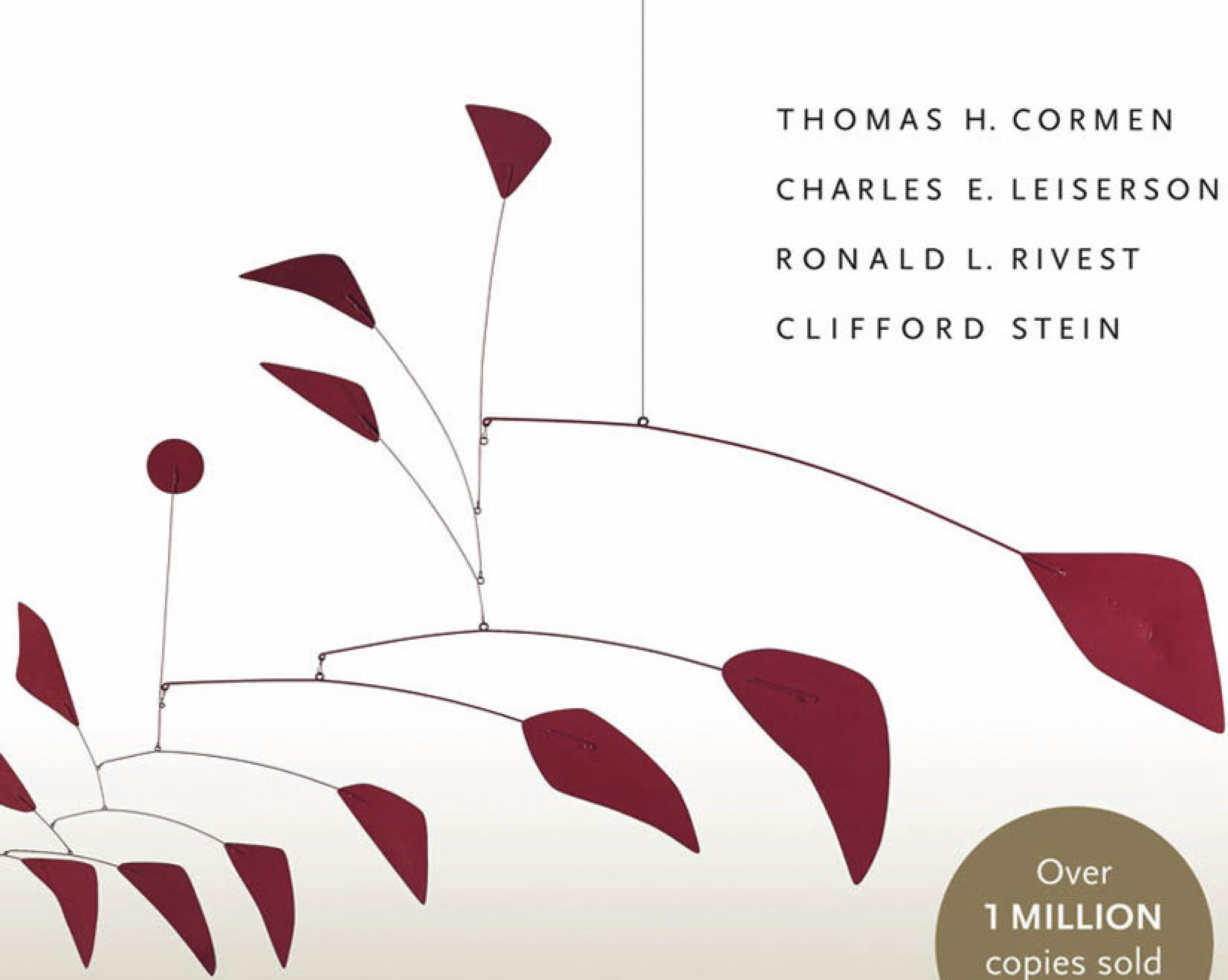
THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN



Over **1 MILLION** copies sold worldwide

INTRODUCTION TO ALGORITHMS

FOURTH EDITION

Introduction to Algorithms

Fourth Edition

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

Introduction to Algorithms

Fourth Edition

The MIT Press
Cambridge, Massachusetts London, England

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Times Roman and MathTime Professional II by the authors.

