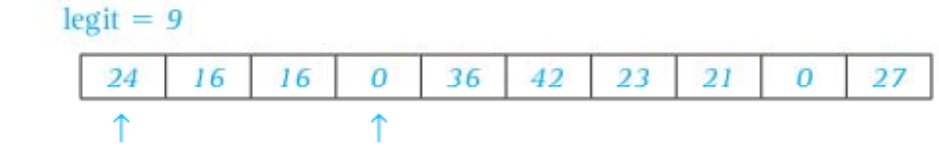
The original configuration is



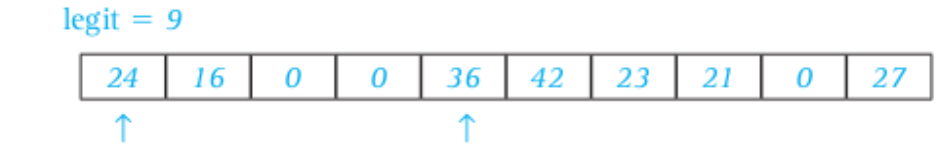
Because the first cell on the left contains a 0, the value of *legit* is reduced by 1, and all of the items to the right of the 0 must be copied one cell left. After the first such copy (of the 24), the scenario looks like



After the second copy (of the 16), we get



And after the third copy (of the 0), we get



Proceeding in this fashion, we find that after we copy the last item (the 27), the result is

Because the right-hand finger has moved past the end of the list, one entire shuffle-left process has been completed. It required copying nine items. We reset the right-hand finger to start again.

We must again examine position 1 for a 0 value because if the original list contained 0 in position 2, it would have been copied into position 1. If the value is not 0, as is the case here, both the left-hand finger and the right-hand finger move forward.

Moving along, we pass over the 16.

Another cycle of seven copies takes place to squeeze out the 0; the result is

The 36, 42, 23, and 21 are passed over, which results in

and then copying three items to squeeze out the final 0 gives

The left-hand finger is pointing at a nonzero element, so another advance of both fingers gives

At this point, we can stop because the left-hand finger is past the number of legitimate data items (*legit* = 7). In total, this algorithm (on this list) examined all 10 data items, to see which ones were 0, and copied 9 + 7 + 3 = 19 items.

A pseudocode version of the shuffle-left algorithm to act on a list of *n* items appears in Figure 3.14. The quantities *left* and *right* correspond to the positions where the left-hand and right-hand fingers point, respectively. You should trace through this algorithm for the preceding example to see that it does what we described.

To analyze the time efficiency of an algorithm, you begin by identifying the fundamental units of work the algorithm performs. For the data cleanup

FIGURE 3.14

```
1.  Get values for n and the n data items
2.  Set the value of legit to n
3.  Set the value of left to 1
4.  Set the value of right to 2
5.  While left is less than or equal to legit do Steps 6 through 14
6.      If the item at position left is not 0 then do Steps 7 and 8
7.          Increase left by 1
8.          Increase right by 1
9.      Else (the item at position left is 0) do Steps 10 through 14
10.         Reduce legit by 1
11.         While right is less than or equal to n do Steps 12 and 13
12.             Copy the item at position right into position (right − 1)
13.             Increase right by 1
14.         Set the value of right to (left + 1)
15. Stop
```

The shuffle-left algorithm for data cleanup

problem, any algorithm must examine each of the n elements in the list to see whether they are 0. This gives a base of at least $\Theta(n)$ work units.

The other unit of work in the shuffle-left algorithm is copying numbers. The best case occurs when the list has no 0 values because no copying is required. The worst case occurs when the list has all 0 values. Because the first element is 0, the remaining $n - 1$ elements are copied one cell left and *legit* is reduced from n to $n - 1$. After the 0 in position 2 gets copied into position 1, the first element is again 0, which again requires $n - 1$ copies and reduces *legit* from $n - 1$ to $n - 2$. This repeats until *legit* is reduced to 0, a total of n times. Thus there are n passes, during each of which $n - 1$ copies are done. The algorithm does

$$n(n-1) = n^2 - n$$

copies. If we were to draw a graph of $n^2 - n$, we would see that for large n, the curve follows the shape of n^2. The second term can be disregarded because as n increases, the n^2 term grows much larger than the n term; the n^2 term dominates and determines the shape of the curve. The shuffle-left algorithm is thus an $\Theta(n^2)$ algorithm in the worst case.

The shuffle-left algorithm is space efficient because it only requires four memory locations to store the quantities *n*, *legit*, *left*, and *right* in addition to the memory required to store the list itself.

The Copy-Over Algorithm. The second algorithm for solving the data cleanup problem also works as we might if we decided to write a new list using a pencil and paper. It scans the list from left to right, copying every legitimate (nonzero) value into a new list that it creates. After this algorithm is finished, the original list still exists, but so does a new list that contains only nonzero values.

For our example, the result would be

0	24	16	0	36	42	23	21	0	27

24	16	36	42	23	21	27

Every list entry is examined to see whether it is 0 (as in the shuffle-left algorithm), and every nonzero list entry is copied once (into the new list), so altogether seven copies are done for this example. This is fewer copies than the shuffle-left algorithm requires, but a lot of extra memory space is required because an almost complete second copy of the list is stored. Figure 3.15 shows the pseudocode for this copy-over algorithm.

The best case for this algorithm occurs if all elements are 0; no copies are done so the work is just the $\Theta(n)$ work to examine each list element and see that it is 0. No extra space is used. The worst case occurs if there are no 0 values in the list. The algorithm copies all n nonzero elements into the new list and doubles the space required. Combining the two types of work units, we find that the copy-over algorithm is only $\Theta(n)$ in time efficiency even in the worst case because $\Theta(n)$ examinations and $\Theta(n)$ copies still equal $\Theta(n)$ steps.

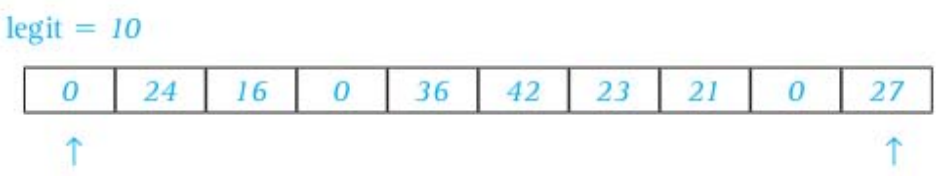
FIGURE 3.15

1. Get values for *n* and the *n* data items
2. Set the value of *left* to 1
3. Set the value of *newposition* to 1
4. While *left* is less than or equal to *n* do Steps 5 through 8
5. If the item at position *left* is not 0 then do Steps 6 and 7
6. Copy the item at position *left* into position *newposition* in new list
7. Increase *newposition* by 1
8. Increase *left* by 1
9. Stop

The copy-over algorithm for data cleanup

Comparing the shuffle-left algorithm and the copy-over algorithm, we see that no 0 elements is the best case of the first algorithm and the worst case of the second, whereas all 0 elements is the worst case of the first and the best case of the second. The second algorithm is more time efficient and less space efficient. This choice is called the *time/space tradeoff*—you gain something by giving up something else. Seldom is it possible to improve both dimensions at once, but our next algorithm accomplishes just that.

The Converging-Pointers Algorithm. For the third algorithm, imagine that we move one finger along the list from left to right and another finger from right to left. The left finger slides to the right over nonzero values. Whenever the left finger encounters a 0 item, we reduce the value of *legit* by 1, copy whatever item is at the right finger into the left-finger position, and slide the right finger one cell left. Initially in our example



And because a 0 is encountered at position *left*, the item at position *right* is copied into its place, and both *legit* and *right* are reduced by 1. This results in

The value of *left* increases until the next 0 is reached.

legit = 9

27	24	16	0	36	42	23	21	0	27
			↑					↑	

Again, the item at position *right* is copied into position *left*, and *legit* and *right* are reduced by 1.

legit = 8

27	24	16	0	36	42	23	21	0	27
			↑				↑		

The item at position left is still 0, so another copy takes place.

legit = 7

27	24	16	21	36	42	23	21	0	27
			↑			↑			

From this point, the left finger advances until it meets the right finger, which is pointing to a nonzero element, and the algorithm stops. Once again, each element is examined to see whether it equals 0. For this example, only three copies are needed—fewer even than for algorithm 2, but this algorithm requires no more memory space than algorithm 1. The pseudocode version of the converging-pointers algorithm is given in Figure 3.16.

FIGURE 3.16

1. Get values for *n* and the *n* data items
2. Set the value of *legit* to *n*
3. Set the value of *left* to 1
4. Set the value of *right* to *n*
5. While *left* is less than *right* do Steps 6 through 10
6. If the item at position *left* is not 0 then increase *left* by 1
7. Else (the item at position *left* is 0) do Steps 8 through 10
8. Reduce *legit* by 1
9. Copy the item at position *right* into position *left*
10. Reduce *right* by 1
11. If the item at position *left* is 0, then reduce *legit* by 1
12. Stop

The converging-pointers algorithm for data cleanup

The best case for this algorithm, as for the shuffle-left algorithm, is a list containing no 0 elements. The worst case, as for the shuffle-left algorithm, is a list of all 0 entries. With such a list, the converging-pointers algorithm repeatedly copies the element at position *right* into the first position, each time reducing the value of *right*. *Right* goes from n to 1, with one copy done at each step, resulting in $n - 1$ copies. This algorithm is $\Theta(n)$ in the worst case. Like the shuffle-left algorithm, it is space efficient. It is possible in this case to beat the time/space trade-off, in part because the data cleanup problem requires no particular ordering of the nonzero elements in the "clean" list; the converging-pointers algorithm moves these elements out of their original order.

