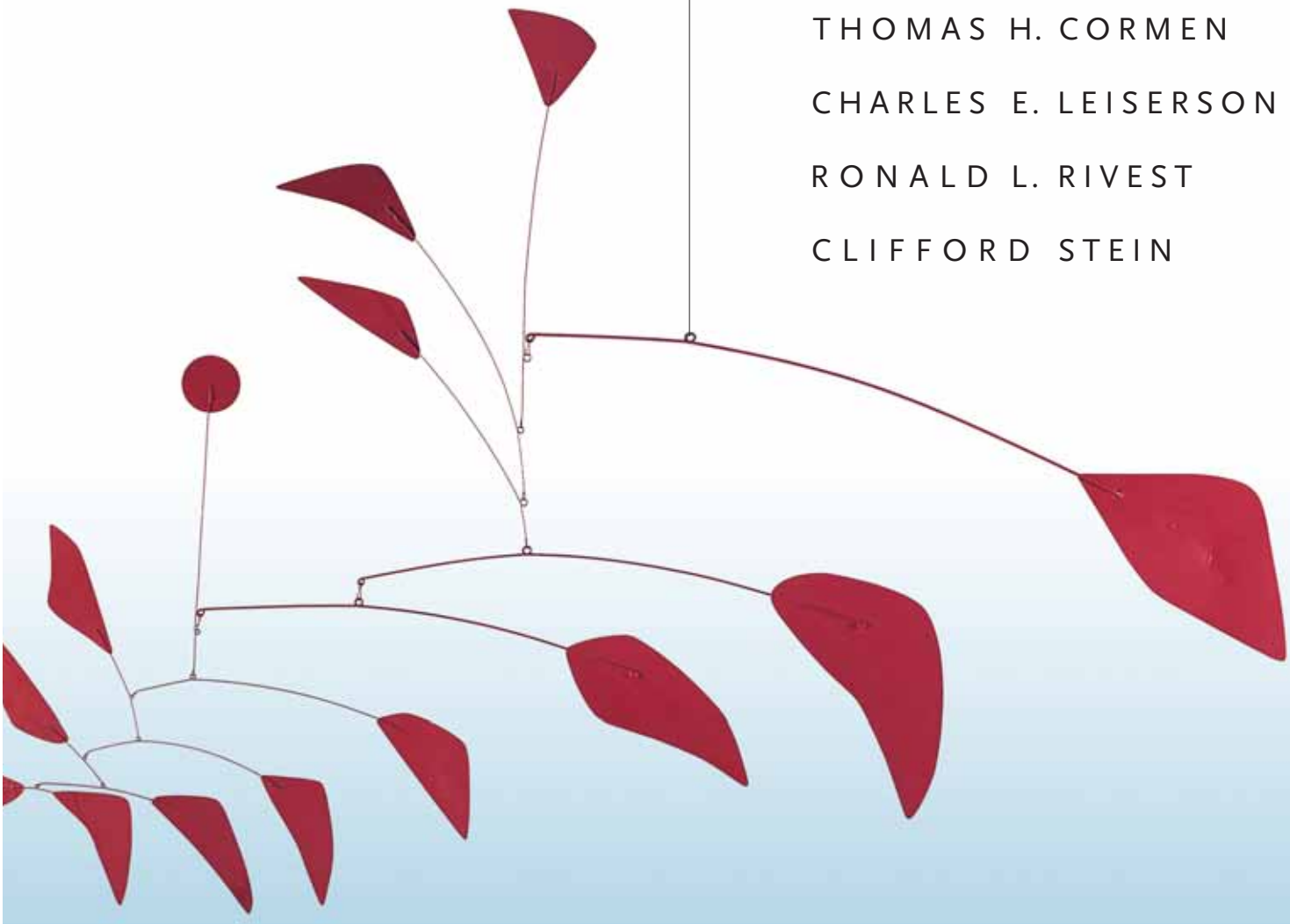
THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN



# INTRODUCTION TO ALGORITHMS

THIRD EDITION

# Introduction to Algorithms

*Third Edition*

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

# Introduction to Algorithms

***Third Edition***

The MIT Press
Cambridge, Massachusetts  London, England

This book was set in Times Roman and Mathtime Pro 2 by the authors.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Introduction to algorithms / Thomas H. Cormen . . . [et al.].—3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-03384-8 (hardcover : alk. paper)—ISBN 978-0-262-53305-8 (pbk. : alk. paper)
1. Computer programming. 2. Computer algorithms. I. Cormen, Thomas H.

QA76.6.I5858 2009
005.1—dc22

2009008593

10 9 8 7 6 5 4 3 2

# Contents

## VIII Appendix: Mathematical Background

# Preface

Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing.

This book provides a comprehensive introduction to the modern study of computer algorithms. It presents many algorithms and covers them in considerable depth, yet makes their design and analysis accessible to all levels of readers. We have tried to keep explanations elementary without sacrificing depth of coverage or mathematical rigor.

Each chapter presents an algorithm, a design technique, an application area, or a related topic. Algorithms are described in English and in a pseudocode designed to be readable by anyone who has done a little programming. The book contains 244 figures—many with multiple parts—illustrating how the algorithms work. Since we emphasize *efficiency* as a design criterion, we include careful analyses of the running times of all our algorithms.

The text is intended primarily for use in undergraduate or graduate courses in algorithms or data structures. Because it discusses engineering issues in algorithm design, as well as mathematical aspects, it is equally well suited for self-study by technical professionals.

In this, the third edition, we have once again updated the entire book. The changes cover a broad spectrum, including new chapters, revised pseudocode, and a more active writing style.

### To the teacher

We have designed this book to be both versatile and complete. You should find it useful for a variety of courses, from an undergraduate course in data structures up through a graduate course in algorithms. Because we have provided considerably more material than can fit in a typical one-term course, you can consider this book to be a "buffet" or "smorgasbord" from which you can pick and choose the material that best supports the course you wish to teach.

You should find it easy to organize your course around just the chapters you need. We have made chapters relatively self-contained, so that you need not worry about an unexpected and unnecessary dependence of one chapter on another. Each chapter presents the easier material first and the more difficult material later, with section boundaries marking natural stopping points. In an undergraduate course, you might use only the earlier sections from a chapter; in a graduate course, you might cover the entire chapter.

We have included 957 exercises and 158 problems. Each section ends with exercises, and each chapter ends with problems. The exercises are generally short questions that test basic mastery of the material. Some are simple self-check thought exercises, whereas others are more substantial and are suitable as assigned homework. The problems are more elaborate case studies that often introduce new material; they often consist of several questions that lead the student through the steps required to arrive at a solution.

Departing from our practice in previous editions of this book, we have made publicly available solutions to some, but by no means all, of the problems and exercises. Our Web site, http://mitpress.mit.edu/algorithms/, links to these solutions. You will want to check this site to make sure that it does not contain the solution to an exercise or problem that you plan to assign. We expect the set of solutions that we post to grow slowly over time, so you will need to check it each time you teach the course.

We have starred ($\star$) the sections and exercises that are more suitable for graduate students than for undergraduates. A starred section is not necessarily more difficult than an unstarred one, but it may require an understanding of more advanced mathematics. Likewise, starred exercises may require an advanced background or more than average creativity.

### To the student

We hope that this textbook provides you with an enjoyable introduction to the field of algorithms. We have attempted to make every algorithm accessible and interesting. To help you when you encounter unfamiliar or difficult algorithms, we describe each one in a step-by-step manner. We also provide careful explanations of the mathematics needed to understand the analysis of the algorithms. If you already have some familiarity with a topic, you will find the chapters organized so that you can skim introductory sections and proceed quickly to the more advanced material.

This is a large book, and your class will probably cover only a portion of its material. We have tried, however, to make this a book that will be useful to you now as a course textbook and also later in your career as a mathematical desk reference or an engineering handbook.

What are the prerequisites for reading this book?

- You should have some programming experience. In particular, you should understand recursive procedures and simple data structures such as arrays and linked lists.
- You should have some facility with mathematical proofs, and especially proofs by mathematical induction. A few portions of the book rely on some knowledge of elementary calculus. Beyond that, Parts I and VIII of this book teach you all the mathematical techniques you will need.

We have heard, loud and clear, the call to supply solutions to problems and exercises. Our Web site, http://mitpress.mit.edu/algorithms/, links to solutions for a few of the problems and exercises. Feel free to check your solutions against ours. We ask, however, that you do not send your solutions to us.

### To the professional

The wide range of topics in this book makes it an excellent handbook on algorithms. Because each chapter is relatively self-contained, you can focus in on the topics that most interest you.

Most of the algorithms we discuss have great practical utility. We therefore address implementation concerns and other engineering issues. We often provide practical alternatives to the few algorithms that are primarily of theoretical interest.

If you wish to implement any of the algorithms, you should find the translation of our pseudocode into your favorite programming language to be a fairly straightforward task. We have designed the pseudocode to present each algorithm clearly and succinctly. Consequently, we do not address error-handling and other software-engineering issues that require specific assumptions about your programming environment. We attempt to present each algorithm simply and directly without allowing the idiosyncrasies of a particular programming language to obscure its essence.

We understand that if you are using this book outside of a course, then you might be unable to check your solutions to problems and exercises against solutions provided by an instructor. Our Web site, http://mitpress.mit.edu/algorithms/, links to solutions for some of the problems and exercises so that you can check your work. Please do not send your solutions to us.

### To our colleagues

We have supplied an extensive bibliography and pointers to the current literature. Each chapter ends with a set of chapter notes that give historical details and references. The chapter notes do not provide a complete reference to the whole field

of algorithms, however. Though it may be hard to believe for a book of this size, space constraints prevented us from including many interesting algorithms.

Despite myriad requests from students for solutions to problems and exercises, we have chosen as a matter of policy not to supply references for problems and exercises, to remove the temptation for students to look up a solution rather than to find it themselves.

### Changes for the third edition

What has changed between the second and third editions of this book? The magnitude of the changes is on a par with the changes between the first and second editions. As we said about the second-edition changes, depending on how you look at it, the book changed either not much or quite a bit.

A quick look at the table of contents shows that most of the second-edition chapters and sections appear in the third edition. We removed two chapters and one section, but we have added three new chapters and two new sections apart from these new chapters.

We kept the hybrid organization from the first two editions. Rather than organizing chapters by only problem domains or according only to techniques, this book has elements of both. It contains technique-based chapters on divide-and-conquer, dynamic programming, greedy algorithms, amortized analysis, NP-Completeness, and approximation algorithms. But it also has entire parts on sorting, on data structures for dynamic sets, and on algorithms for graph problems. We find that although you need to know how to apply techniques for designing and analyzing algorithms, problems seldom announce to you which techniques are most amenable to solving them.

Here is a summary of the most significant changes for the third edition:

- We added new chapters on van Emde Boas trees and multithreaded algorithms, and we have broken out material on matrix basics into its own appendix chapter.
- We revised the chapter on recurrences to more broadly cover the divide-and-conquer technique, and its first two sections apply divide-and-conquer to solve two problems. The second section of this chapter presents Strassen's algorithm for matrix multiplication, which we have moved from the chapter on matrix operations.
- We removed two chapters that were rarely taught: binomial heaps and sorting networks. One key idea in the sorting networks chapter, the 0-1 principle, appears in this edition within Problem 8-7 as the 0-1 sorting lemma for compare-exchange algorithms. The treatment of Fibonacci heaps no longer relies on binomial heaps as a precursor.