Names: Cormen, Thomas H., author. | Leiserson, Charles Eric, author. |
Rivest, Ronald L., author. | Stein, Clifford, author.
Title: Introduction to algorithms / Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein.
Description: Fourth edition. | Cambridge, Massachusetts : The MIT Press,
[2022] | Includes bibliographical references and index.
Identifiers: LCCN 2021037260 | ISBN 9780262046305
Subjects: LCSH: Computer programming. | Computer algorithms.
Classification: LCC QA76.6 .C662 2022 | DDC 005.13--dc23
LC record available at http://lccn.loc.gov/2021037260

10 9 8 7 6 5 4 3 2 1

Contents

Preface *xiii*

I Foundations

Introduction *3*

1 The Role of Algorithms in Computing *5*
1.1 Algorithms *5*
1.2 Algorithms as a technology *12*

2 Getting Started *17*
2.1 Insertion sort *17*
2.2 Analyzing algorithms *25*
2.3 Designing algorithms *34*

3 Characterizing Running Times *49*
3.1 O-notation, Ω-notation, and Θ-notation *50*
3.2 Asymptotic notation: formal definitions *53*
3.3 Standard notations and common functions *63*

4 Divide-and-Conquer *76*
4.1 Multiplying square matrices *80*
4.2 Strassen's algorithm for matrix multiplication *85*
4.3 The substitution method for solving recurrences *90*
4.4 The recursion-tree method for solving recurrences *95*
4.5 The master method for solving recurrences *101*
★ 4.6 Proof of the continuous master theorem *107*
★ 4.7 Akra-Bazzi recurrences *115*

5 Probabilistic Analysis and Randomized Algorithms ***126***
5.1 The hiring problem *126*
5.2 Indicator random variables *130*
5.3 Randomized algorithms *134*
★ 5.4 Probabilistic analysis and further uses of indicator random variables *140*

II Sorting and Order Statistics

Introduction ***157***

6 Heapsort ***161***
6.1 Heaps *161*
6.2 Maintaining the heap property *164*
6.3 Building a heap *167*
6.4 The heapsort algorithm *170*
6.5 Priority queues *172*

7 Quicksort ***182***
7.1 Description of quicksort *183*
7.2 Performance of quicksort *187*
7.3 A randomized version of quicksort *191*
7.4 Analysis of quicksort *193*

8 Sorting in Linear Time ***205***
8.1 Lower bounds for sorting *205*
8.2 Counting sort *208*
8.3 Radix sort *211*
8.4 Bucket sort *215*

9 Medians and Order Statistics ***227***
9.1 Minimum and maximum *228*
9.2 Selection in expected linear time *230*
9.3 Selection in worst-case linear time *236*

III Data Structures

Introduction ***249***

10 Elementary Data Structures ***252***
10.1 Simple array-based data structures: arrays, matrices, stacks, queues *252*
10.2 Linked lists *258*
10.3 Representing rooted trees *265*

11 Hash Tables ***272***
11.1 Direct-address tables *273*
11.2 Hash tables *275*
11.3 Hash functions *282*
11.4 Open addressing *293*
11.5 Practical considerations *301*

12 Binary Search Trees ***312***
12.1 What is a binary search tree? *312*
12.2 Querying a binary search tree *316*
12.3 Insertion and deletion *321*

13 Red-Black Trees ***331***
13.1 Properties of red-black trees *331*
13.2 Rotations *335*
13.3 Insertion *338*
13.4 Deletion *346*