It is hard to define what an "average" case is for any of these algorithms; the amount of work done depends on how many 0 values there are in the list and perhaps on where in the list they occur. If we assume, however, that the number of 0 values is some percentage of n and that these values are scattered throughout the list, then it can be shown that the shuffle-left algorithm will still do $\Theta(n^2)$ work, whereas the converging-pointers algorithm will do $\Theta(n)$. Figure 3.17 summarizes our analysis, although it doesn't reflect the three or four extra memory cells needed to store other quantities used in the algorithms, such as *legit*, *left*, and *right*.

Let's emphasize again the difference between an algorithm that is $\Theta(n)$ in the amount of work it does and one that is $\Theta(n^2)$. In an $\Theta(n)$ algorithm, the work is proportional to n. Hence if you double n, you double the amount of work; if you multiply n by 10, you multiply the work by 10. But in an $\Theta(n^2)$ algorithm, the work is proportional to the *square* of n. Hence if you double n, you multiply the amount of work by 4; if you multiply n by 10, you multiply the work by 100.

This is probably a good place to explain why the distinction between n and $2n$ is important when we are talking about space, but we simply classify n and $8000n$ as $\Theta(n)$ when we are talking about units of work. Units of work translate into time when the algorithm is executed, and time is a much more elastic resource than space. Whereas we want an algorithm to run in the shortest possible time, in many cases there is no fixed limit to the amount of

FIGURE 3.17

	1. Shuffle-left		2. Copy-over		3. Converging-pointers	
	Time	Space	Time	Space	Time	Space
Best case	$\Theta(n)$	n	$\Theta(n)$	n	$\Theta(n)$	n
Worst case	$\Theta(n^2)$	n	$\Theta(n)$	$2n$	$\Theta(n)$	n
Average case	$\Theta(n^2)$	n	$\Theta(n)$	$n \le x \le 2n$	$\Theta(n)$	n

Analysis of three data cleanup algorithms

time that can be expended. There is, however, always a fixed upper bound on the amount of memory that the computer has available to use while executing an algorithm, so we track space consumption more closely.

3.4.2 Binary Search

The sequential search algorithm searches a list of n items for a particular item; it is an $\Theta(n)$ algorithm. Another algorithm, the **binary search algorithm**, is more efficient but it works only when the search list is already sorted.

To understand how binary search operates, let us go back to the problem of searching for *NUMBER* in a reverse telephone directory, but now we assume that the directory is sorted in increasing numerical order by phone number. As we noted in Chapter 2, you would not search for (555) 123-4567 in such a directory by starting with the first number and proceeding sequentially through the list. Instead you would look for this number near the middle of the list, and if you didn't find it immediately, you would continue your search on the front half or the back half of the list.

This is exactly how the binary search algorithm works on a sorted list. It first looks for *NUMBER* at roughly the halfway point in the list. If the number there equals *NUMBER*, the search is over. If *NUMBER* comes numerically before the number at the halfway point, then the search is narrowed to the front half of the list, and the process begins again on this smaller list. If *NUMBER* comes numerically after the number at the halfway point, then the search is narrowed to the back half of the list, and the process begins again on *this* smaller list. The algorithm halts when *NUMBER* is found or when the sublist becomes empty.

Practice Problems

In the data cleanup problem, suppose the original data list is

2	0	4	1

1. Write the data list after completion of algorithm 1, the shuffle-left algorithm.
2. Write the two data lists after completion of algorithm 2, the copy-over algorithm.
3. Write the data list after completion of algorithm 3, the converging-pointers algorithm.
4. Make up a data list such that Step 11 of the converging-pointers algorithm (Figure 3.16) is needed.

FIGURE 3.18

1. Get values for *NUMBER*, n, $T_1, \ldots, T_n$ and $N_1, \ldots, N_n$
2. Set the value of *beginning* to 1 and set the value of *Found* to NO
3. Set the value of *end* to n
4. While *Found* = NO and *beginning* is less than or equal to *end* do Steps 5 through 10
5. Set the value of m to the middle value between *beginning* and *end*
6. If *NUMBER* = T_m, the number found at the midpoint between *beginning* and *end*, then do Steps 7 and 8
7. Print the name of the corresponding person, N_m
8. Set the value of *Found* to YES
9. Else if *NUMBER* < T_m, then set *end* = $m - 1$
10. Else (*NUMBER* > T_m) set *beginning* = $m + 1$
11. If (*Found* = NO) then print the message 'Sorry, this number is not in our directory'
12. Stop

Binary search algorithm (list must be sorted)

Figure 3.18 gives a pseudocode version of the binary search algorithm on a sorted n-element list. Here *beginning* and *end* mark the beginning and end of the section of the list under consideration. Initially the whole list is considered, so at first *beginning* is 1 and *end* is n. If *NUMBER* is not found at the midpoint m of the current section of the list, then setting *end* equal to one less than the midpoint (Step 9) means that at the next pass through the loop, the front half of the current section is searched. Setting *beginning* equal to one more than the midpoint (Step 10) means that at the next pass through the loop, the back half of the current section is searched. Thus, as the algorithm proceeds, the *beginning* marker can move toward the back of the list, and the *end* marker can move toward the front of the list. If the *beginning* marker and the *end* marker cross over—that is, *end* becomes less than *beginning*—then the current section of the list is empty and the search terminates. Of course it also terminates if *NUMBER* is found.

Let's do an example, using seven telephone numbers sorted in increasing order. The following list shows not only the numbers in the list but also their locations in the list. For clarity, we have removed the punctuation from the numbers and written them simply as 10-digit numerals.

2478346543	3563278900	3597211488	5656170224	6485551285	7719215281	8796562127
1	2	3	4	5	6	7

Suppose we search this list for the number 3597211488. We set *beginning* to 1 and *end* to 7; the midpoint between 1 and 7 is 4. We compare 3597211488 with the position 4 number, 5656170224. The number 3597211488 is less than 5656170224, so the algorithm sets *end* to $4 - 1 = 3$ (Step 9) to continue the search on the front half of the list,

2478346543	3563278900	3597211488
1	2	3

The midpoint between *beginning* = 1 and *end* = 3 is 2, so we compare 3597211488 with the position 2 number, 3563278900. The number 3597211488 is greater than 3563278900, so the algorithm sets *beginning* to 2 + 1 = 3 (Step 10) in order to continue the search on the back half of this list, namely

3597211488
3

At the next pass through the loop, the midpoint between *beginning* = 3 and *end* = 3 is 3, so we compare 3597211488 with the position 3 number, 3597211488. We have found the target number; the corresponding name can be printed and *Found* changed to YES. The loop terminates, and then the algorithm terminates.

Now suppose we search this same list for the number 7213350096. As before, the first midpoint is 4, so 7213350096 is compared with 5656170224. Because 7213350096 > 5656170224, the search continues with *beginning* = 5, *end* = 7 on the back half:

6485551285	7719215281	8796562127
5	6	7

The midpoint is 6, so 7213350096 is compared with 7719215281. Because 7213350096 < 7719215281, the search continues with *beginning* = 5, *end* = 5 on the front half:

6485551285
5

The midpoint is 5, so 7213350096 is compared with 6485551285. Because 7213350096 > 6485551285, *beginning* is set to 6 to continue the search on the "back half" of this list. The algorithm checks the condition at Step 4 to see whether to repeat the loop again and finds that *end* is less than *beginning* (*end* = 5, *beginning* = 6). The loop is abandoned, and the algorithm moves on to Step 11 and indicates that our target number is not in the list.

It is easier to see how the binary search algorithm operates if we list the locations of the numbers checked in a treelike structure. The tree in Figure 3.19 shows the possible search locations in a seven-element list. The search starts at the top of the tree, at location 4, the middle of the original list. If the number at location 4 is *NUMBER*, the search halts. If *NUMBER* comes after the number at location 4, the right branch is taken and the next location searched is location 6. If *NUMBER* comes before the number at location 4, the left branch is taken and the next location searched is location 2. If *NUMBER* is not found at location 2, the next location searched is either

1 or 3. Similarly, if *NUMBER* is not found at location 6, the next location searched is either 5 or 7.

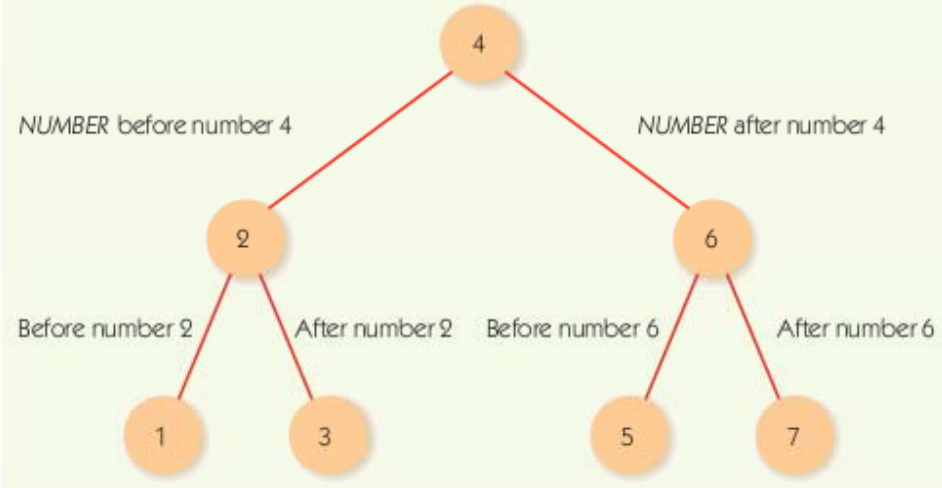
In Figure 3.18, the binary search algorithm, we assume in Step 5 that there is a middle position between *beginning* and *end*. This happens only when there is an odd number of elements in the list. Let us agree to define the "middle" of an even number of entries as the end of the first half of the list. With eight elements, for example, the midpoint position is location 4.