- We revised our treatment of dynamic programming and greedy algorithms. Dynamic programming now leads off with a more interesting problem, rod cutting, than the assembly-line scheduling problem from the second edition. Furthermore, we emphasize memoization a bit more than we did in the second edition, and we introduce the notion of the subproblem graph as a way to understand the running time of a dynamic-programming algorithm. In our opening example of greedy algorithms, the activity-selection problem, we get to the greedy algorithm more directly than we did in the second edition.
- The way we delete a node from binary search trees (which includes red-black trees) now guarantees that the node requested for deletion is the node that is actually deleted. In the first two editions, in certain cases, some other node would be deleted, with its contents moving into the node passed to the deletion procedure. With our new way to delete nodes, if other components of a program maintain pointers to nodes in the tree, they will not mistakenly end up with stale pointers to nodes that have been deleted.
- The material on flow networks now bases flows entirely on edges. This approach is more intuitive than the net flow used in the first two editions.
- With the material on matrix basics and Strassen's algorithm moved to other chapters, the chapter on matrix operations is smaller than in the second edition.
- We have modified our treatment of the Knuth-Morris-Pratt string-matching algorithm.
- We corrected several errors. Most of these errors were posted on our Web site of second-edition errata, but a few were not.
- Based on many requests, we changed the syntax (as it were) of our pseudocode. We now use "$=$" to indicate assignment and "==" to test for equality, just as C, C++, Java, and Python do. Likewise, we have eliminated the keywords **do** and **then** and adopted "**//**" as our comment-to-end-of-line symbol. We also now use dot-notation to indicate object attributes. Our pseudocode remains procedural, rather than object-oriented. In other words, rather than running methods on objects, we simply call procedures, passing objects as parameters.
- We added 100 new exercises and 28 new problems. We also updated many bibliography entries and added several new ones.
- Finally, we went through the entire book and rewrote sentences, paragraphs, and sections to make the writing clearer and more active.

### Web site

You can use our Web site, http://mitpress.mit.edu/algorithms/, to obtain supplementary information and to communicate with us. The Web site links to a list of known errors, solutions to selected exercises and problems, and (of course) a list explaining the corny professor jokes, as well as other content that we might add. The Web site also tells you how to report errors or make suggestions.

### How we produced this book

Like the second edition, the third edition was produced in LaTeX $2_\varepsilon$. We used the Times font with mathematics typeset using the MathTime Pro 2 fonts. We thank Michael Spivak from Publish or Perish, Inc., Lance Carnes from Personal TeX, Inc., and Tim Tregubov from Dartmouth College for technical support. As in the previous two editions, we compiled the index using Windex, a C program that we wrote, and the bibliography was produced with BibTeX. The PDF files for this book were created on a MacBook running OS 10.5.

We drew the illustrations for the third edition using MacDraw Pro, with some of the mathematical expressions in illustrations laid in with the psfrag package for LaTeX $2_\varepsilon$. Unfortunately, MacDraw Pro is legacy software, having not been marketed for over a decade now. Happily, we still have a couple of Macintoshes that can run the Classic environment under OS 10.4, and hence they can run MacDraw Pro—mostly. Even under the Classic environment, we find MacDraw Pro to be far easier to use than any other drawing software for the types of illustrations that accompany computer-science text, and it produces beautiful output.[1] Who knows how long our pre-Intel Macs will continue to run, so if anyone from Apple is listening: *Please create an OS X-compatible version of MacDraw Pro!*

### Acknowledgments for the third edition

We have been working with the MIT Press for over two decades now, and what a terrific relationship it has been! We thank Ellen Faran, Bob Prior, Ada Brunstein, and Mary Reilly for their help and support.

We were geographically distributed while producing the third edition, working in the Dartmouth College Department of Computer Science, the MIT Computer

[1] We investigated several drawing programs that run under Mac OS X, but all had significant shortcomings compared with MacDraw Pro. We briefly attempted to produce the illustrations for this book with a different, well known drawing program. We found that it took at least five times as long to produce each illustration as it took with MacDraw Pro, and the resulting illustrations did not look as good. Hence the decision to revert to MacDraw Pro running on older Macintoshes.

Science and Artificial Intelligence Laboratory, and the Columbia University Department of Industrial Engineering and Operations Research. We thank our respective universities and colleagues for providing such supportive and stimulating environments.

Julie Sussman, P.P.A., once again bailed us out as the technical copyeditor. Time and again, we were amazed at the errors that eluded us, but that Julie caught. She also helped us improve our presentation in several places. If there is a Hall of Fame for technical copyeditors, Julie is a sure-fire, first-ballot inductee. She is nothing short of phenomenal. Thank you, thank you, thank you, Julie! Priya Natarajan also found some errors that we were able to correct before this book went to press. Any errors that remain (and undoubtedly, some do) are the responsibility of the authors (and probably were inserted after Julie read the material).

The treatment for van Emde Boas trees derives from Erik Demaine's notes, which were in turn influenced by Michael Bender. We also incorporated ideas from Javed Aslam, Bradley Kuszmaul, and Hui Zha into this edition.

The chapter on multithreading was based on notes originally written jointly with Harald Prokop. The material was influenced by several others working on the Cilk project at MIT, including Bradley Kuszmaul and Matteo Frigo. The design of the multithreaded pseudocode took its inspiration from the MIT Cilk extensions to C and by Cilk Arts's Cilk++ extensions to C++.

We also thank the many readers of the first and second editions who reported errors or submitted suggestions for how to improve this book. We corrected all the bona fide errors that were reported, and we incorporated as many suggestions as we could. We rejoice that the number of such contributors has grown so great that we must regret that it has become impractical to list them all.

Finally, we thank our wives—Nicole Cormen, Wendy Leiserson, Gail Rivest, and Rebecca Ivry—and our children—Ricky, Will, Debby, and Katie Leiserson; Alex and Christopher Rivest; and Molly, Noah, and Benjamin Stein—for their love and support while we prepared this book. The patience and encouragement of our families made this project possible. We affectionately dedicate this book to them.

THOMAS H. CORMEN *Lebanon, New Hampshire*
CHARLES E. LEISERSON *Cambridge, Massachusetts*
RONALD L. RIVEST *Cambridge, Massachusetts*
CLIFFORD STEIN *New York, New York*

*February 2009*

# Introduction to Algorithms

*Third Edition*

# *I Foundations*

## Introduction

This part will start you thinking about designing and analyzing algorithms. It is intended to be a gentle introduction to how we specify algorithms, some of the design strategies we will use throughout this book, and many of the fundamental ideas used in algorithm analysis. Later parts of this book will build upon this base.

Chapter 1 provides an overview of algorithms and their place in modern computing systems. This chapter defines what an algorithm is and lists some examples. It also makes a case that we should consider algorithms as a technology, alongside technologies such as fast hardware, graphical user interfaces, object-oriented systems, and networks.

In Chapter 2, we see our first algorithms, which solve the problem of sorting a sequence of $n$ numbers. They are written in a pseudocode which, although not directly translatable to any conventional programming language, conveys the structure of the algorithm clearly enough that you should be able to implement it in the language of your choice. The sorting algorithms we examine are insertion sort, which uses an incremental approach, and merge sort, which uses a recursive technique known as "divide-and-conquer." Although the time each requires increases with the value of $n$, the rate of increase differs between the two algorithms. We determine these running times in Chapter 2, and we develop a useful notation to express them.

Chapter 3 precisely defines this notation, which we call asymptotic notation. It starts by defining several asymptotic notations, which we use for bounding algorithm running times from above and/or below. The rest of Chapter 3 is primarily a presentation of mathematical notation, more to ensure that your use of notation matches that in this book than to teach you new mathematical concepts.

Chapter 4 delves further into the divide-and-conquer method introduced in Chapter 2. It provides additional examples of divide-and-conquer algorithms, including Strassen's surprising method for multiplying two square matrices. Chapter 4 contains methods for solving recurrences, which are useful for describing the running times of recursive algorithms. One powerful technique is the "master method," which we often use to solve recurrences that arise from divide-and-conquer algorithms. Although much of Chapter 4 is devoted to proving the correctness of the master method, you may skip this proof yet still employ the master method.

Chapter 5 introduces probabilistic analysis and randomized algorithms. We typically use probabilistic analysis to determine the running time of an algorithm in cases in which, due to the presence of an inherent probability distribution, the running time may differ on different inputs of the same size. In some cases, we assume that the inputs conform to a known probability distribution, so that we are averaging the running time over all possible inputs. In other cases, the probability distribution comes not from the inputs but from random choices made during the course of the algorithm. An algorithm whose behavior is determined not only by its input but by the values produced by a random-number generator is a randomized algorithm. We can use randomized algorithms to enforce a probability distribution on the inputs—thereby ensuring that no particular input always causes poor performance—or even to bound the error rate of algorithms that are allowed to produce incorrect results on a limited basis.

Appendices A–D contain other mathematical material that you will find helpful as you read this book. You are likely to have seen much of the material in the appendix chapters before having read this book (although the specific definitions and notational conventions we use may differ in some cases from what you have seen in the past), and so you should think of the Appendices as reference material. On the other hand, you probably have not already seen most of the material in Part I. All the chapters in Part I and the Appendices are written with a tutorial flavor.

# 1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers? In this chapter, we will answer these questions.

## 1.1 Algorithms

Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

For example, we might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here is how we formally define the ***sorting problem***:

**Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

**Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

For example, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such an input sequence is called an ***instance*** of the sorting problem. In general, an ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of good sorting algorithms at our disposal. Which algorithm is best for a given application depends on—among other factors—the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output. We say that a correct algorithm ***solves*** the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate. We shall see an example of an algorithm with a controllable error rate in Chapter 31 when we study algorithms for finding large prime numbers. Ordinarily, however, we shall be concerned only with correct algorithms.

An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

### What kinds of problems are solved by algorithms?

Sorting is by no means the only computational problem for which algorithms have been developed. (You probably suspected as much when you saw the size of this book.) Practical applications of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has made great progress toward the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although the solutions to the various problems involved are beyond the scope of this book, many methods to solve these biological problems use ideas from several of the chapters in this book, thereby enabling scientists to accomplish tasks while using resources efficiently. The savings are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.
- The Internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data will travel (techniques for solving such problems appear in

Chapter 24), and using a search engine to quickly find pages on which particular information resides (related techniques are in Chapters 11 and 32).

- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures (covered in Chapter 31), which are based on numerical algorithms and number theory.
- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming, which we shall study in Chapter 29.

Although some of the details of these examples are beyond the scope of this book, we do give underlying techniques that apply to these problems and problem areas. We also show how to solve many specific problems, including the following:

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and we wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if we disallow routes that cross over themselves. How do we choose which of all possible routes is the shortest? Here, we model the road map (which is itself a model of the actual roads) as a graph (which we will meet in Part VI and Appendix B), and we wish to find the shortest path from one vertex to another in the graph. We shall see how to solve this problem efficiently in Chapter 24.
- We are given two ordered sequences of symbols, $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and we wish to find a longest common subsequence of $X$ and $Y$. A subsequence of $X$ is just $X$ with some (or possibly all or none) of its elements removed. For example, one subsequence of $\langle A, B, C, D, E, F, G \rangle$ would be $\langle B, C, E, G \rangle$. The length of a longest common subsequence of $X$ and $Y$ gives one measure of how similar these two sequences are. For example, if the two sequences are base pairs in DNA strands, then we might consider them similar if they have a long common subsequence. If $X$ has $m$ symbols and $Y$ has $n$ symbols, then $X$ and $Y$ have $2^m$ and $2^n$ possible subsequences,

respectively. Selecting all possible subsequences of $X$ and $Y$ and matching them up could take a prohibitively long time unless $m$ and $n$ are very small. We shall see in Chapter 15 how to use a general technique known as dynamic programming to solve this problem much more efficiently.