IV Advanced Design and Analysis Techniques

Introduction ***361***

14 Dynamic Programming ***362***
14.1 Rod cutting *363*
14.2 Matrix-chain multiplication *373*
14.3 Elements of dynamic programming *382*
14.4 Longest common subsequence *393*
14.5 Optimal binary search trees *400*

15 Greedy Algorithms ***417***
15.1 An activity-selection problem *418*
15.2 Elements of the greedy strategy *426*
15.3 Huffman codes *431*
15.4 Offline caching *440*

16 Amortized Analysis ***448***
16.1 Aggregate analysis *449*
16.2 The accounting method *453*
16.3 The potential method *456*
16.4 Dynamic tables *460*

V Advanced Data Structures

Introduction ***477***

17 Augmenting Data Structures ***480***
17.1 Dynamic order statistics *480*
17.2 How to augment a data structure *486*
17.3 Interval trees *489*

18 B-Trees ***497***
18.1 Definition of B-trees *501*
18.2 Basic operations on B-trees *504*
18.3 Deleting a key from a B-tree *513*

19 Data Structures for Disjoint Sets ***520***
19.1 Disjoint-set operations *520*
19.2 Linked-list representation of disjoint sets *523*
19.3 Disjoint-set forests *527*
★ 19.4 Analysis of union by rank with path compression *531*

VI Graph Algorithms

Introduction ***547***

20 Elementary Graph Algorithms ***549***
20.1 Representations of graphs *549*
20.2 Breadth-first search *554*
20.3 Depth-first search *563*
20.4 Topological sort *573*
20.5 Strongly connected components *576*

21 Minimum Spanning Trees ***585***
21.1 Growing a minimum spanning tree *586*
21.2 The algorithms of Kruskal and Prim *591*

22 Single-Source Shortest Paths ***604***
22.1 The Bellman-Ford algorithm *612*
22.2 Single-source shortest paths in directed acyclic graphs *616*
22.3 Dijkstra's algorithm *620*
22.4 Difference constraints and shortest paths *626*
22.5 Proofs of shortest-paths properties *633*

23 All-Pairs Shortest Paths ***646***
23.1 Shortest paths and matrix multiplication *648*
23.2 The Floyd-Warshall algorithm *655*
23.3 Johnson's algorithm for sparse graphs *662*

24 Maximum Flow ***670***
24.1 Flow networks *671*
24.2 The Ford-Fulkerson method *676*
24.3 Maximum bipartite matching *693*

25 Matchings in Bipartite Graphs ***704***
25.1 Maximum bipartite matching (revisited) *705*
25.2 The stable-marriage problem *716*
25.3 The Hungarian algorithm for the assignment problem *723*

VII Selected Topics

Introduction ***745***

26 Parallel Algorithms ***748***
26.1 The basics of fork-join parallelism *750*
26.2 Parallel matrix multiplication *770*
26.3 Parallel merge sort *775*

27 Online Algorithms ***791***
27.1 Waiting for an elevator *792*
27.2 Maintaining a search list *795*
27.3 Online caching *802*

28 Matrix Operations ***819***
28.1 Solving systems of linear equations *819*
28.2 Inverting matrices *833*
28.3 Symmetric positive-definite matrices and least-squares approximation *838*

29 Linear Programming ***850***
29.1 Linear programming formulations and algorithms *853*
29.2 Formulating problems as linear programs *860*
29.3 Duality *866*

30 Polynomials and the FFT ***877***
30.1 Representing polynomials *879*
30.2 The DFT and FFT *885*
30.3 FFT circuits *894*

31 Number-Theoretic Algorithms *903*
31.1 Elementary number-theoretic notions *904*
31.2 Greatest common divisor *911*
31.3 Modular arithmetic *916*
31.4 Solving modular linear equations *924*
31.5 The Chinese remainder theorem *928*
31.6 Powers of an element *932*
31.7 The RSA public-key cryptosystem *936*
★ 31.8 Primality testing *942*