1 2 3 4 5 6 7 8

With this understanding, the binary search algorithm can be used on sorted lists of any size. And of course the list items need not be numbers; they could also be text items that are sorted alphabetically.

Like the sequential search algorithm, the binary search algorithm relies on comparisons, so to analyze the algorithm, we count the number of comparisons as an indication of the work done. The best case, as in sequential search, requires only one comparison—the target is located on the first try. The worst case, as in sequential search, occurs when the target is not in the list. However, we learn this much sooner in binary search than in sequential search. In our list of seven telephone numbers, only three comparisons are needed to determine that 7213350096 is not in the list. The number of comparisons needed is the number of circles in some branch from the top to the bottom of the tree in Figure 3.19. These circles represent searches at the midpoints of the whole list, half the list, one quarter of the list, and so on. This process continues as long as the sublists can be cut in half.

FIGURE 3.19



Binary search tree for a seven-element list

Let's do a minor mathematical digression here. The number of times a number n can be cut in half and not go below 1 is called the *logarithm of n to the base 2*, which is abbreviated lg n (also written in some texts as $\log_2 n$). For example, if n is 16, then we can do four such divisions by 2:

$16 / 2 = 8$
$8 / 2 = 4$
$4 / 2 = 2$
$2 / 2 = 1$

so lg 16 = 4. This is another way of saying that $2^4 = 16$. In general,

lg $n = m$ is equivalent to $2^m = n$

Figure 3.20 shows a few values of n and lg n. From these, we can see that as n doubles, lg n increases by only 1, so lg n grows much more slowly than n. Figure 3.21 shows the two basic shapes of n and lg n and again conveys that lg n grows much more slowly than n.

Remember the analogy we suggested earlier about the difference in time consumed between $\Theta(n^2)$ algorithms, equivalent to various modes of walking, and $\Theta(n)$ algorithms, equivalent to various modes of driving? We carry that analogy further by saying that algorithms of order of magnitude lg n, $\Theta(\lg n)$, are like various modes of flying. Changing the coefficients of lg n can mean that we go from a Piper Cub to an F-22 Raptor but flying, in any form, is still a fundamentally different—and much faster—mode of travel than walking or driving.

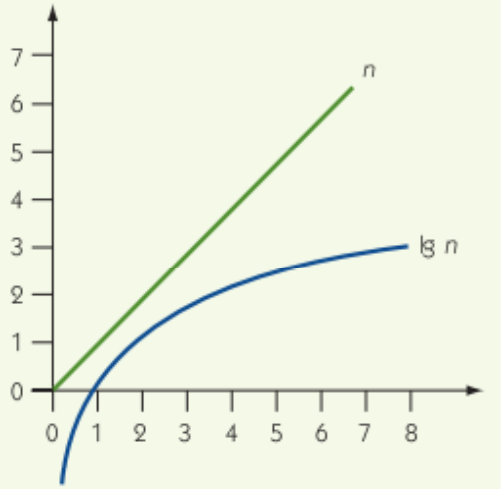
Suppose we are doing a binary search on n items. In the worst case, as we have seen, the number of comparisons is related to the number of times the list of length n can be halved. Binary search does $\Theta(\lg n)$ comparisons in the worst case (see Exercise 31 at the end of the chapter for an exact formula

FIGURE 3.20

n	lg n
8	3
16	4
32	5
64	6
128	7

Values for n and lg n

FIGURE 3.21



A comparison of n and lg n

for the worst case). As a matter of fact, it also does $\Theta(\lg n)$ comparisons in the average case (although the exact value is a smaller number than in the worst case). This is because most of the items in the list occur at or near the bottom of the tree, where the maximum amount of work must be done. As Figure 3.19 shows, relatively few locations, where the target might be found and the algorithm terminate sooner, are higher in the tree.

The two search algorithms, binary search and sequential search, differ in the order of magnitude of the work they do. Binary search is an $\Theta(\lg n)$ algorithm, whereas sequential search is an $\Theta(n)$ algorithm, in both the worst case and the average case. To compare the binary search algorithm with the sequential search algorithm, suppose there are 100 elements in the list. In the worst case, sequential search requires 100 comparisons, and binary search requires 7 ($2^7 = 128$). In the average case, sequential search requires about 50 comparisons, and binary search 6 or 7 (still much less work). The improvement in binary search becomes even more apparent as the search list gets longer. For example, if $n = 100{,}000$, then in the worst case, sequential search requires 100,000 comparisons, whereas binary search requires 17 (because $2^{16} = 65{,}536$ and $2^{17} = 131{,}072$). If we wrote two programs, one using sequential search and one using binary search, and ran them on a computer that can do 1,000 comparisons per second, then to determine that an item is not in the list (the worst case) the sequential search program would use

$$100{,}000 \text{ comparisons} \times \frac{1}{1{,}000} \text{seconds/comparision} = 100 \text{ seconds}$$

or 1.67 minutes, just to do the necessary comparisons, disregarding the constant factor for advancing the index. The binary search program would use

$$17 \text{ comparisons} \times \frac{1}{1{,}000} \text{seconds/comparison} = 0.017 \text{ seconds}$$

to do the comparisons, disregarding a constant factor for updating the values of *beginning* and *end*. This is quite a difference.

Suppose our two programs are used with the 350,000,000 numbers we assume are in the reverse telephone directory. On the average, the sequential search program needs about

$$\frac{350{,}000{,}000}{2} \text{comparisons} \times \frac{1}{1{,}000} \text{seconds/comparison} = 175{,}000 \text{ seconds}$$

(over 2 days!) just to do the comparisons to find a number in the list, whereas the binary search program needs (because $2^{28} = 268{,}435{,}456$ and $2^{29} = 536{,}870{,}912$) about

$$29 \text{ comparisons} \times \frac{1}{1{,}000} \text{seconds/comparison} = 0.029 \text{ seconds}$$

This is an even more impressive difference. Furthermore, it's a difference due to the inherent inefficiency of an $\Theta(n)$ algorithm compared with an $\Theta(\lg n)$ algorithm; the difference can be mitigated but not eliminated by using a

faster computer. If our computer does 50,000 comparisons per second, then the average times become about

$$\frac{350,000,000}{2}\text{comparisons} \times \frac{1}{50,000}\text{seconds/comparison} = 3,500\text{seconds}$$

or nearly an hour for sequential search and about

$$29\text{comparisons} \times \frac{1}{50,000}\text{seconds/comparison} = 0.00058\text{seconds}$$

for binary search. The sequential search alternative is simply not acceptable. That is why analyzing algorithms and choosing the best one can be so important. We also see, as we noted in Chapter 2, that the way the problem data are organized can greatly affect the best choice of algorithm.

The binary search algorithm works only on a list that has already been sorted. An unsorted list could be sorted before using a binary search, but sorting also takes a lot of work, as we have seen. If a list is to be searched only a few times for a few particular items, then it is more efficient to do sequential search on the unsorted list (a few $\Theta(n)$ tasks). But if the list is to be searched repeatedly, it is more efficient to sort it and then use binary search: one $\Theta(n^2)$ task and many $\Theta(\lg n)$ tasks, as opposed to many $\Theta(n)$ tasks.

As to space efficiency, binary search, like sequential search, requires only a small amount of additional storage to keep track of beginning, end, and midpoint positions in the list. Thus, it is space efficient; in this case, we did not have to sacrifice space efficiency to gain time efficiency. But we did have to sacrifice generality—binary search works only on a sorted list whereas sequential search works on any list.

Practice Problems

1. Suppose that, using the list of seven numbers from this section, we try binary search to decide whether 6485551285 is in the list. What numbers would be compared with 6485551285?

2. Suppose that, using the list of seven numbers from this section, we try binary search to decide whether 9342426855 is in the list. What numbers would be compared with 9342426855?

3. In the worst case, how many comparisons will be required to find a single Social Security number (SSN) from among the approximately 453,700,000 numbers issued since Social Security began if the data file is sorted numerically by SSN? How many will be required in the worst case if the data file is sorted alphabetically by the individual's last name?

Laboratory Experience 5

In this Laboratory Experience, you will be able to run animations of the shuffle-left algorithm and the converging-pointers algorithm for the data cleanup problem. You'll be able to see the left and right pointers take on different values, which represent changing positions in the data list. As the algorithms run on various lists, you can count the number of copies of data elements that are required and see how they relate to the original positions of any 0 items in the list.

You will also work with an animation of the binary search algorithm and see how the work done compares with the theoretical results we discovered in this section.

3.4.3 Pattern Matching

The pattern-matching algorithm in Chapter 2 involves finding all occurrences of a pattern of the form $P_1P_2 \ldots P_m$ within text of the form $T_1T_2 \ldots T_n$. Recall that the algorithm simply does a "forward march" through the text, at each position attempting to match each pattern character against the text characters. The process stops only after text position $n - m + 1$, when the remaining text is not as long as the pattern so that there could not possibly be a match. This algorithm is interesting to analyze because it involves two measures of input size: n, the length of the text string, and m, the length of the pattern string. The unit of work is comparison of a pattern character with a text character.

Surprisingly, both the best case and the worst case of this algorithm can occur when the pattern is not in the text at all. The difference hinges on exactly *how* the pattern fails to be in the text. The best case occurs if the first character of the pattern is nowhere in the text, as in

Text: *KLMNPQRSTX*
Pattern: *ABC*

In this case, $n - m + 1$ comparisons are required, trying (unsuccessfully) to match P_1 with $T_1, T_2, \ldots, T_{n-m+1}$ in turn. Each comparison fails, and the algorithm slides the pattern forward to try again at the next position in the text.