- We are given a mechanical design in terms of a library of parts, where each part may include instances of other parts, and we need to list the parts in order so that each part appears before any part that uses it. If the design comprises $n$ parts, then there are $n!$ possible orders, where $n!$ denotes the factorial function. Because the factorial function grows faster than even an exponential function, we cannot feasibly generate each possible order and then verify that, within that order, each part appears before the parts using it (unless we have only a few parts). This problem is an instance of topological sorting, and we shall see in Chapter 22 how to solve this problem efficiently.
- We are given $n$ points in the plane, and we wish to find the convex hull of these points. The convex hull is the smallest convex polygon containing the points. Intuitively, we can think of each point as being represented by a nail sticking out from a board. The convex hull would be represented by a tight rubber band that surrounds all the nails. Each nail around which the rubber band makes a turn is a vertex of the convex hull. (See Figure 33.6 on page 1029 for an example.) Any of the $2^n$ subsets of the points might be the vertices of the convex hull. Knowing which points are vertices of the convex hull is not quite enough, either, since we also need to know the order in which they appear. There are many choices, therefore, for the vertices of the convex hull. Chapter 33 gives two good methods for finding the convex hull.

These lists are far from exhaustive (as you again have probably surmised from this book's heft), but exhibit two characteristics that are common to many interesting algorithmic problems:

1. They have many candidate solutions, the overwhelming majority of which do not solve the problem at hand. Finding one that does, or one that is "best," can present quite a challenge.
2. They have practical applications. Of the problems in the above list, finding the shortest path provides the easiest examples. A transportation firm, such as a trucking or railroad company, has a financial interest in finding shortest paths through a road or rail network because taking shorter paths results in lower labor and fuel costs. Or a routing node on the Internet may need to find the shortest path through the network in order to route a message quickly. Or a person wishing to drive from New York to Boston may want to find driving directions from an appropriate Web site, or she may use her GPS while driving.

Not every problem solved by algorithms has an easily identified set of candidate solutions. For example, suppose we are given a set of numerical values representing samples of a signal, and we want to compute the discrete Fourier transform of these samples. The discrete Fourier transform converts the time domain to the frequency domain, producing a set of numerical coefficients, so that we can determine the strength of various frequencies in the sampled signal. In addition to lying at the heart of signal processing, discrete Fourier transforms have applications in data compression and multiplying large polynomials and integers. Chapter 30 gives an efficient algorithm, the fast Fourier transform (commonly called the FFT), for this problem, and the chapter also sketches out the design of a hardware circuit to compute the FFT.

### Data structures

This book also contains several data structures. A ***data structure*** is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

### Technique

Although you can use this book as a "cookbook" for algorithms, you may someday encounter a problem for which you cannot readily find a published algorithm (many of the exercises and problems in this book, for example). This book will teach you techniques of algorithm design and analysis so that you can develop algorithms on your own, show that they give the correct answer, and understand their efficiency. Different chapters address different aspects of algorithmic problem solving. Some chapters address specific problems, such as finding medians and order statistics in Chapter 9, computing minimum spanning trees in Chapter 23, and determining a maximum flow in a network in Chapter 26. Other chapters address techniques, such as divide-and-conquer in Chapter 4, dynamic programming in Chapter 15, and amortized analysis in Chapter 17.

### Hard problems

Most of this book is about efficient algorithms. Our usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efficient solution is known. Chapter 34 studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven

that an efficient algorithm for one cannot exist. In other words, no one knows whether or not efficient algorithms exist for NP-complete problems. Second, the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Computer scientists are intrigued by how a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

You should know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If you can show that the problem is NP-complete, you can instead spend your time developing an efficient algorithm that gives a good, but not the best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each day, it loads up each delivery truck at the depot and sends it around to deliver goods to several addresses. At the end of the day, each truck must end up back at the depot so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by each truck. This problem is the well-known "traveling-salesman problem," and it is NP-complete. It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that give an overall distance which is not too far above the smallest possible. Chapter 35 discusses such "approximation algorithms."

### Parallelism

For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, however: because power density increases superlinearly with clock speed, chips run the risk of melting once their clock speeds become high enough. In order to perform more computations per second, therefore, chips are being designed to contain not just one but several processing "cores." We can liken these multicore computers to several sequential computers on a single chip; in other words, they are a type of "parallel computer." In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind. Chapter 27 presents a model for "multithreaded" algorithms, which take advantage of multiple cores. This model has advantages from a theoretical standpoint, and it forms the basis of several successful computer programs, including a championship chess program.

### Exercises

***1.1-1***
Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

***1.1-2***
Other than speed, what other measures of efficiency might one use in a real-world setting?

***1.1-3***
Select a data structure that you have seen previously, and discuss its strengths and limitations.

***1.1-4***
How are the shortest-path and traveling-salesman problems given above similar? How are they different?

***1.1-5***
Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

## 1.2 Algorithms as a technology

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement.

Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

### Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort $n$ items, where $c_1$ is a constant that does not depend on $n$. That is, it takes time roughly proportional to $n^2$. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and $c_2$ is another constant that also does not depend on $n$. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size $n$. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of $n$ in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when $n$ equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size $n$ becomes large enough, merge sort's advantage of $\lg n$ vs. $n$ will more than compensate for the difference in constant factors. No matter how much smaller $c_1$ is than $c_2$, there will always be a crossover point beyond which merge sort is faster.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort $n$ numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50 n \lg n$ instructions. To sort 10 million numbers, computer A takes

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)} ,$$

while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)} .$$

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.

### Algorithms and other technologies

The example above shows that we should consider algorithms, like computer hardware, as a ***technology***. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

You might wonder whether algorithms are truly that important on contemporary computers in light of other advanced technologies, such as

- advanced computer architectures and fabrication technologies,
- easy-to-use, intuitive, graphical user interfaces (GUIs),
- object-oriented systems,
- integrated Web technologies, and
- fast networking, both wired and wireless.

The answer is yes. Although some applications do not explicitly require algorithmic content at the application level (such as some simple, Web-based applications), many do. For example, consider a Web-based service that determines how to travel from one location to another. Its implementation would rely on fast hardware, a graphical user interface, wide-area networking, and also possibly on object orientation. However, it would also require algorithms for certain operations, such as finding routes (probably using a shortest-path algorithm), rendering maps, and interpolating addresses.

Moreover, even an application that does not require algorithmic content at the application level relies heavily upon algorithms. Does the application rely on fast hardware? The hardware design used algorithms. Does the application rely on graphical user interfaces? The design of any GUI relies on algorithms. Does the application rely on networking? Routing in networks relies heavily on algorithms. Was the application written in a language other than machine code? Then it was processed by a compiler, interpreter, or assembler, all of which make extensive use

of algorithms. Algorithms are at the core of most technologies used in contemporary computers.

Furthermore, with the ever-increasing capacities of computers, we use them to solve larger problems than ever before. As we saw in the above comparison between insertion sort and merge sort, it is at larger problem sizes that the differences in efficiency between algorithms become particularly prominent.

Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled programmers from the novices. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.

### Exercises

***1.2-1***
Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

***1.2-2***
Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size $n$, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of $n$ does insertion sort beat merge sort?

***1.2-3***
What is the smallest value of $n$ such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is $2^n$ on the same machine?

## Problems

***1-1 Comparison of running times***
For each function $f(n)$ and time $t$ in the following table, determine the largest size $n$ of a problem that can be solved in time $t$, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

| | 1 second | 1 minute | 1 hour | 1 day | 1 month | 1 year | 1 century |
|---|---|---|---|---|---|---|---|
| $\lg n$ | | | | | | | |
| $\sqrt{n}$ | | | | | | | |
| $n$ | | | | | | | |
| $n \lg n$ | | | | | | | |
| $n^2$ | | | | | | | |
| $n^3$ | | | | | | | |
| $2^n$ | | | | | | | |
| $n!$ | | | | | | | |

## Chapter notes

There are many excellent texts on the general topic of algorithms, including those by Aho, Hopcroft, and Ullman [5, 6]; Baase and Van Gelder [28]; Brassard and Bratley [54]; Dasgupta, Papadimitriou, and Vazirani [82]; Goodrich and Tamassia [148]; Hofri [175]; Horowitz, Sahni, and Rajasekaran [181]; Johnsonbaugh and Schaefer [193]; Kingston [205]; Kleinberg and Tardos [208]; Knuth [209, 210, 211]; Kozen [220]; Levitin [235]; Manber [242]; Mehlhorn [249, 250, 251]; Purdom and Brown [287]; Reingold, Nievergelt, and Deo [293]; Sedgewick [306]; Sedgewick and Flajolet [307]; Skiena [318]; and Wilf [356]. Some of the more practical aspects of algorithm design are discussed by Bentley [42, 43] and Gonnet [145]. Surveys of the field of algorithms can also be found in the *Handbook of Theoretical Computer Science, Volume A* [342] and the CRC *Algorithms and Theory of Computation Handbook* [25]. Overviews of the algorithms used in computational biology can be found in textbooks by Gusfield [156], Pevzner [275], Setubal and Meidanis [310], and Waterman [350].

# 2 Getting Started

This chapter will familiarize you with the framework we shall use throughout the book to think about the design and analysis of algorithms. It is self-contained, but it does include several references to material that we introduce in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We define a "pseudocode" that should be familiar to you if you have done computer programming, and we use it to show how we shall specify our algorithms. Having specified the insertion sort algorithm, we then argue that it correctly sorts, and we analyze its running time. The analysis introduces a notation that focuses on how that time increases with the number of items to be sorted. Following our discussion of insertion sort, we introduce the divide-and-conquer approach to the design of algorithms and use it to develop an algorithm called merge sort. We end with an analysis of merge sort's running time.

## 2.1 Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1:

**Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

**Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The numbers that we wish to sort are also known as the ***keys***. Although conceptually we are sorting a sequence, the input comes to us in the form of an array with $n$ elements.

In this book, we shall typically describe algorithms as programs written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python, or Pascal. If you have been introduced to any of these languages, you should have little trouble

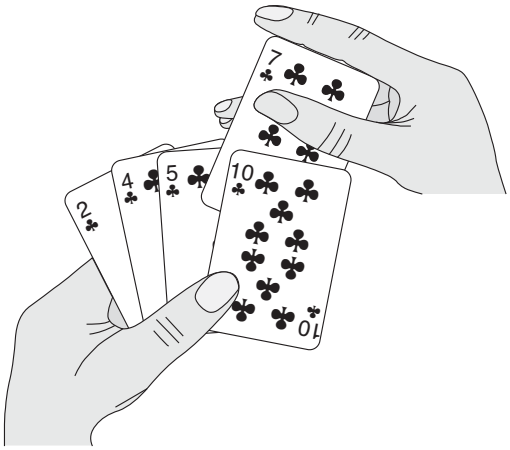



**Figure 2.1** Sorting a hand of cards using insertion sort.

reading our algorithms. What separates pseudocode from "real" code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of "real" code. Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

We present our pseudocode for insertion sort as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1..n]$ containing a sequence of length $n$ that is to be sorted. (In the code, the number $n$ of elements in $A$ is denoted by $A.length$.) The algorithm sorts the input numbers ***in place***: it rearranges the numbers within the array $A$, with at most a constant number of them stored outside the array at any time. The input array $A$ contains the sorted output sequence when the INSERTION-SORT procedure is finished.

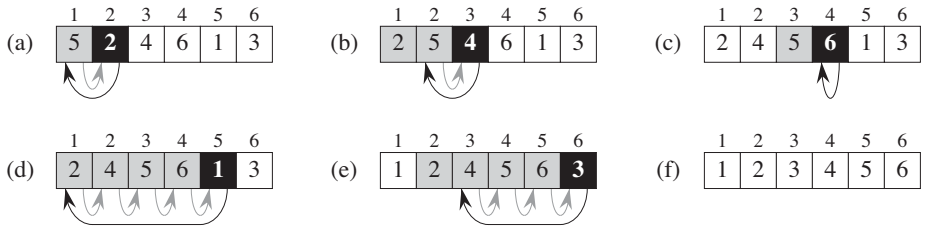



**Figure 2.2** The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

INSERTION-SORT$(A)$

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

### Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index $j$ indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by $j$, the subarray consisting of elements $A[1 . . j - 1]$ constitutes the currently sorted hand, and the remaining subarray $A[j + 1 . . n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 . . j - 1]$ are the elements *originally* in positions 1 through $j - 1$, but now in sorted order. We state these properties of $A[1 . . j - 1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 . . j - 1]$ consists of the elements originally in $A[1 . . j - 1]$, but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

**Initialization:** It is true prior to the first iteration of the loop.

**Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.

**Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. (Of course, we are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property is perhaps the most important one, since we are using the loop invariant to show correctness. Typically, we use the loop invariant along with the condition that caused the loop to terminate. The termination property differs from how we usually use mathematical induction, in which we apply the inductive step infinitely; here, we stop the "induction" when the loop terminates.