32 String Matching *957*
32.1 The naive string-matching algorithm *960*
32.2 The Rabin-Karp algorithm *962*
32.3 String matching with finite automata *967*
★ 32.4 The Knuth-Morris-Pratt algorithm *975*
32.5 Suffix arrays *985*

33 Machine-Learning Algorithms *1003*
33.1 Clustering *1005*
33.2 Multiplicative-weights algorithms *1015*
33.3 Gradient descent *1022*

34 NP-Completeness *1042*
34.1 Polynomial time *1048*
34.2 Polynomial-time verification *1056*
34.3 NP-completeness and reducibility *1061*
34.4 NP-completeness proofs *1072*
34.5 NP-complete problems *1080*

35 Approximation Algorithms *1104*
35.1 The vertex-cover problem *1106*
35.2 The traveling-salesperson problem *1109*
35.3 The set-covering problem *1115*
35.4 Randomization and linear programming *1119*
35.5 The subset-sum problem *1124*

VIII Appendix: Mathematical Background

Introduction *1139*

A Summations *1140*
A.1 Summation formulas and properties *1140*
A.2 Bounding summations *1145*

B **Sets, Etc.** ***1153***
B.1 Sets *1153*
B.2 Relations *1158*
B.3 Functions *1161*
B.4 Graphs *1164*
B.5 Trees *1169*

C **Counting and Probability** ***1178***
C.1 Counting *1178*
C.2 Probability *1184*
C.3 Discrete random variables *1191*
C.4 The geometric and binomial distributions *1196*
★ C.5 The tails of the binomial distribution *1203*

D **Matrices** ***1214***
D.1 Matrices and matrix operations *1214*
D.2 Basic matrix properties *1219*

Bibliography ***1227***

Index ***1251***

Preface

Not so long ago, anyone who had heard the word "algorithm" was almost certainly a computer scientist or mathematician. With computers having become prevalent in our modern lives, however, the term is no longer esoteric. If you look around your home, you'll find algorithms running in the most mundane places: your microwave oven, your washing machine, and, of course, your computer. You ask algorithms to make recommendations to you: what music you might like or what route to take when driving. Our society, for better or for worse, asks algorithms to suggest sentences for convicted criminals. You even rely on algorithms to keep you alive, or at least not to kill you: the control systems in your car or in medical equipment.[1] The word "algorithm" appears somewhere in the news seemingly every day.

Therefore, it behooves you to understand algorithms not just as a student or practitioner of computer science, but as a citizen of the world. Once you understand algorithms, you can educate others about what algorithms are, how they operate, and what their limitations are.

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 accessible to all levels of readers. All the analyses are laid out, some simple, some more involved. We have tried to keep explanations clear 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 231 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 the algorithms.

[1] To understand many of the ways in which algorithms influence our daily lives, see the book by Fry [162].

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 fourth edition, we have once again updated the entire book. The changes cover a broad spectrum, including new chapters and sections, color illustrations, and what we hope you'll find to be a more engaging 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 select 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. Whereas in an undergraduate course, you might use only some sections from a chapter, in a graduate course, you might cover the entire chapter.

We have included 931 exercises and 162 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, but many are substantial and suitable as assigned homework. The problems include more elaborate case studies which often introduce new material. They often consist of several parts that lead the student through the steps required to arrive at a solution.

As with the third edition of this book, we have made publicly available solutions to some, but by no means all, of the problems and exercises. You can find these solutions on our website, http://mitpress.mit.edu/algorithms/. You will want to check this site to see whether it contains the solution to an exercise or problem that you plan to assign. Since the set of solutions that we post might grow over time, we recommend that you check the site 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 and supporting figures to help you visualize what is going on.

Since this book is large, your class will probably cover only a portion of its material. Although we hope that you will find this book helpful to you as a course textbook now, we have also tried to make it comprehensive enough to warrant space on your future professional bookshelf.