The maximum amount of work is done if the pattern *almost* occurs everywhere in the text. Consider, for example, the following case:

Text: *AAAAAAAAA*
Pattern: *AAAB*

Starting with T_1, the first text character, the match with the first pattern character is successful. The match with the second text character and the second pattern character is also successful. Indeed $m - 1$ characters of the pattern match with the text before the mth comparison proves a failure. The process starts over from the second text character, T_2. Once again, m comparisons are required to find a mismatch. Altogether, m comparisons are required for each of the $n - m + 1$ starting positions in the text.

Another version of the worst case occurs when the pattern is found at each location in the text, as in

Text: AAAAAAAAA
Pattern: AAAA

This results in the same comparisons as are done for the other worst case, the only difference being that the comparison of the last pattern character is successful.

Unlike our simple examples, pattern matching usually involves a pattern length that is short compared with the text length, that is, when m is much less than n. In such cases, $n - m + 1$ is essentially n. The pattern-matching algorithm is therefore $\Theta(n)$ in the best case and $\Theta(m \times n)$ in the worst case.

It requires somewhat improbable situations to create the worst cases we have described. In general, the forward-march algorithm performs quite well on text and patterns consisting of ordinary words. Other pattern-matching algorithms are conceptually more complex but require less work in the worst case.

3.4.4 Summary

Figure 3.22 shows an order-of-magnitude summary of the time efficiency for the algorithms we have analyzed.

FIGURE 3.22

Problem	Unit of Work	Algorithm	Best Case	Worst Case	Average Case
Searching	Comparisons	Sequential search	1	$\Theta(n)$	$\Theta(n)$
		Binary search	1	$\Theta(\lg n)$	$\Theta(\lg n)$
Sorting	Comparisons and exchanges	Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Data cleanup	Examinations and copies	Shuffle-left	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
		Copy-over	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
		Converging-pointers	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Pattern matching	Character comparisons	Forward march	$\Theta(n)$	$\Theta(m \times n)$	

Order-of-magnitude time efficiency summary

Practice Problem

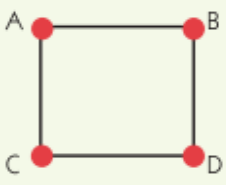
Use the first sample pattern and text given in Section 3.4.3 for the worst case of the pattern-matching algorithm. What is m? What is n? What is $m \times n$? This algorithm is $\Theta(m \times n)$ in the worst case, but what is the exact number of comparisons done?

3.5 When Things Get Out of Hand

We have so far found examples of algorithms that are $\Theta(\lg n)$, $\Theta(n)$, and $\Theta(n^2)$ in time efficiency. Order of magnitude determines how quickly the values grow as n increases. An algorithm of order lg n does less work as n increases than does an algorithm of order n, which in turn does less work than one of order n^2. The work done by any of these algorithms is no worse than a constant multiple of n^2, which is a polynomial in n. Therefore, these algorithms are **polynomially bounded** in the amount of work they do as n increases.

Some algorithms must do work that is not polynomially bounded. Consider four cities, A, B, C, and D, that are connected as shown in Figure 3.23, and ask the following question: Is it possible to start at city A, go through every other city exactly once, and end up back at A? Of course, we as humans can immediately see in this small problem that the answer is "yes" and that there are two such paths: A-B-D-C-A and A-C-D-B-A. However, an algorithm doesn't get to "see" the entire picture at once, as we can; it has available to it only isolated facts such as "A is connected to B and to C," "B is connected to A and to D," and so on. If the number of *nodes* and

FIGURE 3.23

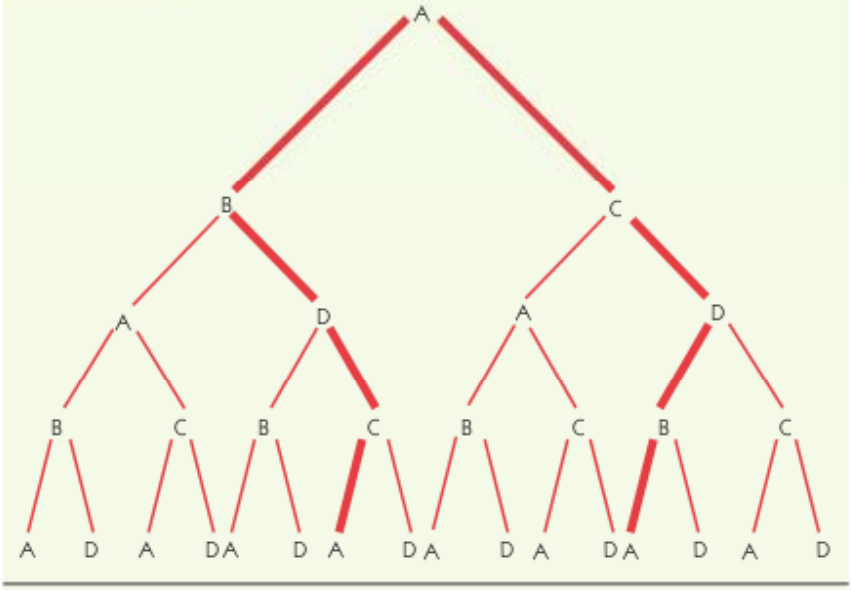


Four connected cities

connecting *edges* is large, humans also might not "see" the solution immediately. A collection of nodes and connecting edges is called a *graph*. A path through a graph that begins and ends at the same node and goes through all other nodes exactly once is called a *Hamiltonian circuit*, named for the Irish mathematician William Rowan Hamilton (1805–1865). If there are n nodes in the graph, then a Hamiltonian circuit, if it exists, must have exactly n links. In the case of the four cities, for instance, if the path must go through exactly A, B, C, D, and A (in some order), then there are five nodes on the path (counting A twice) and four links.

Our problem is to determine whether an arbitrary graph has a Hamiltonian circuit. An algorithm to solve this problem examines all possible paths through the graph that are the appropriate length to see whether any of them are Hamiltonian circuits. The algorithm can trace all paths by beginning at the starting node and choosing at each node where to go next. Without going into the details of such an algorithm, let's represent the possible paths with four links in the graph of Figure 3.23. Again, we use a tree structure. In Figure 3.24, A is the tree "root," and at each node in the tree, the nodes directly below it are the choices for the next node. Thus, any time B appears in the tree, it has the two nodes A and D below it because edges exist from B to A and from B to D. The "branches" of the tree are all the possible paths from A with four links. Once the tree has been built, an examination of the paths shows that only the two dark paths in the figure represent Hamiltonian circuits.

FIGURE 3.24



Hamiltonian circuits among all paths from A in Figure 3.23 with four links

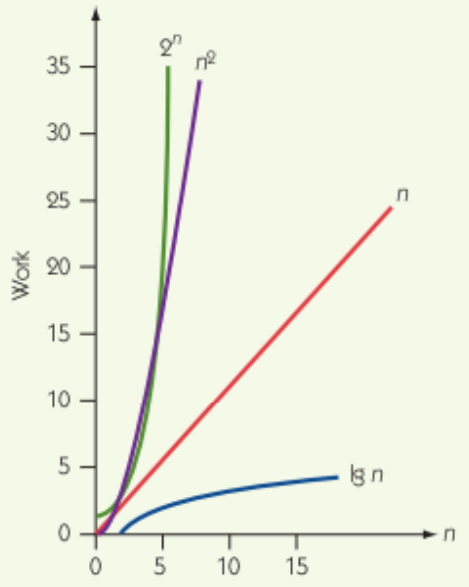
The number of paths that must be examined is the number of nodes at the bottom level of the tree. There is one node at the top of the tree; we'll call the top of the tree level 0. The number of nodes is multiplied by 2 for each level down in the tree. At level 1, there are 2 nodes; at level 2, there are $2^2 = 4$ nodes; at level 3, there are $2^3 = 8$ nodes; and at level 4, the bottom of the tree, there are $2^4 = 16$ nodes.

Suppose we are looking for a Hamiltonian circuit in a graph with n nodes and two choices at each node. The bottom of the corresponding tree is at level n, and there are 2^n paths to examine. If we take the examination of a single path as a unit of work, then this algorithm must do 2^n units of work. This is more work than any polynomial in n. An $\Theta(2^n)$ algorithm is called an **exponential algorithm**. Hence the trial-and-error approach to solving this Hamiltonian circuit problem is an exponential algorithm. (We could improve on this algorithm by letting it stop tracing a path whenever a repeated node different from the starting node is encountered, but it is still exponential. If there are more than two choices at a node, the amount of work is even greater.)

Figure 3.25 shows the four curves lg n, n, n^2, and 2^n. The rapid growth of 2^n is not really apparent here, however, because that curve is off the scale for values of n above 5. Figure 3.26 compares these four curves for values of n that are still small, but even so, 2^n is already far outdistancing the other values.

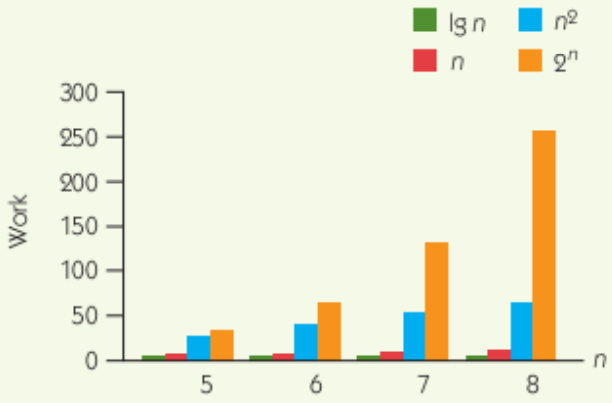
To appreciate fully why the order of magnitude of an algorithm is important, let's again imagine that we are running various algorithms as

FIGURE 3.25



Comparison of lg n, n, n^2, and 2^n

FIGURE 3.26



Comparisons of lg n, n, n^2, and 2^n for larger values of n

FIGURE 3.27

Order	n = 10	50	100	1,000
lg *n*	0.0003 sec	0.0006 sec	0.0007 sec	0.001 sec
n	0.001 sec	0.005 sec	0.01 sec	0.1 sec
n^2	0.01 sec	0.25 sec	1 sec	1.67 min
2^n	0.1024 sec	3,570 years	4×10^{16} centuries	*Too big to compute!!*

A comparison of four orders of magnitude

programs on a computer that can perform a single operation (unit of work) in 0.0001 second. Figure 3.27 shows the amount of time it takes for algorithms of $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n^2)$, and $\Theta(2^n)$ to complete their work for various values of n.