Let us see how these properties hold for insertion sort.

**Initialization:** We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \,..\, j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

**Maintenance:** Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \,..\, j]$ then consists of the elements originally in $A[1 \,..\, j]$, but in sorted order. Incrementing $j$ for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. At this point, however,

[1] When the loop is a **for** loop, the moment at which we check the loop invariant just prior to the first iteration is immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable $j$ but before the first test of whether $j \le A.length$.

we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

**Termination:** Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases $j$ by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for $j$ in the wording of loop invariant, we have that the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order. Observing that the subarray $A[1 \,..\, n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

### Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if-else** statements[2] as well. Using indentation instead of conventional indicators of block structure, such as **begin** and **end** statements, greatly reduces clutter while preserving, or even enhancing, clarity.[3]
- The looping constructs **while**, **for**, and **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.[4] In this book, the loop counter retains its value after exiting the loop, unlike some situations that arise in C++, Java, and Pascal. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound. We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $j = 2$ **to** $A.length$, and so when this loop terminates, $j = A.length + 1$ (or, equivalently, $j = n + 1$, since $n = A.length$). We use the keyword **to** when a **for** loop increments its loop

---

[2] In an **if-else** statement, we indent **else** at the same level as its matching **if**. Although we omit the keyword **then**, we occasionally refer to the portion executed when the test following **if** is true as a ***then clause***. For multiway tests, we use **elseif** for tests after the first one.

[3] Each pseudocode procedure in this book appears on one page so that you will not have to discern levels of indentation in code that is split across pages.

[4] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat-until** loops, and its **for** loops operate a little differently from the **for** loops in this book.

counter in each iteration, and we use the keyword **downto** when a **for** loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.

- The symbol "//" indicates that the remainder of the line is a comment.
- A multiple assignment of the form $i = j = e$ assigns to both variables $i$ and $j$ the value of expression $e$; it should be treated as equivalent to the assignment $j = e$ followed by the assignment $i = j$.
- Variables (such as $i$, $j$, and *key*) are local to the given procedure. We shall not use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the $i$th element of the array $A$. The notation ".." is used to indicate a range of values within an array. Thus, $A[1..j]$ indicates the subarray of $A$ consisting of the $j$ elements $A[1], A[2], \ldots, A[j]$.
- We typically organize compound data into ***objects***, which are composed of ***attributes***. We access a particular attribute using the syntax found in many object-oriented programming languages: the object name, followed by a dot, followed by the attribute name. For example, we treat an array as an object with the attribute *length* indicating how many elements it contains. To specify the number of elements in an array $A$, we write $A.length$.

  We treat a variable representing an array or object as a pointer to the data representing the array or object. For all attributes $f$ of an object $x$, setting $y = x$ causes $y.f$ to equal $x.f$. Moreover, if we now set $x.f = 3$, then afterward not only does $x.f$ equal 3, but $y.f$ equals 3 as well. In other words, $x$ and $y$ point to the same object after the assignment $y = x$.

  Our attribute notation can "cascade." For example, suppose that the attribute $f$ is itself a pointer to some type of object that has an attribute $g$. Then the notation $x.f.g$ is implicitly parenthesized as $(x.f).g$. In other words, if we had assigned $y = x.f$, then $x.f.g$ is the same as $y.g$.

  Sometimes, a pointer will refer to no object at all. In this case, we give it the special value NIL.
- We pass parameters to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's attributes are not. For example, if $x$ is a parameter of a called procedure, the assignment $x = y$ within the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible. Similarly, arrays are passed by pointer, so that

a pointer to the array is passed, rather than the entire array, and changes to individual array elements are visible to the calling procedure.

- A **return** statement immediately transfers control back to the point of call in the calling procedure. Most **return** statements also take a value to pass back to the caller. Our pseudocode differs from many programming languages in that we allow multiple values to be returned in a single **return** statement.
- The boolean operators "and" and "or" are ***short circuiting***. That is, when we evaluate the expression "$x$ and $y$" we first evaluate $x$. If $x$ evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and so we do not evaluate $y$. If, on the other hand, $x$ evaluates to TRUE, we must evaluate $y$ to determine the value of the entire expression. Similarly, in the expression "$x$ or $y$" we evaluate the expression $y$ only if $x$ evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $x.f = y$" without worrying about what happens when we try to evaluate $x.f$ when $x$ is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called. The calling procedure is responsible for handling the error, and so we do not specify what action to take.

## Exercises

### *2.1-1*

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

### *2.1-2*

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

### *2.1-3*

Consider the ***searching problem***:

**Input:** A sequence of $n$ numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value $v$.

**Output:** An index $i$ such that $v = A[i]$ or the special value NIL if $v$ does not appear in $A$.

Write pseudocode for ***linear search***, which scans through the sequence, looking for $v$. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

### *2.1-4*

Consider the problem of adding two $n$-bit binary integers, stored in two $n$-element arrays $A$ and $B$. The sum of the two integers should be stored in binary form in

an $(n + 1)$-element array $C$. State the problem formally and write pseudocode for adding the two integers.

## 2.2 Analyzing algorithms

***Analyzing*** an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine (RAM)*** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size $n$, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of $n$, enabling us to index the individual input elements, and we restrict $c$ to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no; it takes several instructions to compute $x^y$ when $x$ and $y$ are real numbers. In restricted situations, however, exponentiation is a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by $k$ positions to the left. In most computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2, so that shifting the bits by $k$ positions to the left is equivalent to multiplication by $2^k$. Therefore, such computers can compute $2^k$ in one constant-time instruction by shifting the integer 1 by $k$ positions to the left, as long as $k$ is no more than the number of bits in a computer word. We will endeavor to avoid such gray areas in the RAM model, but we will treat computation of $2^k$ as a constant-time operation when $k$ is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory. Several computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. A handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book will not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Even though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way that is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.

### Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the array size $n$ for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the $i$th line takes time $c_i$, where $c_i$ is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[5]

In the following discussion, our expression for the running time of INSERTION-SORT will evolve from a messy formula that uses all the statement costs $c_i$ to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n = A.length$, we let $t_j$ denote the number of times the **while** loop test in line 5 is executed for that value of $j$. When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

---

[5] There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, later in this book we might say "sort the points by $x$-coordinate," which, as we shall see, takes more than a constant amount of time. Also, note that a statement that calls a subroutine takes constant time, though the subroutine, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

| INSERTION-SORT$(A)$ | *cost* | *times* |
|---|---|---|
| 1 **for** $j = 2$ **to** $A.length$ | $c_1$ | $n$ |
| 2 $key = A[j]$ | $c_2$ | $n-1$ |
| 3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$. | 0 | $n-1$ |
| 4 $i = j-1$ | $c_4$ | $n-1$ |
| 5 **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 $A[i+1] = A[i]$ | $c_6$ | $\sum_{j=2}^{n}(t_j-1)$ |
| 7 $i = i-1$ | $c_7$ | $\sum_{j=2}^{n}(t_j-1)$ |
| 8 $A[i+1] = key$ | $c_8$ | $n-1$ |

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes $c_i$ steps to execute and executes $n$ times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of $n$ values, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \; . \end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \le key$ in line 5 when $i$ has its initial value of $j-1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; . \end{aligned}$$

We can express this running time as $an + b$ for *constants* $a$ and $b$ that depend on the statement costs $c_i$; it is thus a ***linear function*** of $n$.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

---

[6] This characteristic does not necessarily hold for a resource such as memory. A statement that references $m$ words of memory and is executed $n$ times does not necessarily reference $mn$ distinct words of memory.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(see Appendix A for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants $a$, $b$, and $c$ that again depend on the statement costs $c_i$; it is thus a ***quadratic function*** of $n$.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

### Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size $n$. We give three reasons for this orientation.

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.

- The "average case" is often roughly as bad as the worst case. Suppose that we randomly choose $n$ numbers and apply insertion sort. How long does it take to determine where in subarray $A[1 \,..\, j-1]$ to insert element $A[j]$? On average, half the elements in $A[1 \,..\, j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1 \,..\, j-1]$, and so $t_j$ is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the ***average-case*** running time of an algorithm; we shall see the technique of ***probabilistic analysis*** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a ***randomized algorithm***, which makes random choices, to allow a probabilistic analysis and yield an ***expected*** running time. We explore randomized algorithms more in Chapter 5 and in several other subsequent chapters.

### Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure. First, we ignored the actual cost of each statement, using the constants $c_i$ to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants $a$, $b$, and $c$ that depend on the statement costs $c_i$. We thus ignored not only the actual statement costs, but also the abstract costs $c_i$.

We shall now make one more simplifying abstraction: it is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., $an^2$), since the lower-order terms are relatively insignificant for large values of $n$. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of $n^2$ from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of $n$-squared"). We shall use $\Theta$-notation informally in this chapter, and we will define it precisely in Chapter 3.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower

order of growth. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.

### Exercises

***2.2-1***
Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of $\Theta$-notation.

***2.2-2***
Consider sorting $n$ numbers stored in array $A$ by first finding the smallest element of $A$ and exchanging it with the element in $A[1]$. Then find the second smallest element of $A$, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of $A$. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all $n$ elements? Give the best-case and worst-case running times of selection sort in $\Theta$-notation.

***2.2-3***
Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in $\Theta$-notation? Justify your answers.

***2.2-4***
How can we modify almost any algorithm to have a good best-case running time?

## 2.3 Designing algorithms

We can choose from a wide range of algorithm design techniques. For insertion sort, we used an ***incremental*** approach: having sorted the subarray $A[1 \mathinner{..} j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1 \mathinner{..} j]$.

In this section, we examine an alternative design approach, known as "divide-and-conquer," which we shall explore in more detail in Chapter 4. We'll use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that we will see in Chapter 4.