What are the prerequisites for reading this book?

- You need some programming experience. In particular, you should understand recursive procedures and simple data structures, such as arrays and linked lists (although Section 10.2 covers linked lists and a variant that you may find new).
- 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. Although this book uses mathematics throughout, Part I and Appendices A–D teach you all the mathematical techniques you will need.

Our website, http://mitpress.mit.edu/algorithms/, links to solutions for some of the problems and exercises. Feel free to check your solutions against ours. We ask, however, that you 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 on the topics most relevant to you.

Since most of the algorithms we discuss have great practical utility, we 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. If you are used to 0-origin arrays, you might find our frequent practice of indexing arrays from 1 a minor stumbling block. You can always either subtract 1 from our indices or just overallocate the array and leave position 0 unused.

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 website, 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 adopted the policy of not citing references for them, removing the temptation for students to look up a solution rather than to discover it themselves.

Changes for the fourth edition

As we said about the changes for the second and third editions, 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 third-edition chapters and sections appear in the fourth edition. We removed three chapters and several sections, but we have added three new chapters and several new sections apart from these new chapters.

We kept the hybrid organization from the first three editions. Rather than organizing chapters only by problem domains or only according to techniques, this book incorporates elements of both. It contains technique-based chapters on divide-and-conquer, dynamic programming, greedy algorithms, amortized analysis, augmenting data structures, 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.

Some of the changes in the fourth edition apply generally across the book, and some are specific to particular chapters or sections. Here is a summary of the most significant general changes:

- We added 140 new exercises and 22 new problems. We also improved many of the old exercises and problems, often as the result of reader feedback. (Thanks to all readers who made suggestions.)

- We have color! With designers from the MIT Press, we selected a limited palette, devised to convey information and to be pleasing to the eye. (We are delighted to display red-black trees in—get this—red and black!) To enhance readability, defined terms, pseudocode comments, and page numbers in the index are in color.
- Pseudocode procedures appear on a tan background to make them easier to spot, and they do not necessarily appear on the page of their first reference. When they don't, the text directs you to the relevant page. In the same vein, nonlocal references to numbered equations, theorems, lemmas, and corollaries include the page number.
- We removed topics that were rarely taught. We dropped in their entirety the chapters on Fibonacci heaps, van Emde Boas trees, and computational geometry. In addition, the following material was excised: the maximum-subarray problem, implementing pointers and objects, perfect hashing, randomly built binary search trees, matroids, push-relabel algorithms for maximum flow, the iterative fast Fourier transform method, the details of the simplex algorithm for linear programming, and integer factorization. You can find all the removed material on our website, http://mitpress.mit.edu/algorithms/.
- We reviewed the entire book and rewrote sentences, paragraphs, and sections to make the writing clearer, more personal, and gender neutral. For example, the "traveling-salesman problem" in the previous editions is now called the "traveling-salesperson problem." We believe that it is critically important for engineering and science, including our own field of computer science, to be welcoming to everyone. (The one place that stumped us is in Chapter 13, which requires a term for a parent's sibling. Because the English language has no such gender-neutral term, we regretfully stuck with "uncle.")
- The chapter notes, bibliography, and index were updated, reflecting the dramatic growth of the field of algorithms since the third edition.
- We corrected errors, posting most corrections on our website of third-edition errata. Those that were reported while we were in full swing preparing this edition were not posted, but were corrected in this edition. (Thanks again to all readers who helped us identify issues.)

The specific changes for the fourth edition include the following:

- We renamed Chapter 3 and added a section giving an overview of asymptotic notation before delving into the formal definitions.
- Chapter 4 underwent substantial changes to improve its mathematical foundation and make it more robust and intuitive. The notion of an algorithmic recurrence was introduced, and the topic of ignoring floors and ceilings in recur-

rences was addressed more rigorously. The second case of the master theorem incorporates polylogarithmic factors, and a rigorous proof of a "continuous" version of the master theorem is now provided. We also present the powerful and general Akra-Bazzi method (without proof).