The expression 2^n grows unbelievably fast. An algorithm of $\Theta(2^n)$ can take so long to solve even a small problem that it is of no practical value. Even if we greatly increase the speed of the computer, the results are much the same. We now see more than ever why we added *efficiency* as a desirable feature for an algorithm and why future advances in computer technology won't change this. No matter how fast computers get, they will not be able to solve a problem of size $n = 100$ using an algorithm of $\Theta(2^n)$ in any reasonable period of time.

The algorithm we have described here for testing an arbitrary graph for Hamiltonian circuits is an example of a *brute force algorithm*—one that beats the problem into submission by trying all possibilities. In Chapter 1, we described a brute force algorithm for winning a chess game; it consisted of looking at all possible game scenarios from any given point on and then picking a winning one. This is also an exponential algorithm. Some very practical problems have exponential solution algorithms. For example, an email message that you send over the Internet is routed along the shortest possible path through intermediate computers from your mail server computer to the destination mail server computer. An exponential algorithm to solve this problem would examine all possible paths to the destination and then use the shortest one. As you can imagine, the Internet uses a better (more efficient) algorithm than this one!

Are there problems for which no polynomially bounded algorithm exists? Such problems are called intractable; they are solvable, but the solution algorithms all require so much work as to be virtually useless. The Hamiltonian circuit problem is suspected to be such a problem, but we don't really know for sure! No one has yet found a solution algorithm that works in polynomial time, but neither has anyone proved that such

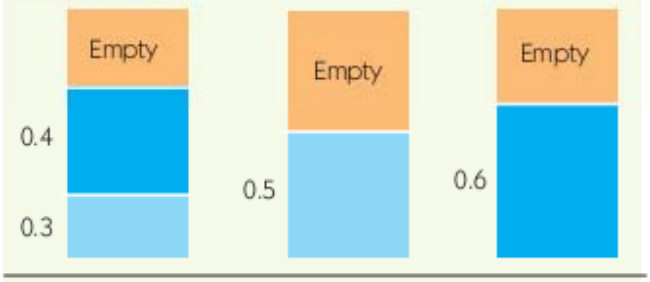
an algorithm does not exist. This is a problem of great interest in computer science. A surprising number of problems fall into this "suspected intractable" category. Here's another one, called the *bin-packing problem*: Given an unlimited number of bins of volume X units and given n objects, all of volume between 0 and X, find the minimum number of bins needed to store the n objects. A brute force algorithm would try all possibilities, which again is not a polynomial algorithm. Any manufacturer who ships sets of various items in standard-sized cartons or anyone who wants to store variable-length video clips on a set of DVDs in the most efficient way would be interested in a polynomial algorithm that solves this minimization problem.

Problems for which no known polynomial solution algorithm exists are sometimes approached via approximation algorithms. These algorithms don't give the exact answer to the problem, but they provide a close approximation to a solution. (See Exercise 2 of Chapter 1.) For example, an approximation algorithm to solve the bin-packing problem is to take the objects in order, put the first one into bin 1, and stuff each remaining object into the first bin that can hold it. This (reasonable) approach may not give the absolute minimum number of bins needed, but it gives a first cut at the answer. (Anyone who has watched passengers stowing carry-on baggage in an airplane has seen this approximation algorithm at work.)

For example, suppose a sequence of four objects with volumes of 0.3, 0.4, 0.5, and 0.6 are stored in bins of size 1.0 using the "first-fit" algorithm described previously. The result requires three bins, which would be packed as shown in Figure 3.28. However, this is not the optimal solution (see Exercise 39 at the end of the chapter).

In Chapter 12, we will learn that there are problems that cannot be solved algorithmically, even if we are willing to accept an extremely inefficient solution.

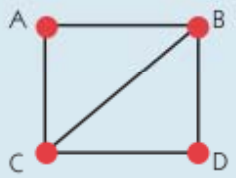
FIGURE 3.28



A first-fit solution to a bin-packing problem

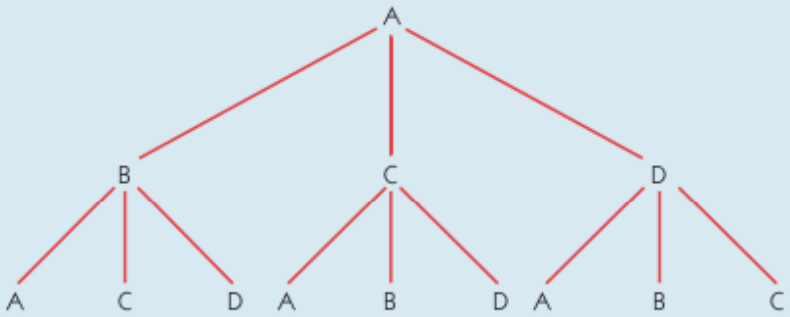
Practice Problems

1. Consider the following graph:



   Draw a tree similar to Figure 3.24 showing all paths from A and highlighting those that are Hamiltonian circuits (these are the same two circuits as before). How many paths must be examined?

2. The following tree shows all paths with two links that begin at node A in some graph. Draw the graph.



3. If an algorithm were determined to have an order of magnitude of n^n, do you think it would be classified as polynomial or exponential? Explain.

3.6 Summary of Level 1

We defined computer science as the study of algorithms, so it is appropriate that Level 1 was devoted to exploring algorithms in more detail. In Chapter 2, we discussed how to represent algorithms using pseudocode. Pseudocode provides us with a flexible language for expressing the building blocks from which algorithms can be constructed. These building blocks include assigning a particular value to a quantity, choosing one of two next steps on the basis of some condition, or repeating steps in a loop.

Laboratory Experience 6

The various sorting algorithms examined in Laboratory Experience 4 (selection sort, quicksort, etc.) do different amounts of work on the same data sets. But how do these various workloads affect the actual running time of the algorithms? In this Laboratory Experience, you can run these sorting algorithms and find their wall-clock running time on different sizes of input. In addition, because you can see the patterns of values falling into place in a large list while an algorithm runs, you will get a much better understanding of how each sorting algorithm moves values around to accomplish its task.

We developed algorithmic solutions to three very practical problems: searching a list of items for a particular target value, finding the largest number in a list of numbers, and searching for a particular pattern of characters within a segment of text. In Chapter 3, we noted that computer scientists develop algorithms to be *used* and thus there is a set of desirable properties for algorithms, including ease of understanding, elegance, and efficiency, in addition to correctness. Of these, efficiency—which may be either time efficiency or space efficiency—is the most easily quantifiable.

A convenient way to classify the time efficiency of algorithms is by examining the order of magnitude of the work they do. Algorithms that are of differing orders of magnitude do fundamentally different amounts of work. Regardless of the constant factor that reflects peripheral work or how fast the computer on which these algorithms execute, for problems with sufficiently large input, the algorithm of the lowest order of magnitude requires the least time.

We analyzed the time efficiency of the sequential search algorithm and discovered that it is an $\Theta(n)$ algorithm in both the worst case and the average case. We found a selection sort algorithm that is $\Theta(n^2)$, we found a binary search algorithm that is $\Theta(\lg n)$, and we analyzed the pattern-matching algorithm from Chapter 2. By examining the data cleanup problem, we learned that algorithms that solve the same task can indeed differ in the order of magnitude of the work they do, sometimes by employing a time/space trade-off. We also learned that there are algorithms that require more than polynomially bounded time to complete their work and that such algorithms may take so long to execute, regardless of the speed of the computer

on which they are run, that they provide no practical solution. Some important problems may be intractable–that is, have no polynomially bounded solution algorithms.

Some computer scientists work on trying to decide whether a particular problem is intractable. Some work on finding more efficient algorithms for problems–such as searching and sorting–that are so common that a more efficient algorithm would greatly improve productivity. Still others seek to discover algorithms for new problems. Thus, as we said, the study of algorithms underlies much of computer science. But everything we have done so far has been a pencil-and-paper exercise. In terms of the definition of computer science that we gave in Chapter 1, we have been looking at the formal and mathematical properties of algorithms. It is time to move on to the next part of that definition: the hardware realizations of algorithms. When we execute real algorithms on real computers, those computers are electronic devices. How does an electronic device "understand" an algorithm and carry out its instructions? We begin to explore these questions in Chapter 4.

EXERCISES

1. a. Use Gauss's approach to find the following sum:

   $2 + 4 + 6 + \ldots + 100$

   b. Use Gauss's approach to find a formula for the sum of the even numbers from 2 to $2n$:

   $2 + 4 + 6 + \ldots + 2n$

   Your formula will be an expression involving n.