### 2.3.1 The divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

**Combine** the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

**Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE$(A, p, q, r)$, where $A$ is an array and $p$, $q$, and $r$ are indices into the array such that $p \le q < r$. The procedure assumes that the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p\,..\,r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto

the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most $n$ basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use $\infty$ as the sentinel value, so that whenever a card with $\infty$ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length $n_1$ of the subarray $A[p \,..\, q]$, and line 2 computes the length $n_2$ of the subarray $A[q + 1 \,..\, r]$. We create arrays $L$ and $R$ ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray $A[p \,..\, q]$ into $L[1 \,..\, n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q + 1 \,..\, r]$ into $R[1 \,..\, n_2]$. Lines 8–9 put the sentinels at the ends of the arrays $L$ and $R$. Lines 10–17, illus-

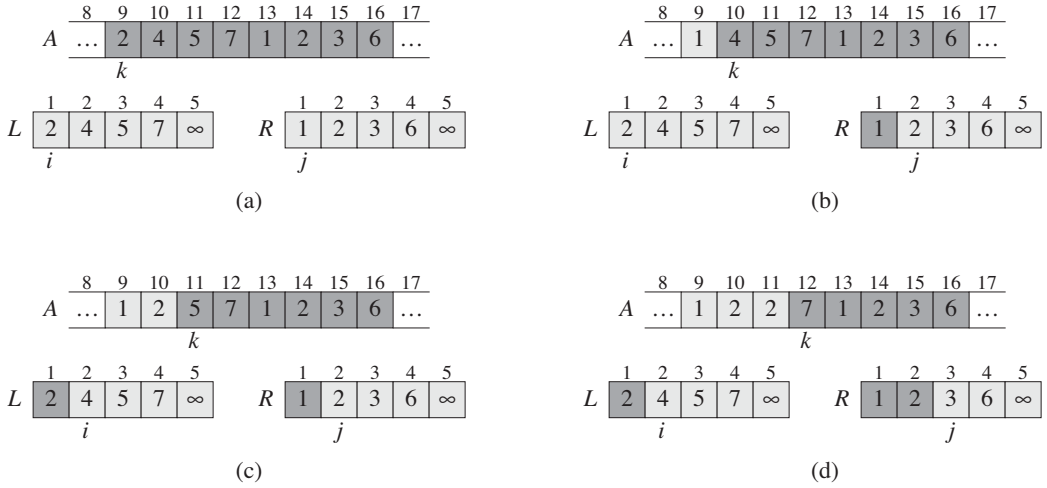



**Figure 2.3** The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6\rangle$. After copying and inserting sentinels, the array $L$ contains $\langle 2, 4, 5, 7, \infty\rangle$, and the array $R$ contains $\langle 1, 2, 3, 6, \infty\rangle$. Lightly shaded positions in $A$ contain their final values, and lightly shaded positions in $L$ and $R$ contain values that have yet to be copied back into $A$. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in $A$ contain values that will be copied over, and heavily shaded positions in $L$ and $R$ contain values that have already been copied back into $A$. **(a)–(h)** The arrays $A$, $L$, and $R$, and their respective indices $k$, $i$, and $j$ prior to each iteration of the loop of lines 12–17.

trated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

**Initialization:** Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of $L$ and $R$, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

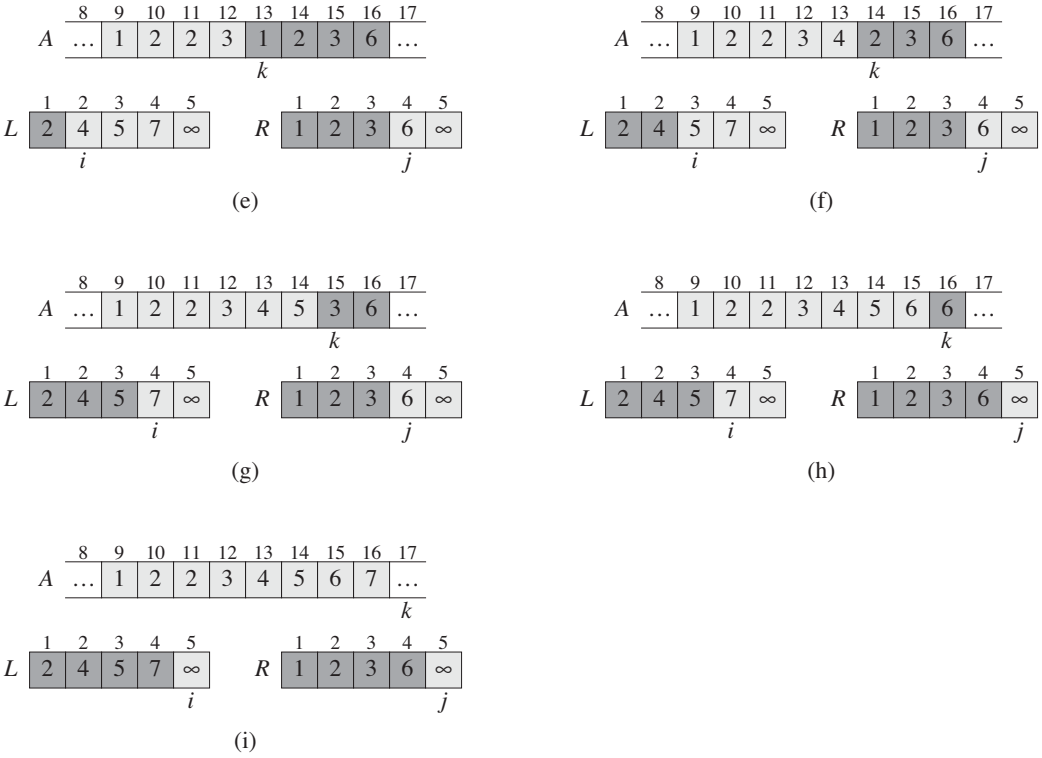



**Figure 2.3, continued** **(i)** The arrays and indices at termination. At this point, the subarray in $A[9\,..\,16]$ is sorted, and the two sentinels in $L$ and $R$ are the only two elements in these arrays that have not been copied into $A$.

**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into $A$. Because $A[p\,..\,k-1]$ contains the $k-p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p\,..\,k]$ will contain the $k-p+1$ smallest elements. Incrementing $k$ (in the **for** loop update) and $i$ (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

**Termination:** At termination, $k = r+1$. By the loop invariant, the subarray $A[p\,..\,k-1]$, which is $A[p\,..\,r]$, contains the $k-p = r-p+1$ smallest elements of $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$, in sorted order. The arrays $L$ and $R$ together contain $n_1+n_2+2 = r-p+3$ elements. All but the two largest have been copied back into $A$, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of lines 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time,[7] and there are $n$ iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT$(A, p, r)$ sorts the elements in the subarray $A[p..r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index $q$ that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.[8]

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, A.length)$, where once again $A.length = n$. Figure 2.4 illustrates the operation of the procedure bottom-up when $n$ is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length $n$.

### 2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size $n$ in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

---

[7] We shall see in Chapter 3 how to formally interpret equations containing $\Theta$-notation.

[8] The expression $\lceil x \rceil$ denotes the least integer greater than or equal to $x$, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to $x$. These notations are defined in Chapter 3. The easiest way to verify that setting $q$ to $\lfloor (p + r)/2 \rfloor$ yields subarrays $A[p..q]$ and $A[q+1..r]$ of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, is to examine the four cases that arise depending on whether each of $p$ and $r$ is odd or even.

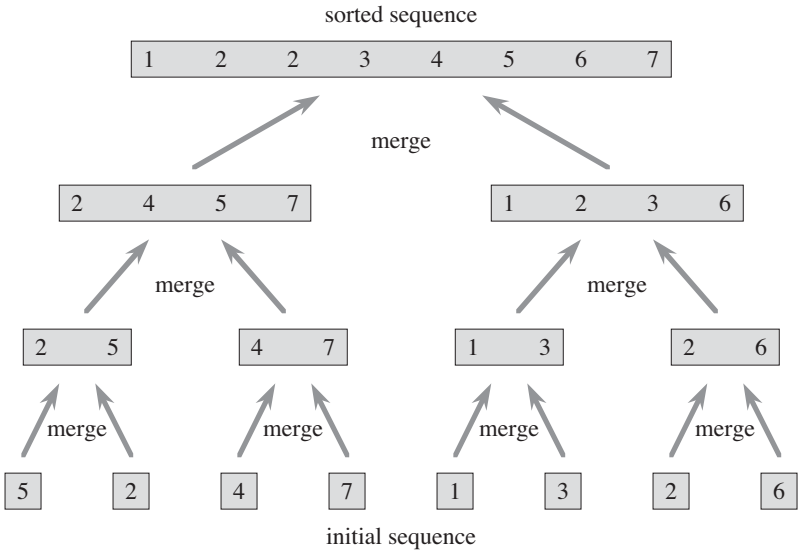



**Figure 2.4** The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size $n$. If the problem size is small enough, say $n \leq c$ for some constant $c$, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields $a$ subproblems, each of which is $1/b$ the size of the original. (For merge sort, both $a$ and $b$ are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) It takes time $T(n/b)$ to solve one subproblem of size $n/b$, and so it takes time $aT(n/b)$ to solve $a$ of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \ . \end{cases}$$

In Chapter 4, we shall see how to solve common recurrences of this form.

### Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that

the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on $n$ numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

**Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

**Combine:** We have already noted that the MERGE procedure on an $n$-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of $n$, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{2.1}$$

In Chapter 4, we shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to the recurrence (2.1) is $T(n) = \Theta(n \lg n)$. Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases} \tag{2.2}$$

where the constant $c$ represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.[9]

[9] It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting $c$ be the larger of these times and understanding that our recurrence gives an upper bound on the running time, or by letting $c$ be the lesser of these times and understanding that our recurrence gives a lower bound on the running time. Both bounds are on the order of $n \lg n$ and, taken together, give a $\Theta(n \lg n)$ running time.

Figure 2.5 shows how we can solve recurrence (2.2). For convenience, we assume that $n$ is an exact power of 2. Part (a) of the figure shows $T(n)$, which we expand in part (b) into an equivalent tree representing the recurrence. The $cn$ term is the root (the cost incurred at the top level of recursion), and the two subtrees of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost incurred at each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of $c$. Part (d) shows the resulting ***recursion tree***.

Next, we add the costs across each level of the tree. The top level has total cost $cn$, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, and so on. In general, the level $i$ below the top has $2^i$ nodes, each contributing a cost of $c(n/2^i)$, so that the $i$th level below the top has total cost $2^i\, c(n/2^i) = cn$. The bottom level has $n$ nodes, each contributing a cost of $c$, for a total cost of $cn$.

The total number of levels of the recursion tree in Figure 2.5 is $\lg n + 1$, where $n$ is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim. The base case occurs when $n = 1$, in which case the tree has only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree with $2^i$ leaves is $\lg 2^i + 1 = i + 1$ (since for any value of $i$, we have that $\lg 2^i = i$). Because we are assuming that the input size is a power of 2, the next input size to consider is $2^{i+1}$. A tree with $n = 2^{i+1}$ leaves has one more level than a tree with $2^i$ leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels, each costing $cn$, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant $c$ gives the desired result of $\Theta(n \lg n)$.

## Exercises

### *2.3-1*

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

### *2.3-2*

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array $L$ or $R$ has had all its elements copied back to $A$ and then copying the remainder of the other array back into $A$.

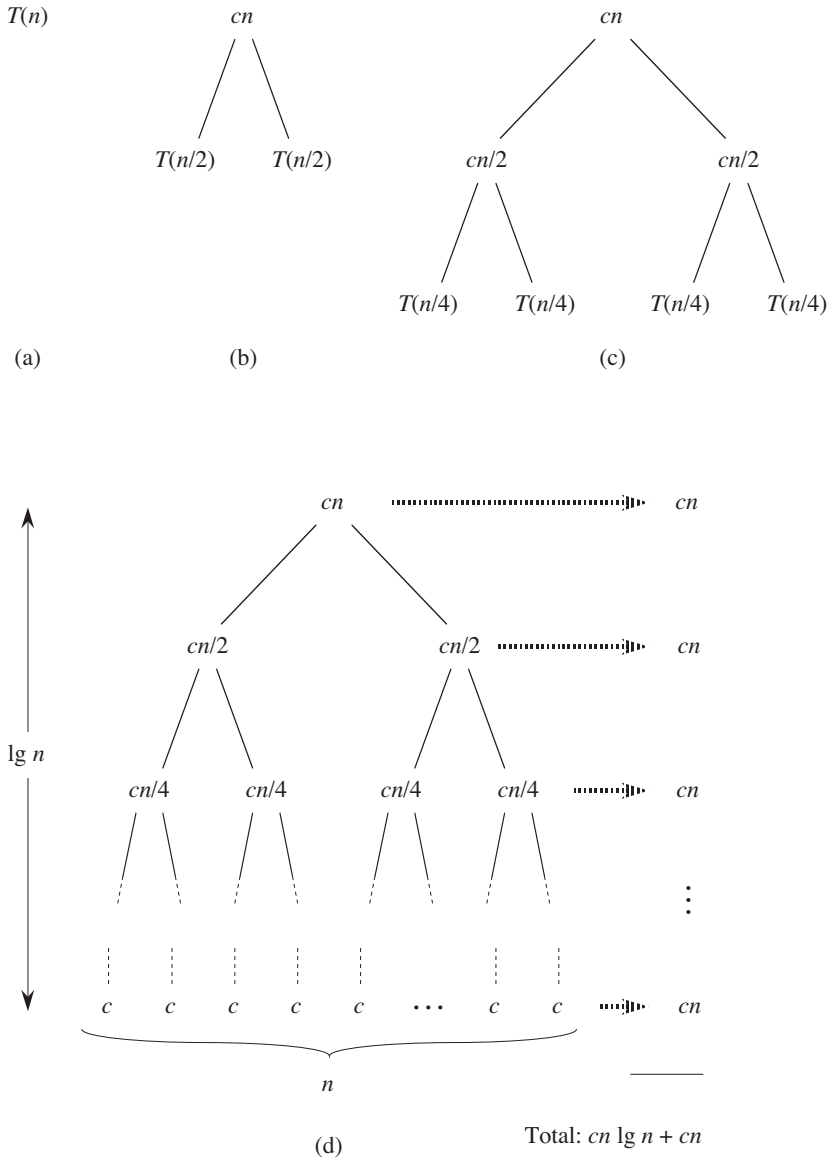



**Figure 2.5** How to construct a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of $cn$. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

***2.3-3***
Use mathematical induction to show that when $n$ is an exact power of 2, the solution of the recurrence

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

is $T(n) = n \lg n$.