2. An English Christmas carol, "The Twelve Days of Christmas," dates from the late 1700s. The twelve verses in the song are cumulative, each verse adding an additional gift given by "my true love." The twelfth verse says "On the twelfth day of Christmas, my true love gave to me …"

   12 Drummers Drumming

   11 Pipers Piping

   10 Lords-a-Leaping

   … and so forth down to …

   3 French Hens

   2 Turtle Doves

   And a Partridge in a Pear Tree.

   a. Use Gauss's formula to find the total number of gifts given on Day 12.

   b. How many total gifts are given over all 12 days? *Hint:*

$$1(2)+2(3)+3(4)+\ldots+n(n+1)=\frac{n(n+1)(n+2)}{3}$$

3. The *Fibonacci sequence* of numbers is defined as follows: The first and second numbers are both 1. After that, each number in the sequence is the sum of the two preceding numbers. Thus, the Fibonacci sequence is as follows:

   1, 1, 2, 3, 5, 8, 13, 21, ...

   If $F(n)$ stands for the nth value in the sequence, then this definition can be expressed as

   $F(1) = 1$

   $F(2) = 1$

   $F(n) = F(n - 1) + F(n - 2)$ for $n > 2$

   a. The value of F at position n is defined using the value of F at two smaller positions. Something that is defined in terms of "smaller versions" of itself is said to be *recursive*, so the Fibonacci sequence is recursive. Using the definition of the Fibonacci sequence, compute the value of $F(20)$.

   b. A formula for $F(n)$ is

   $$F(n) = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

   Using the formula (and a calculator), compute the value of $F(20)$.

   c. What is your opinion on the relative clarity, elegance, and efficiency of the two algorithms (using the definition and using the formula) to compute $F(20)$? Would your answer change if you considered $F(100)$?

   d. If you have access to a spreadsheet, program it to compute successive values of the Fibonacci sequence. (*Hint:* Enter 1 in cells A1 and B1, and enter = A1 + B1 in cell C1. Continue along row 1 with a similar formula.) Use this method to compute $F(20)$.

   e. Elsewhere on the same spreadsheet, use one cell to enter the value of n, and another cell to enter the formula from part (b) above. (*Hint:* you will need the squareroot function and the power function.) Use this method to compute $F(20)$.

   f. On your spreadsheet, use both methods to compute $F(50)$. Which appears to be more efficient?

4. A tennis tournament has 342 players. A single match involves 2 players. The winner of a match will play the winner of a match in the next round, whereas losers are eliminated from the tournament. The 2 players who have won all previous rounds play in the final game, and the winner wins the tournament. What is the total number of matches needed to determine the winner?

   a. Here is one algorithm to answer this question. Compute 342/2 = 171 to get the number of pairs (matches) in the first round, which results in 171 winners to go on to the second round. Compute 171/2 = 85 with 1 left over, which results in 85 matches in the second round and 85 winners, plus the 1 left over, to go on to the third round. For the third round compute 86/2 = 43, so the third round has 43 matches, and so on. The total number of matches is 171 + 85 + 43 + ... Finish this process to find the total number of matches.

   b. Here is another algorithm to solve this problem. Each match results in exactly one loser, so there must be the same number of matches as losers in the tournament. Compute the total number of losers in the entire tournament. (*Hint:* This isn't really a computation; it is a one-sentence argument.)

   c. What is your opinion on the relative clarity, elegance, and efficiency of the two algorithms?

5. We have said that the average number of comparisons needed to find a target value in an n-element list using sequential search is slightly higher than $n/2$. In this problem, we find an exact expression for this average.

   a. Suppose the list has an odd number of items, say 15. At what position is the middle item? Using sequential search, how many comparisons

are required to find the middle item? Repeat this exercise with a few more odd numbers until you can do the following: If there are n items in the list and n is an odd number, write an expression for the number of comparisons required to find the middle item.

b. Suppose the list has an even number of items, say 16. At what positions are the two "middle" items? Using sequential search, how many comparisons are required to find each of these? What is the average of these two numbers? Repeat this exercise with a few more even numbers until you can do the following: If there are n items in the list and n is an even number, write an expression for the average number of comparisons required to find the two middle items.

c. Noting that half the items in a list fall before the midpoint and half after the midpoint, use your answer to parts a and b to write an exact expression for the average number of comparisons done using sequential search to find a target value that occurs in an n-element list.

6. Here is a list of seven names:

   Sherman, Jane, Ted, Elise, Raul, Maki, John

   Search this list for each name in turn, using sequential search and counting the number of comparisons for each name. Now take the seven comparison counts and find their average. Did you get a number that you expected? Why?

7. The American Museum of Natural History in New York City contains more than 32 million specimens and artifacts in its various collections, including the world's largest collection of dinosaur fossils. Many of these are in storage away from public view, but all must be carefully inventoried.

   a. Suppose the inventory is unordered (!) and a sequential search is done to locate a specific artifact. Given that the search is executed on a computer that can do 12,000 comparisons per second, about how much time on the average would the search require?

   b. Assuming the inventory is sorted, about how much time would a binary search require?

8. In the Flipping Pancakes box, the original algorithm given requires at most $2n - 3$ flips in the worst case. The claim is made that the new algorithm, which requires at most $(5n + 5)/3$ flips, is a better algorithm. How many pancakes do you need to have before the second algorithm is indeed faster? (Use a calculator or spreadsheet.)

9. Perform a selection sort on the list 7, 4, 2, 9, 6. Show the list after each exchange that has an effect on the list ordering.

10. The selection sort algorithm could be modified to stop when the unsorted section of the list contains only one number, because that one number must be in the correct position. Show that this modification would have no effect on the number of comparisons required to sort an n-element list.

Exercises 11–14 refer to another algorithm, called **bubble sort**, which sorts an n-element list. Bubble sort makes multiple passes through the list from front to back, each time exchanging pairs of entries that are out of order. Here is a pseudocode version:

1. Get values for n and the n list items
2. Set the marker U for the unsorted section at the end of the list
3. While the unsorted section has more than one element, do Steps 4 through 8
4. Set the current element marker C at the second element of the list
5. While C has not passed U, do Steps 6 and 7
6. If the item at position C is less than the item to its left, then exchange these two items
7. Move C to the right one position
8. Move U left one position
9. Stop

11. For each of the following lists, perform a bubble sort, and show the list after each exchange. Compare the number of exchanges done here and in the Practice Problem at the end of Section 3.3.3.

a. 4, 8, 2, 6

b. 12, 3, 6, 8, 2, 5, 7

c. D, B, G, F, A, C, E, H

12. Explain why the bubble sort algorithm does $\Theta(n^2)$ comparisons on an n-element list.

13. Suppose selection sort and bubble sort are both performed on a list that is already sorted. Does bubble sort do fewer exchanges than selection sort? Explain.

14. Bubble sort can be improved. **Smart bubble sort** keeps track of how many exchanges are done within any single pass through the unsorted section of the list. If no exchanges occur, then the list is sorted and the algorithm should stop.

    a. Write a pseudocode version of the smart bubble sort algorithm.

    b. Perform a smart bubble sort on the following list. How many comparisons are required?

       7, 4, 12, 9, 11

    c. Describe the best-case scenario for smart bubble sort on an n-element list. How many comparisons are required? How many exchanges are required?

    d. Under what circumstances does smart bubble sort do the same number of comparisons as regular bubble sort?

Exercises 15–17 refer to still another sorting algorithm, called **mergesort**. Mergesort breaks the list to be sorted into smaller and smaller lists until there is just a bunch of one-element (and thus obviously sorted) lists, then assembles the smaller sorted lists back together into larger and larger sorted lists. Here is a pseudocode version:

1. Get values for n and the n list items
2. While the current list has more than 1 item, do Steps 3 through 6
3. Split the list into two halves
4. Sort the first half of the list using mergesort
5. Sort the second half of the list using mergesort
6. Merge the two sorted halves A and B into a new sorted list C by comparing the next two items from A and B and always choosing the smaller value to go into C
7. Stop

Mergesort works by using the result of mergesort on two smaller lists, so mergesort, like the Fibonacci sequence in Exercise 3, is a recursive algorithm.

Step 6 in this algorithm deserves an example. Suppose that at one point in running mergesort we have two sorted lists A and B, as follows:

$A = 2, 9 \quad B = 6, 7$

To create C, we compare 2 from A and 6 from B. Because 2 is smaller, it is removed from A and goes into C.

$A = 9 \quad B = 6, 7 \quad C = 2$

Now compare 9 from A and 6 from B. Because 6 is smaller, it is removed from B and goes into C.

$A = 9 \quad B = 7 \quad C = 2, 6$

Comparing 9 and 7 results in

$A = 9 \quad B = \quad C = 2, 6, 7$

Finally, let us agree to count comparing 9 to nothing as a legitimate, if trivial, comparison resulting in 9, which is added to C, the final sorted list.

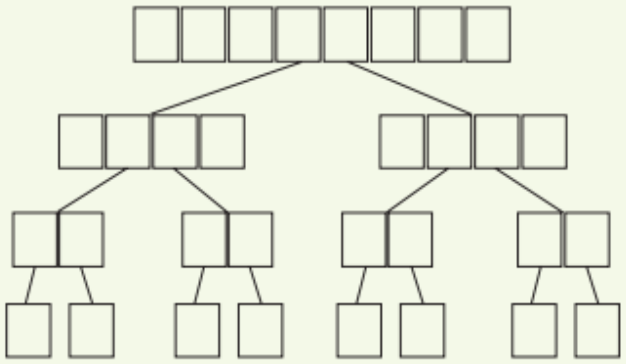
$A = \quad B = \quad C = 2, 6, 7, 9$

15. Show the steps in merging A and B into C where

    $A = 8, 12, 19, 34 \quad B = 3, 5, 15, 21$

16. Use mergesort to sort the list 6, 3, 1, 9. Count the total number of comparisons, including trivial comparisons.

17. The work in this algorithm occurs in Step 6 where sorted lists are merged together. The "work unit" is the comparison of two items. If A and B are each of size $k/2$, then k comparisons are needed to merge them into a sorted list C of size k. If the original list has 8 elements, then we can picture mergesort as shown on the next page:



Starting from the bottom of the tree, merges occur at three levels (1-element lists merge to 2-element lists, lists of size 2 merge to lists of size 4, and lists of size 4 merge to the final list of size 8) and, looking across the entire tree, at each level there are 8 comparisons done to create the level above it. Therefore the total amount of work done is $3 \times 8 = 24$ comparisons.

a. If the original list has n items, how many levels will the tree have at which merges occur? (*Hint:* How many times can the number n be cut in half?)

b. If the original list has n items, how many comparisons occur at each merge level?

c. What is the order of magnitude of mergesort?

d. Does your answer to part c agree with your answer to Exercise 16? Explain.

18. This exercise discusses a new algorithm to sort an n-element list.

a. A permutation of a list is any arrangement of the list items. For example, 2, 4 and 4, 2 are the two permutations of the list 2, 4. Find all permutations of the list 4, 3, 7.

b. Given an n-element list, the number of permutations can be counted as follows. There are n positions in the list. Any of the n items can occupy position 1:

$\underline{n}$, ___, ___, ___, ... ___, ___, ___

Once the item for position 1 has been chosen, any of the remaining $n - 1$ items can occupy position 2:

$\underline{n}$, $\underline{n-1}$, ___, ___, ... ___, ___, ___

There are then $n - 2$ choices for position 3, etc., until there is only one choice left for the last position:

$\underline{n}$, $\underline{n-1}$, $\underline{n-2}$, $\underline{n-3}$, ... 3, $\underline{2}$, $\underline{1}$

The total number of permutations is the product

$n(n-1)(n-2)(n-3)\ldots(3)(2)(1)$

This value is called n *factorial* and is denoted by $n!$. Compute the value of $3!$. How many permutations did you find in part a?

c. Here is a pseudocode description of the new sorting algorithm. It first generates all possible permutations of the list elements and then looks for which of these is the sorted-order permutation.

1. Get values for n and the n list items
2. Set the value of permutation counter i to 1
3. While ($i \leq n!$), do Steps 4 through 6
4. Create a new empty list of size n
5. Write the next permutation in this list
6. Increase the value of i by 1
7. Set the value of list counter j to 1
8. Set the value of Sorted to NO
9. While ($j \leq n!$) and (Sorted = NO), do Steps 10 through 13
10. Check whether list j is sorted
11. If list j is sorted then
12. Set the value of Sorted to YES
13. Increase the value of j by 1
14. Print list ($j - 1$)
15. Stop

To write this algorithm in complete detail, we would need to explain Step 5:

Write the next permutation in this list

and Step 10:

Check whether list j is sorted

in terms of more primitive operations. For simplicity, assume that each execution of Step 5

is one work unit and that each execution of Step 10 is also one work unit. Explain the best case for this algorithm and give an expression for the total number of work units required. Explain the worst case and give an expression for the total number of work units required.

d. Selection sort is $\Theta(n^2)$ and the new sorting algorithm is $\Theta(n!)$. Fill in the following table, assuming a work rate of 0.0001 seconds per unit of work.

	n			
Order	3	5	10	20
n^2				
$n!$				

e. Comment on the space efficiency of the new algorithm.

19. Algorithms A and B perform the same task. On input of size n, algorithm A executes $0.003n^2$ instructions, and algorithm B executes $243n$ instructions. Find the approximate value of n above which algorithm B is more efficient. (You may use a calculator or spreadsheet.)

20. Suppose a metropolitan area is divided into four school districts: 1, 2, 3, 4. The State Board of Education keeps track of the number of student transfers from one district to another and the student transfers within a district. This information is recorded each year in a 4×4 table as shown here. The entry in row 1, column 3 (314), for example, shows the number of student transfers from district 1 to district 3 for the year. The entry in row 1, column 1 (243) shows the number of student transfers within district 1.

	1	2	3	4
1	243	187	314	244
2	215	420	345	172
3	197	352	385	261
4	340	135	217	344

Suppose there are n school districts, and the Board of Education maintains an $n \times n$ table.

a. Write a pseudocode algorithm to print the table, that is, to print each of the entries in the table. Write an expression for the number of print statements the algorithm executes.

b. Write a pseudocode algorithm to print n copies of the table, one to give to each of the n school district supervisors. Write an expression for the number of print statements the algorithm executes.

c. What is the order of magnitude of the work done by the algorithm in part b if the unit of work is printing a table element?

21. Write the data list that results from running the shuffle-left algorithm to clean up the following data. Find the exact number of copies done.

3	0	0	2	6	7	0	0	5	1

22. Write the resulting data list and find the exact number of copies done by the converging-pointers algorithm when it is executed on the data in Exercise 21.

23. Explain in words how to modify the shuffle-left data cleanup algorithm to slightly reduce the number of copies it makes. (*Hint:* Must item n always be copied?) If this modified algorithm is run on the data list of Exercise 21, exactly how many copies are done?

24. The shuffle-left algorithm for data cleanup is supposed to perform $n(n - 1)$ copies on a list consisting of n 0s (zeros). Confirm this result for the following list:

0 0 0 0 0 0

25. Consider the following sorted list of names.

Arturo, Elsa, JoAnn, John, José, Lee, Snyder, Tracy

a. Use binary search to decide whether Elsa is in this list. What names will be compared with Elsa?

b. Use binary search to decide whether Tracy is in this list. What names will be compared with Tracy?

c. Use binary search to decide whether Emile is in this list. What names will be compared with Emile?

26. Use the binary search algorithm to decide whether 35 is in the following list:

    3, 6, 7, 9, 12, 14, 18, 21, 22, 31, 43

    What numbers will be compared with 35?

27. If a list is already sorted in ascending order, a modified sequential search algorithm can be used that compares against each element in turn, stopping if a list element exceeds the target value. Write a pseudocode version of this **short sequential search** algorithm.

28. This exercise refers to short sequential search (see Exercise 27).

    a. What is the worst-case number of comparisons of short sequential search on a sorted n-element list?

    b. What is the approximate average number of comparisons to find an element that is in a sorted list using short sequential search?

    c. Is short sequential search ever more efficient than regular sequential search? Explain.

29. Draw the tree structure that describes binary search on the eight-element list in Exercise 25. What is the number of comparisons in the worst case? Give an example of a name to search for that requires that many comparisons.

30. Draw the tree structure that describes binary search on a list with 16 elements. What is the number of comparisons in the worst case?

31. We want to find an exact formula for the number of comparisons that binary search requires in the worst case on an n-element list. (We already know the formula is $\Theta(\lg n)$.)

    a. If x is a number that is not an integer, then $\lfloor x \rfloor$, called the *floor function* of x, is defined to be the largest integer less than or equal to x. For example, $\lfloor 3.7 \rfloor = 3$ and $\lfloor 5 \rfloor = 5$. Find the following values: $\lfloor 1.2 \rfloor$, $\lfloor 2.3 \rfloor$, $\lfloor 8.9 \rfloor$, $\lfloor -4.6 \rfloor$.

    b. If n is not a power of 2, then $\lg n$ is not an integer. If n is between 8 and 16, for example, then $\lg n$ is between 3 and 4 (because $\lg 8 = 3$ and $\lg 16 = 4$). Complete the following table of values:

n	$\lfloor \lg n \rfloor$
2	1
3	
4	2
5	
6	
7	
8	3

    c. For $n = 2, 3, 4, 5, 6, 7, 8$, draw a tree structure similar to Figure 3.19 to describe the positions searched by binary search. For each value of n, use the tree structure to find the number of comparisons in the worst case, and complete the following table:

n	*Number of Compares, Worst Case*
2	
3	
4	3
5	
6	
7	3
8	

    d. Comparing the tables of parts b and c, find a formula involving $\lfloor \lg n \rfloor$ for the number of comparisons binary search requires in the worst case on an n-element list. Test your formula by drawing trees for other values of n.

32. Using the tree in Figure 3.19, find the number of comparisons to find each of items 1–7 in a seven-element list using binary search. Then find the average. Compare this with the worst case.

33. At the end of Section 3.4.2, we talked about the trade-off between using sequential search on an

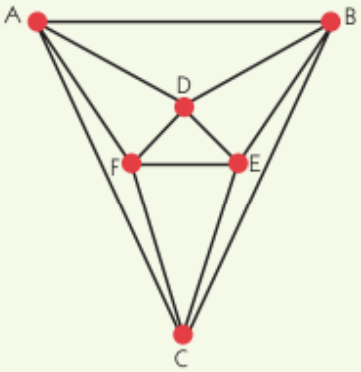
unsorted list as opposed to sorting the list and then using binary search. If the list size is $n = 100{,}000$, about how many worst-case searches must be done before the second alternative is better in terms of number of comparisons? (*Hint:* Let p represent the number of searches done.)

34. Suppose the pattern-matching problem is changed to require locating only the first instance, if any, of the pattern within the text.

    a. Describe the worst case, give an example, and give the exact number of comparisons (of a pattern character with a text character) required.

    b. Describe the best case, give an example, and give the exact number of comparisons required.

35. Suppose you use the brute force (unimproved) algorithm to search for a Hamiltonian circuit in the graph shown here.



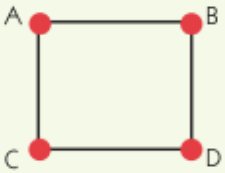
    a. How many links would such a circuit have?

    b. How many different paths would this algorithm examine?

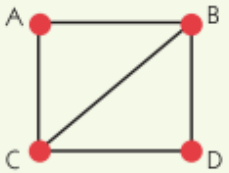
    c. If this algorithm is run on a computer where it takes 0.0001 seconds to examine a single path, what is the total time required to examine all paths?