***2.3-4***
We can express insertion sort as a recursive procedure as follows. In order to sort $A[1\,..\,n]$, we recursively sort $A[1\,..\,n-1]$ and then insert $A[n]$ into the sorted array $A[1\,..\,n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

***2.3-5***
Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence $A$ is sorted, we can check the midpoint of the sequence against $v$ and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

***2.3-6***
Observe that the **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1\,..\,j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

***2.3-7*** ★
Describe a $\Theta(n \lg n)$-time algorithm that, given a set $S$ of $n$ integers and another integer $x$, determines whether or not there exist two elements in $S$ whose sum is exactly $x$.

## Problems

***2-1 Insertion sort on small arrays in merge sort***
Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when

subproblems become sufficiently small. Consider a modification to merge sort in which $n/k$ sublists of length $k$ are sorted using insertion sort and then merged using the standard merging mechanism, where $k$ is a value to be determined.

***a.*** Show that insertion sort can sort the $n/k$ sublists, each of length $k$, in $\Theta(nk)$ worst-case time.

***b.*** Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

***c.*** Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of $k$ as a function of $n$ for which the modified algorithm has the same running time as standard merge sort, in terms of $\Theta$-notation?

***d.*** How should we choose $k$ in practice?

### *2-2 Correctness of bubblesort*

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

***a.*** Let $A'$ denote the output of BUBBLESORT$(A)$. To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n] , \tag{2.3}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

***b.*** State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

***c.*** Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

***d.*** What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

### *2-3 Correctness of Horner's rule*

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots)) \,, \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for $x$:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

***a.*** In terms of $\Theta$-notation, what is the running time of this code fragment for Horner's rule?

***b.*** Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

***c.*** Consider the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3,
>
> $$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \,.$$
>
> Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

***d.*** Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

### *2-4 Inversions*

Let $A[1..n]$ be an array of $n$ distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair $(i, j)$ is called an ***inversion*** of $A$.

***a.*** List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

***b.*** What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

***c.*** What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

***d.*** Give an algorithm that determines the number of inversions in any permutation on $n$ elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

## Chapter notes

In 1968, Knuth published the first of three volumes with the general title *The Art of Computer Programming* [209, 210, 211]. The first volume ushered in the modern study of computer algorithms with a focus on the analysis of running time, and the full series remains an engaging and worthwhile reference for many of the topics presented here. According to Knuth, the word "algorithm" is derived from the name "al-Khowârizmî," a ninth-century Persian mathematician.

Aho, Hopcroft, and Ullman [5] advocated the asymptotic analysis of algorithms—using notations that Chapter 3 introduces, including $\Theta$-notation—as a means of comparing relative performance. They also popularized the use of recurrence relations to describe the running times of recursive algorithms.

Knuth [211] provides an encyclopedic treatment of many sorting algorithms. His comparison of sorting algorithms (page 381) includes exact step-counting analyses, like the one we performed here for insertion sort. Knuth's discussion of insertion sort encompasses several variations of the algorithm. The most important of these is Shell's sort, introduced by D. L. Shell, which uses insertion sort on periodic subsequences of the input to produce a faster sorting algorithm.

Merge sort is also described by Knuth. He mentions that a mechanical collator capable of merging two decks of punched cards in a single pass was invented in 1938. J. von Neumann, one of the pioneers of computer science, apparently wrote a program for merge sort on the EDVAC computer in 1945.

The early history of proving programs correct is described by Gries [153], who credits P. Naur with the first article in this field. Gries attributes loop invariants to R. W. Floyd. The textbook by Mitchell [256] describes more recent progress in proving programs correct.

# 3 Growth of Functions

The order of growth of the running time of an algorithm, defined in Chapter 2, gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Once the input size $n$ becomes large enough, merge sort, with its $\Theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\Theta(n^2)$. Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort in Chapter 2, the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

This chapter gives several standard methods for simplifying the asymptotic analysis of algorithms. The next section begins by defining several types of "asymptotic notation," of which we have already seen an example in $\Theta$-notation. We then present several notational conventions used throughout this book, and finally we review the behavior of functions that commonly arise in the analysis of algorithms.

## 3.1 Asymptotic notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$. Such notations are convenient for describing the worst-case running-time function $T(n)$, which usually is defined only on integer input sizes. We sometimes find it convenient, however, to *abuse* asymptotic notation in a va-

riety of ways. For example, we might extend the notation to the domain of real numbers or, alternatively, restrict it to a subset of the natural numbers. We should make sure, however, to understand the precise meaning of the notation so that when we abuse, we do not *misuse* it. This section defines the basic asymptotic notations and also introduces some common abuses.

### Asymptotic notation, functions, and running times

We will use asymptotic notation primarily to describe the running times of algorithms, as when we wrote that insertion sort's worst-case running time is $\Theta(n^2)$. Asymptotic notation actually applies to functions, however. Recall that we characterized insertion sort's worst-case running time as $an^2 + bn + c$, for some constants $a$, $b$, and $c$. By writing that insertion sort's running time is $\Theta(n^2)$, we abstracted away some details of this function. Because asymptotic notation applies to functions, what we were writing as $\Theta(n^2)$ was the function $an^2 + bn + c$, which in that case happened to characterize the worst-case running time of insertion sort.

In this book, the functions to which we apply asymptotic notation will usually characterize the running times of algorithms. But asymptotic notation can apply to functions that characterize some other aspect of algorithms (the amount of space they use, for example), or even to functions that have nothing whatsoever to do with algorithms.

Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand *which* running time we mean. Sometimes we are interested in the worst-case running time. Often, however, we wish to characterize the running time no matter what the input. In other words, we often wish to make a blanket statement that covers all inputs, not just the worst case. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.

### $\Theta$-notation

In Chapter 2, we found that the worst-case running time of insertion sort is $T(n) = \Theta(n^2)$. Let us define what this notation means. For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}\ .^{1}$$

[1] Within set notation, a colon means "such that."

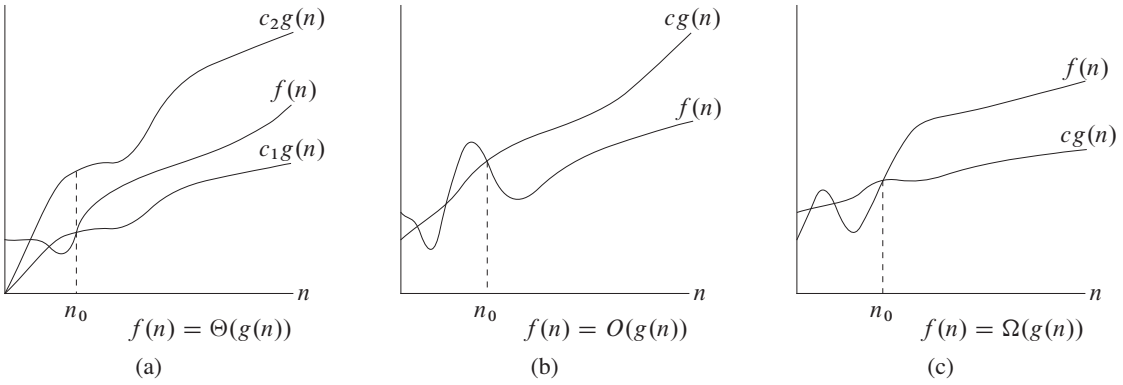



**Figure 3.1** Graphic examples of the $\Theta$, $O$, and $\Omega$ notations. In each part, the value of $n_0$ shown is the minimum possible value; any greater value would also work. **(a)** $\Theta$-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$, and $c_2$ such that at and to the right of $n_0$, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** $O$-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants $n_0$ and $c$ such that at and to the right of $n_0$, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** $\Omega$-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants $n_0$ and $c$ such that at and to the right of $n_0$, the value of $f(n)$ always lies on or above $cg(n)$.

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants $c_1$ and $c_2$ such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large $n$. Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion. You might be confused because we abuse equality in this way, but we shall see later in this section that doing so has its advantages.

Figure 3.1(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) = \Theta(g(n))$. For all values of $n$ at and to the right of $n_0$, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be ***asymptotically nonnegative***, that is, that $f(n)$ be nonnegative whenever $n$ is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all sufficiently large $n$.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within $\Theta$-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

In Chapter 2, we introduced an informal notion of $\Theta$-notation that amounted to throwing away lower-order terms and ignoring the leading coefficient of the highest-order term. Let us briefly justify this intuition by using the formal definition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants $c_1$, $c_2$, and $n_0$ such that

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for all $n \geq n_0$. Dividing by $n^2$ yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 .$$

We can make the right-hand inequality hold for any value of $n \geq 1$ by choosing any constant $c_2 \geq 1/2$. Likewise, we can make the left-hand inequality hold for any value of $n \geq 7$ by choosing any constant $c_1 \leq 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.

We can also use the formal definition to verify that $6n^3 \neq \Theta(n^2)$. Suppose for the purpose of contradiction that $c_2$ and $n_0$ exist such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$. But then dividing by $n^2$ yields $n \leq c_2/6$, which cannot possibly hold for arbitrarily large $n$, since $c_2$ is constant.

Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large $n$. When $n$ is large, even a tiny fraction of the highest-order term suffices to dominate the lower-order terms. Thus, setting $c_1$ to a value that is slightly smaller than the coefficient of the highest-order term and setting $c_2$ to a value that is slightly larger permits the inequalities in the definition of $\Theta$-notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes $c_1$ and $c_2$ by a constant factor equal to the coefficient.

As an example, consider any quadratic function $f(n) = an^2 + bn + c$, where $a$, $b$, and $c$ are constants and $a > 0$. Throwing away the lower-order terms and ignoring the constant yields $f(n) = \Theta(n^2)$. Formally, to show the same thing, we take the constants $c_1 = a/4$, $c_2 = 7a/4$, and $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$. You may verify that $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ for all $n \geq n_0$. In general, for any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$, where the $a_i$ are constants and $a_d > 0$, we have $p(n) = \Theta(n^d)$ (see Problem 3-1).

Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$, or $\Theta(1)$. This latter notation is a minor abuse, however, because the

expression does not indicate what variable is tending to infinity.[2] We shall often use the notation $\Theta(1)$ to mean either a constant or a constant function with respect to some variable.

### $O$-notation

The $\Theta$-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use $O$-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of $g$ of $n$" or sometimes just "oh of $g$ of $n$") the set of functions

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

We use $O$-notation to give an upper bound on a function, to within a constant factor. Figure 3.1(b) shows the intuition behind $O$-notation. For all values $n$ at and to the right of $n_0$, the value of the function $f(n)$ is on or below $cg(n)$.

We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since $\Theta$-notation is a stronger notion than $O$-notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\Theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$. What may be more surprising is that when $a > 0$, any *linear* function $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = \max(1, -b/a)$.