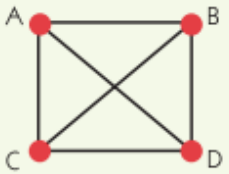
    d. On the same computer, what is the approximate total time required to examine all paths in a graph with 12 nodes, each of which has four choices for the next node?

36. An *Euler path* in a graph (named for the Swiss mathematician Leonhard Euler, 1707–1783) is a path that uses each edge of the graph exactly once. For example, this graph clearly has an Euler path A-B-D-C-A.



    a. Decide which of the graphs below have Euler paths, and write out such a path if one exists. (Unlike Hamiltonian circuits, an Euler path need not end at the same node from which it starts, and can go through a given node more than once.)

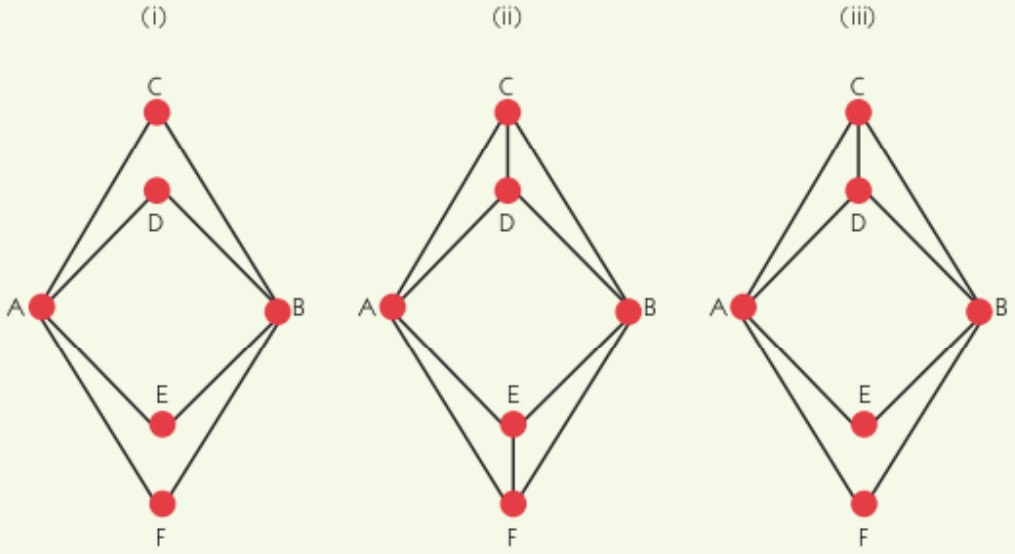




    b. Like the Hamiltonian circuit problem, there is a brute force algorithm to determine whether a graph has an Euler path, again by testing all possibilities. You probably used a variation of this algorithm to solve part a. But the Euler path problem has another solution. The *degree* of a node is the number of edges at that node. A node with odd degree is an *odd node* and a node with even degree is an *even node*. In the first graph of part a, nodes A and D are even (degree 2) and nodes B and C are odd (degree 3). An Euler path exists in any graph with exactly 0 or 2 odd nodes. Which of the graphs at the top of the next page have Euler paths and why? Write out such a path if one exists.

    c. Here is a pseudocode version of this algorithm for a graph with n nodes labeled 1 to n:

        1. Get values for n and all edges of the graph
        2. Set the value of *Odds* to 0
        3. Set the value of node counter i to 1



4. While ($i \leq n$), do Steps 5 through 13
5. Set the value of node counter j to 1
6. Set the value of *Degree* to 0
7. While ($j \leq n$), do Steps 8 through 10
8. If an edge *i-j* exists then
9. Increase *Degree* by 1
10. Increase j by 1
11. If *Degree* is an odd number then
12. Increase *Odds* by 1
13. Increase i by 1
14. If *Odds* > 2 then
15. Print the message 'No Euler path'

    Else
16. Print the message 'Euler path exists'
17. Stop

What is the order of magnitude of this algorithm where the "work unit" is checking whether an edge exists (Step 8)?

d. Is the Euler path problem intractable?

37. At about what value of n does an algorithm that does $100n^2$ instructions become more efficient than one that does $0.01(2^n)$ instructions? (Use a calculator or spreadsheet.)

38. a. An algorithm that is $\Theta(n)$ takes 10 seconds to execute on a particular computer when $n = 100$. How long would you expect it to take when $n = 500$?

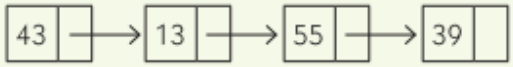
    b. An algorithm that is $\Theta(n^2)$ takes 10 seconds to execute on a particular computer when $n = 100$. How long would you expect it to take when $n = 500$?

39. Find an optimal solution to the bin-packing problem described in Section 3.5.

40. In the data cleanup problem, we assumed that the items were stored in a list with a fixed number of positions. Each item could be examined by giving its position in the list. This arrangement of data is called an array. Here is an array of four items:

43	13	55	39
1	2	3	4

Another way to arrange items is to have a way to locate the first item and then have each item "point to" the next item. This arrangement of data is called a *linked list*. Here are the same four items in a linked list arrangement:

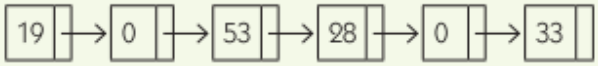


To examine any item in a linked list, one must start with the first item and follow the pointers to the desired item.

Unlike arrays, which are fixed in size, linked lists can shrink and grow. An item can be eliminated from a linked list by changing the pointer to that item so that it points to the next item instead.

a. Draw the linked list that results when item 13 is eliminated from the foregoing linked list.

b. Draw the linked list that results when data cleanup is performed on the following linked list.



c. Describe (informally) an algorithm to do data cleanup on a linked list. You may assume that neither the first item nor the last item has a value of 0, and you may assume the existence of operations such as "follow pointer" and "change pointer." If these operations are the unit of work used, show that your algorithm is an $\Theta(n)$ algorithm, where n is the number of items in the list.

41. Below is a pseudocode algorithm that prints a set of output values:

1. Get value for n
2. Set the value of k to 1
3. While k is less than or equal to n, do Steps 4 through 8
4. Set the value of j to one-half n
5. While j is greater than or equal to 1, do Steps 6 through 7
6. Print the value of j
7. Set the value of j to one-half its former value
8. Increase k by 1
9. Stop

a. Let n have the value 4. Write the values printed by this algorithm.

b. Let n have the value 8. Write the values printed by this algorithm.

c. Which of the following best describes the efficiency of this algorithm, where the "work unit" is printing a value?

$\Theta(n^2)$ $\Theta(n \lg n)$ $\Theta(n)$ $\Theta(\lg n)$

d. How many work units would you expect this algorithm to do if $n = 16$?

42. Chapter 2 contains an algorithm that finds the largest value in a list of n values.

a. What is the order of magnitude of the largest-value algorithm, where the "work unit" is comparisons of values from the list?

b. Suppose that you want to find the second-largest value in the list. Find the order of magnitude of the work done if you use the following algorithm: Sort the list, using selection sort, then directly get the second-largest value.

c. Suppose that you want to find the second-largest value in the list. Find the order of magnitude of the work done if you use the following algorithm: Run the largest-value algorithm twice, first to find (and eliminate from the list) the largest value, then to find the second-largest value.

CHALLENGE WORK

1. You are probably familiar with the children's song "Old MacDonald Had a Farm." The first verse is

   Old MacDonald had a farm, E-I-E-I-O.

   And on that farm he had a cow, E-I-E-I-O.

   With a moo-moo here and a moo-moo there,

   Here a moo, there a moo,

   Everywhere a moo-moo,

   Old MacDonald had a farm, E-I-E-I-O.

   In successive verses, more animals are added, and the middle refrain gets longer and longer. For example, the second verse is

   Old MacDonald had a farm, E-I-E-I-O.

   And on that farm he had a pig, E-I-E-I-O.

   With an oink-oink here and an oink-oink there,

   Here an oink, there an oink,

   Everywhere an oink-oink,

   With a moo-moo here and a moo-moo there,

   Here a moo, there a moo,

   Everywhere a moo-moo,

   Old MacDonald had a farm, E-I-E-I-O.

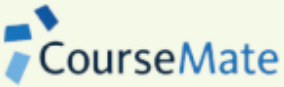
   a. Show that after n verses of this song have been sung, the total number of syllables sung would be given by the expression $22n(n + 1)/2 + 37n$. (You may assume that all animal names and all animal sounds consist of one syllable, as in cow, pig, moo, oink, and so on.)

   b. If singing this song is the algorithm, and the "work unit" is singing one syllable, what is the order of magnitude of the algorithm?[1]

2. *Linear programming* involves selecting values for a large number of quantities so that they satisfy a set of inequalities (such as $x + y + z \leq 100$) while at the same time maximizing (or minimizing) some particular function of these variables. Linear programming has many applications in communications and manufacturing. A trial-and-error approach to a linear programming problem would involve guessing at values for these variables until all of the inequalities are satisfied, but this might not produce the desired maximum (or minimum) value. In addition, real-world problems may involve hundreds or thousands of variables. A common algorithm to solve linear programming problems is called the *simplex method*. Although the simplex method works well for many common applications, including those that involve thousands of variables, its worst-case order of magnitude is exponential. Find information on the work of N. Karmarkar of Bell Labs, who discovered another algorithm for linear programming that is of polynomial order in the worst case and is faster than the simplex method in average cases.

[1]This exercise is based on work found in Chavey, D., "Songs and the Analysis of Algorithms," *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium* (1996), pp. 4–8.

ADDITIONAL RESOURCES



For additional print and/or online resources relevant to this chapter, visit the CourseMate for this text at login.cengagebrain.com.