If you have seen $O$-notation before, you might find it strange that we should write, for example, $n = O(n^2)$. In the literature, we sometimes find $O$-notation informally describing asymptotically tight bounds, that is, what we have defined using $\Theta$-notation. In this book, however, when we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is. Distinguishing asymptotic upper bounds from asymptotically tight bounds is standard in the algorithms literature.

Using $O$-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm from Chapter 2 immediately yields an $O(n^2)$ upper bound on the worst-case running time: the cost of each iteration of the inner loop is bounded from above by $O(1)$ (constant), the indices $i$

---

[2] The real problem is that our ordinary notation for functions does not distinguish functions from values. In $\lambda$-calculus, the parameters to a function are clearly specified: the function $n^2$ could be written as $\lambda n.n^2$, or even $\lambda r.r^2$. Adopting a more rigorous notation, however, would complicate algebraic manipulations, and so we choose to tolerate the abuse.

and $j$ are both at most $n$, and the inner loop is executed at most once for each of the $n^2$ pairs of values for $i$ and $j$.

Since $O$-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input—the blanket statement we discussed earlier. Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input. The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on *every* input. For example, we saw in Chapter 2 that when the input is already sorted, insertion sort runs in $\Theta(n)$ time.

Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given $n$, the actual running time varies, depending on the particular input of size $n$. When we say "the running time is $O(n^2)$," we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of $n$, no matter what particular input of size $n$ is chosen, the running time on that input is bounded from above by the value $f(n)$. Equivalently, we mean that the worst-case running time is $O(n^2)$.

### $\Omega$-notation

Just as $O$-notation provides an asymptotic *upper* bound on a function, $\Omega$-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of $g$ of $n$" or sometimes just "omega of $g$ of $n$") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

Figure 3.1(c) shows the intuition behind $\Omega$-notation. For all values $n$ at or to the right of $n_0$, the value of $f(n)$ is on or above $cg(n)$.

From the definitions of the asymptotic notations we have seen thus far, it is easy to prove the following important theorem (see Exercise 3.1-5).

***Theorem 3.1***

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

As an example of the application of this theorem, our proof that $an^2 + bn + c = \Theta(n^2)$ for any constants $a$, $b$, and $c$, where $a > 0$, immediately implies that $an^2 + bn + c = \Omega(n^2)$ and $an^2 + bn + c = O(n^2)$. In practice, rather than using Theorem 3.1 to obtain asymptotic upper and lower bounds from asymptotically tight bounds, as we did for this example, we usually use it to prove asymptotically tight bounds from asymptotic upper and lower bounds.

When we say that the *running time* (no modifier) of an algorithm is $\Omega(g(n))$, we mean that *no matter what particular input of size n is chosen for each value of n*, the running time on that input is at least a constant times $g(n)$, for sufficiently large $n$. Equivalently, we are giving a lower bound on the best-case running time of an algorithm. For example, the best-case running time of insertion sort is $\Omega(n)$, which implies that the running time of insertion sort is $\Omega(n)$.

The running time of insertion sort therefore belongs to both $\Omega(n)$ and $O(n^2)$, since it falls anywhere between a linear function of $n$ and a quadratic function of $n$. Moreover, these bounds are asymptotically as tight as possible: for instance, the running time of insertion sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted). It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$, since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

### Asymptotic notation in equations and inequalities

We have already seen how asymptotic notation can be used within mathematical formulas. For example, in introducing $O$-notation, we wrote "$n = O(n^2)$." We might also write $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. How do we interpret such formulas?

When the asymptotic notation stands alone (that is, not within a larger formula) on the right-hand side of an equation (or inequality), as in $n = O(n^2)$, we have already defined the equal sign to mean set membership: $n \in O(n^2)$. In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name. For example, the formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n)$ is some function in the set $\Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, in Chapter 2 we expressed the worst-case running time of merge sort as the recurrence

$$T(n) = 2T(n/2) + \Theta(n) .$$

If we are interested only in the asymptotic behavior of $T(n)$, there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term $\Theta(n)$.

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears. For example, in the expression

$$\sum_{i=1}^{n} O(i) ,$$

there is only a single anonymous function (a function of $i$). This expression is thus *not* the same as $O(1) + O(2) + \cdots + O(n)$, which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

$$2n^2 + \Theta(n) = \Theta(n^2) .$$

We interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.* Thus, our example means that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all $n$. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

We can chain together a number of such relationships, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

We can interpret each equation separately by the rules above. The first equation says that there is *some* function $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$ for all $n$. The second equation says that for *any* function $g(n) \in \Theta(n)$ (such as the $f(n)$ just mentioned), there is *some* function $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$ for all $n$. Note that this interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively gives us.

### *o*-notation

The asymptotic upper bound provided by $O$-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use $o$-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of $g$ of $n$") as the set

$$o(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The definitions of $O$-notation and $o$-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in $o$-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as $n$ approaches infinity; that is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \,. \tag{3.1}$$

Some authors use this limit as a definition of the $o$-notation; the definition in this book also restricts the anonymous functions to be asymptotically nonnegative.

### $\omega$-notation

By analogy, $\omega$-notation is to $\Omega$-notation as $o$-notation is to $O$-notation. We use $\omega$-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$ .

Formally, however, we define $\omega(g(n))$ ("little-omega of $g$ of $n$") as the set

$$\begin{aligned}\omega(g(n)) = \{f(n) :\ & \text{for any positive constant } c > 0, \text{ there exists a constant} \\ & n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} \,.\end{aligned}$$

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

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

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as $n$ approaches infinity.

### Comparing functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

**Transitivity:**

$$\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)) \,, \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)) \,, \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)) \,, \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)) \,, \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)) \,.
\end{aligned}$$

**Reflexivity:**

$$\begin{aligned}
f(n) &= \Theta(f(n)) \,, \\
f(n) &= O(f(n)) \,, \\
f(n) &= \Omega(f(n)) \,.
\end{aligned}$$

**Symmetry:**

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

**Transpose symmetry:**

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions $f$ and $g$ and the comparison of two real numbers $a$ and $b$:

$$f(n) = O(g(n)) \quad \text{is like} \quad a \leq b ,$$
$$f(n) = \Omega(g(n)) \quad \text{is like} \quad a \geq b ,$$
$$f(n) = \Theta(g(n)) \quad \text{is like} \quad a = b ,$$
$$f(n) = o(g(n)) \quad \text{is like} \quad a < b ,$$
$$f(n) = \omega(g(n)) \quad \text{is like} \quad a > b .$$

We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

One property of real numbers, however, does not carry over to asymptotic notation:

**Trichotomy:** For any two real numbers $a$ and $b$, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions $f(n)$ and $g(n)$, it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions $n$ and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

## Exercises

### *3.1-1*

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of $\Theta$-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

### *3.1-2*

Show that for any real constants $a$ and $b$, where $b > 0$,

$$(n + a)^b = \Theta(n^b) . \tag{3.2}$$

***3.1-3***
Explain why the statement, "The running time of algorithm $A$ is at least $O(n^2)$," is meaningless.

***3.1-4***
Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

***3.1-5***
Prove Theorem 3.1.

***3.1-6***
Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

***3.1-7***
Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

***3.1-8***
We can extend our notation to the case of two parameters $n$ and $m$ that can go to infinity independently at different rates. For a given function $g(n,m)$, we denote by $O(g(n,m))$ the set of functions

$$O(g(n,m)) = \{f(n,m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \le f(n,m) \le cg(n,m) \text{ for all } n \ge n_0 \text{ or } m \ge m_0\}\ .$$

Give corresponding definitions for $\Omega(g(n,m))$ and $\Theta(g(n,m))$.

## 3.2 Standard notations and common functions

This section reviews some standard mathematical functions and notations and explores the relationships among them. It also illustrates the use of the asymptotic notations.

### Monotonicity

A function $f(n)$ is ***monotonically increasing*** if $m \le n$ implies $f(m) \le f(n)$. Similarly, it is ***monotonically decreasing*** if $m \le n$ implies $f(m) \ge f(n)$. A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

### Floors and ceilings

For any real number $x$, we denote the greatest integer less than or equal to $x$ by $\lfloor x \rfloor$ (read "the floor of $x$") and the least integer greater than or equal to $x$ by $\lceil x \rceil$ (read "the ceiling of $x$"). For all real $x$,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 . \tag{3.3}$$

For any integer $n$,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n ,$$

and for any real number $x \geq 0$ and integers $a, b > 0$,

$$\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil , \tag{3.4}$$

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor , \tag{3.5}$$

$$\left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b} , \tag{3.6}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \geq \frac{a - (b-1)}{b} . \tag{3.7}$$

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

### Modular arithmetic

For any integer $a$ and any positive integer $n$, the value $a \bmod n$ is the ***remainder*** (or ***residue***) of the quotient $a/n$:

$$a \bmod n = a - n \lfloor a/n \rfloor . \tag{3.8}$$

It follows that

$$0 \leq a \bmod n < n . \tag{3.9}$$

Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(a \bmod n) = (b \bmod n)$, we write $a \equiv b \pmod{n}$ and say that $a$ is ***equivalent*** to $b$, modulo $n$. In other words, $a \equiv b \pmod{n}$ if $a$ and $b$ have the same remainder when divided by $n$. Equivalently, $a \equiv b \pmod{n}$ if and only if $n$ is a divisor of $b - a$. We write $a \not\equiv b \pmod{n}$ if $a$ is not equivalent to $b$, modulo $n$.

### Polynomials

Given a nonnegative integer $d$, a ***polynomial in n of degree d*** is a function $p(n)$ of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where the constants $a_0, a_1, \ldots, a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. For an asymptotically positive polynomial $p(n)$ of degree $d$, we have $p(n) = \Theta(n^d)$. For any real constant $a \geq 0$, the function $n^a$ is monotonically increasing, and for any real constant $a \leq 0$, the function $n^a$ is monotonically decreasing. We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant $k$.

### Exponentials

For all real $a > 0$, $m$, and $n$, we have the following identities:

$$\begin{aligned} a^0 &= 1 , \\ a^1 &= a , \\ a^{-1} &= 1/a , \\ (a^m)^n &= a^{mn} , \\ (a^m)^n &= (a^n)^m , \\ a^m a^n &= a^{m+n} . \end{aligned}$$

For all $n$ and $a \geq 1$, the function $a^n$ is monotonically increasing in $n$. When convenient, we shall assume $0^0 = 1$.

We can relate the rates of growth of polynomials and exponentials by the following fact. For all real constants $a$ and $b$ such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 , \tag{3.10}$$

from which we can conclude that

$$n^b = o(a^n) .$$

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

Using $e$ to denote $2.71828\ldots$, the base of the natural logarithm function, we have for all real $x$,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \tag{3.11}$$

where "!" denotes the factorial function defined later in this section. For all real $x$, we have the inequality

$$e^x \geq 1 + x \,, \tag{3.12}$$

where equality holds only when $x = 0$. When $|x| \leq 1$, we have the approximation

$$1 + x \leq e^x \leq 1 + x + x^2 \,. \tag{3.13}$$

When $x \to 0$, the approximation of $e^x$ by $1 + x$ is quite good:

$$e^x = 1 + x + \Theta(x^2) \,.$$

(In this equation, the asymptotic notation is used to describe the limiting behavior as $x \to 0$ rather than as $x \to \infty$.) We have for all $x$,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x \,. \tag{3.14}$$

### Logarithms

We shall use the following notations:

$$\begin{aligned}
\lg n &= \log_2 n \quad \text{(binary logarithm)} \,, \\
\ln n &= \log_e n \quad \text{(natural logarithm)} \,, \\
\lg^k n &= (\lg n)^k \quad \text{(exponentiation)} \,, \\
\lg \lg n &= \lg(\lg n) \quad \text{(composition)} \,.
\end{aligned}$$

An important notational convention we shall adopt is that *logarithm functions will apply only to the next term in the formula*, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg(n + k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

For all real $a > 0$, $b > 0$, $c > 0$, and $n$,

$$\begin{aligned}
a &= b^{\log_b a} \,, \\
\log_c(ab) &= \log_c a + \log_c b \,, \\
\log_b a^n &= n \log_b a \,, \\
\log_b a &= \frac{\log_c a}{\log_c b} \,, \qquad (3.15) \\
\log_b(1/a) &= -\log_b a \,, \\
\log_b a &= \frac{1}{\log_a b} \,, \\
a^{\log_b c} &= c^{\log_b a} \,, \qquad (3.16)
\end{aligned}$$

where, in each equation above, logarithm bases are not 1.

By equation (3.15), changing the base of a logarithm from one constant to another changes the value of the logarithm by only a constant factor, and so we shall often use the notation "$\lg n$" when we don't care about constant factors, such as in $O$-notation. Computer scientists find 2 to be the most natural base for logarithms because so many algorithms and data structures involve splitting a problem into two parts.

There is a simple series expansion for $\ln(1+x)$ when $|x| < 1$:

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

We also have the following inequalities for $x > -1$:

$$\frac{x}{1+x} \le \ln(1+x) \le x , \tag{3.17}$$

where equality holds only for $x = 0$.

We say that a function $f(n)$ is ***polylogarithmically bounded*** if $f(n) = O(\lg^k n)$ for some constant $k$. We can relate the growth of polynomials and polylogarithms by substituting $\lg n$ for $n$ and $2^a$ for $a$ in equation (3.10), yielding

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 .$$

From this limit, we can conclude that

$$\lg^b n = o(n^a)$$

for any constant $a > 0$. Thus, any positive polynomial function grows faster than any polylogarithmic function.

### Factorials

The notation $n!$ (read "$n$ factorial") is defined for integers $n \ge 0$ as

$$n! = \begin{cases} 1 & \text{if } n = 0 , \\ n \cdot (n-1)! & \text{if } n > 0 . \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

A weak upper bound on the factorial function is $n! \le n^n$, since each of the $n$ terms in the factorial product is at most $n$. ***Stirling's approximation***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.18}$$

where $e$ is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well. As Exercise 3.2-3 asks you to prove,

$$\begin{aligned} n! &= o(n^n) , \\ n! &= \omega(2^n) , \\ \lg(n!) &= \Theta(n \lg n) , \end{aligned} \tag{3.19}$$

where Stirling's approximation is helpful in proving equation (3.19). The following equation also holds for all $n \geq 1$:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.20}$$

where

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} . \tag{3.21}$$

### Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied $i$ times to an initial value of $n$. Formally, let $f(n)$ be a function over the reals. For nonnegative integers $i$, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

### The iterated logarithm function

We use the notation $\lg^* n$ (read "log star of $n$") to denote the iterated logarithm, defined as follows. Let $\lg^{(i)} n$ be as defined above, with $f(n) = \lg n$. Because the logarithm of a nonpositive number is undefined, $\lg^{(i)} n$ is defined only if $\lg^{(i-1)} n > 0$. Be sure to distinguish $\lg^{(i)} n$ (the logarithm function applied $i$ times in succession, starting with argument $n$) from $\lg^i n$ (the logarithm of $n$ raised to the $i$th power). Then we define the iterated logarithm function as

$$\lg^* n = \min \left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\} .$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^*(2^{65536}) &= 5 . \end{aligned}$$

Since the number of atoms in the observable universe is estimated to be about $10^{80}$, which is much less than $2^{65536}$, we rarely encounter an input size $n$ such that $\lg^* n > 5$.

### Fibonacci numbers

We define the ***Fibonacci numbers*** by the following recurrence:

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned} \tag{3.22}$$

Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence

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

Fibonacci numbers are related to the ***golden ratio*** $\phi$ and to its conjugate $\hat{\phi}$, which are the two roots of the equation

$$x^2 = x + 1 \tag{3.23}$$

and are given by the following formulas (see Exercise 3.2-6):

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots , \\ \hat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots . \end{aligned} \tag{3.24}$$

Specifically, we have

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} ,$$

which we can prove by induction (Exercise 3.2-7). Since $|\hat{\phi}| < 1$, we have

$$\begin{aligned} \frac{|\hat{\phi}^i|}{\sqrt{5}} &< \frac{1}{\sqrt{5}} \\ &< \frac{1}{2} , \end{aligned}$$

which implies that

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor , \tag{3.25}$$

which is to say that the $i$th Fibonacci number $F_i$ is equal to $\phi^i / \sqrt{5}$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially.

## Exercises

***3.2-1***
Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n) + g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n) \cdot g(n)$ is monotonically increasing.

***3.2-2***
Prove equation (3.16).

***3.2-3***
Prove equation (3.19). Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.

***3.2-4*** ⋆
Is the function $\lceil \lg n \rceil !$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil !$ polynomially bounded?

***3.2-5*** ⋆
Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

***3.2-6***
Show that the golden ratio $\phi$ and its conjugate $\widehat{\phi}$ both satisfy the equation $x^2 = x + 1$.

***3.2-7***
Prove by induction that the $i$th Fibonacci number satisfies the equality

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}} ,$$

where $\phi$ is the golden ratio and $\widehat{\phi}$ is its conjugate.

***3.2-8***
Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n / \ln n)$.

## Problems

### *3-1 Asymptotic behavior of polynomials*

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where $a_d > 0$, be a degree-$d$ polynomial in $n$, and let $k$ be a constant. Use the definitions of the asymptotic notations to prove the following properties.

***a.*** If $k \geq d$, then $p(n) = O(n^k)$.

***b.*** If $k \leq d$, then $p(n) = \Omega(n^k)$.

***c.*** If $k = d$, then $p(n) = \Theta(n^k)$.

***d.*** If $k > d$, then $p(n) = o(n^k)$.

***e.*** If $k < d$, then $p(n) = \omega(n^k)$.

### *3-2 Relative asymptotic growths*

Indicate, for each pair of expressions $(A, B)$ in the table below, whether $A$ is $O$, $o$, $\Omega$, $\omega$, or $\Theta$ of $B$. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

| | $A$ | $B$ | $O$ | $o$ | $\Omega$ | $\omega$ | $\Theta$ |
|---|---|---|---|---|---|---|---|
| ***a.*** | $\lg^k n$ | $n^\epsilon$ | | | | | |
| ***b.*** | $n^k$ | $c^n$ | | | | | |
| ***c.*** | $\sqrt{n}$ | $n^{\sin n}$ | | | | | |
| ***d.*** | $2^n$ | $2^{n/2}$ | | | | | |
| ***e.*** | $n^{\lg c}$ | $c^{\lg n}$ | | | | | |
| ***f.*** | $\lg(n!)$ | $\lg(n^n)$ | | | | | |

### *3-3 Ordering by asymptotic growth rates*

***a.*** Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

| | | | | | |
|---|---|---|---|---|---|
| $\lg(\lg^* n)$ | $2^{\lg^* n}$ | $(\sqrt{2})^{\lg n}$ | $n^2$ | $n!$ | $(\lg n)!$ |
| $(\frac{3}{2})^n$ | $n^3$ | $\lg^2 n$ | $\lg(n!)$ | $2^{2^n}$ | $n^{1/\lg n}$ |
| $\ln\ln n$ | $\lg^* n$ | $n \cdot 2^n$ | $n^{\lg\lg n}$ | $\ln n$ | $1$ |
| $2^{\lg n}$ | $(\lg n)^{\lg n}$ | $e^n$ | $4^{\lg n}$ | $(n+1)!$ | $\sqrt{\lg n}$ |
| $\lg^*(\lg n)$ | $2^{\sqrt{2\lg n}}$ | $n$ | $2^n$ | $n \lg n$ | $2^{2^{n+1}}$ |

***b.*** Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

### *3-4 Asymptotic notation properties*

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

***a.*** $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

***b.*** $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

***c.*** $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large $n$.

***d.*** $f(n) = O(g(n))$ implies $2^{f(n)} = O\left(2^{g(n)}\right)$.

***e.*** $f(n) = O\left((f(n))^2\right)$.

***f.*** $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

***g.*** $f(n) = \Theta(f(n/2))$.

***h.*** $f(n) + o(f(n)) = \Theta(f(n))$.

### *3-5 Variations on O and Ω*

Some authors define $\Omega$ in a slightly different way than we do; let's use $\overset{\infty}{\Omega}$ (read "omega infinity") for this alternative definition. We say that $f(n) = \overset{\infty}{\Omega}(g(n))$ if there exists a positive constant $c$ such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers $n$.

***a.*** Show that for any two functions $f(n)$ and $g(n)$ that are asymptotically nonnegative, either $f(n) = O(g(n))$ or $f(n) = \overset{\infty}{\Omega}(g(n))$ or both, whereas this is not true if we use $\Omega$ in place of $\overset{\infty}{\Omega}$.

***b.*** Describe the potential advantages and disadvantages of using $\overset{\infty}{\Omega}$ instead of $\Omega$ to characterize the running times of programs.

Some authors also define $O$ in a slightly different manner; let's use $O'$ for the alternative definition. We say that $f(n) = O'(g(n))$ if and only if $|f(n)| = O(g(n))$.

***c.*** What happens to each direction of the "if and only if" in Theorem 3.1 if we substitute $O'$ for $O$ but still use $\Omega$?

Some authors define $\widetilde{O}$ (read "soft-oh") to mean $O$ with logarithmic factors ignored:

$$\widetilde{O}(g(n)) = \{f(n) : \text{ there exist positive constants } c, k, \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n)\lg^k(n) \text{ for all } n \ge n_0\} .$$

***d.*** Define $\widetilde{\Omega}$ and $\widetilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.

### *3-6 Iterated functions*

We can apply the iteration operator $^*$ used in the $\lg^*$ function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function $f_c^*$ by

$$f_c^*(n) = \min\left\{i \ge 0 : f^{(i)}(n) \le c\right\} ,$$

which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the number of iterated applications of the function $f$ required to reduce its argument down to $c$ or less.

For each of the following functions $f(n)$ and constants $c$, give as tight a bound as possible on $f_c^*(n)$.

| | $f(n)$ | $c$ | $f_c^*(n)$ |
|---|---|---|---|
| ***a.*** | $n-1$ | 0 | |
| ***b.*** | $\lg n$ | 1 | |
| ***c.*** | $n/2$ | 1 | |
| ***d.*** | $n/2$ | 2 | |
| ***e.*** | $\sqrt{n}$ | 2 | |
| ***f.*** | $\sqrt{n}$ | 1 | |
| ***g.*** | $n^{1/3}$ | 2 | |
| ***h.*** | $n/\lg n$ | 2 | |

## Chapter notes

Knuth [209] traces the origin of the $O$-notation to a number-theory text by P. Bachmann in 1892. The $o$-notation was invented by E. Landau in 1909 for his discussion of the distribution of prime numbers. The $\Omega$ and $\Theta$ notations were advocated by Knuth [213] to correct the popular, but technically sloppy, practice in the literature of using $O$-notation for both upper and lower bounds. Many people continue to use the $O$-notation where the $\Theta$-notation is more technically precise. Further discussion of the history and development of asymptotic notations appears in works by Knuth [209, 213] and Brassard and Bratley [54].

Not all authors define the asymptotic notations in the same way, although the various definitions agree in most common situations. Some of the alternative definitions encompass functions that are not asymptotically nonnegative, as long as their absolute values are appropriately bounded.

Equation (3.20) is due to Robbins [297]. Other properties of elementary mathematical functions can be found in any good mathematical reference, such as Abramowitz and Stegun [1] or Zwillinger [362], or in a calculus book, such as Apostol [18] or Thomas et al. [334]. Knuth [209] and Graham, Knuth, and Patashnik [152] contain a wealth of material on discrete mathematics as used in computer science.