- The deterministic order-statistic algorithm in Chapter 9 is slightly different, and the analyses of both the randomized and deterministic order-statistic algorithms have been revamped.
- In addition to stacks and queues, Section 10.1 discusses ways to store arrays and matrices.
- Chapter 11 on hash tables includes a modern treatment of hash functions. It also emphasizes linear probing as an efficient method for resolving collisions when the underlying hardware implements caching to favor local searches.
- To replace the sections on matroids in Chapter 15, we converted a problem in the third edition about offline caching into a full section.
- Section 16.4 now contains a more intuitive explanation of the potential functions to analyze table doubling and halving.
- Chapter 17 on augmenting data structures was relocated from Part III to Part V, reflecting our view that this technique goes beyond basic material.
- Chapter 25 is a new chapter about matchings in bipartite graphs. It presents algorithms to find a matching of maximum cardinality, to solve the stable-marriage problem, and to find a maximum-weight matching (known as the "assignment problem").
- Chapter 26, on task-parallel computing, has been updated with modern terminology, including the name of the chapter.
- Chapter 27, which covers online algorithms, is another new chapter. In an online algorithm, the input arrives over time, rather than being available in its entirety at the start of the algorithm. The chapter describes several examples of online algorithms, including determining how long to wait for an elevator before taking the stairs, maintaining a linked list via the move-to-front heuristic, and evaluating replacement policies for caches.
- In Chapter 29, we removed the detailed presentation of the simplex algorithm, as it was math heavy without really conveying many algorithmic ideas. The chapter now focuses on the key aspect of how to model problems as linear programs, along with the essential duality property of linear programming.
- Section 32.5 adds to the chapter on string matching the simple, yet powerful, structure of suffix arrays.

- Chapter 33, on machine learning, is the third new chapter. It introduces several basic methods used in machine learning: clustering to group similar items together, weighted-majority algorithms, and gradient descent to find the minimizer of a function.
- Section 34.5.6 summarizes strategies for polynomial-time reductions to show that problems are NP-hard.
- The proof of the approximation algorithm for the set-covering problem in Section 35.3 has been revised.

Website

You can use our website, http://mitpress.mit.edu/algorithms/, to obtain supplementary information and to communicate with us. The website links to a list of known errors, material from the third edition that is not included in the fourth edition, solutions to selected exercises and problems, Python implementations of many of the algorithms in this book, a list explaining the corny professor jokes (of course), as well as other content, which we may add to. The website also tells you how to report errors or make suggestions.

How we produced this book

Like the previous three editions, the fourth edition was produced in LaTeX 2_ε. We used the Times font with mathematics typeset using the MathTime Professional II fonts. As in all previous editions, we compiled the index using Windex, a C program that we wrote, and produced the bibliography using BIBTeX. The PDF files for this book were created on a MacBook Pro running macOS 10.14.

Our plea to Apple in the preface of the third edition to update MacDraw Pro for macOS 10 went for naught, and so we continued to draw illustrations on pre-Intel Macs running MacDraw Pro under the Classic environment of older versions of macOS 10. Many of the mathematical expressions appearing in illustrations were laid in with the psfrag package for LaTeX 2_ε.

Acknowledgments for the fourth edition

We have been working with the MIT Press since we started writing the first edition in 1987, collaborating with several directors, editors, and production staff. Throughout our association with the MIT Press, their support has always been outstanding. Special thanks to our editors Marie Lee, who put up with us for far too long, and Elizabeth Swayze, who pushed us over the finish line. Thanks also to Director Amy Brand and to Alex Hoopes.

As in the third edition, we were geographically distributed while producing the fourth edition, working in the Dartmouth College Department of Computer Science; the MIT Computer Science and Artificial Intelligence Laboratory and the MIT Department of Electrical Engineering and Computer Science; and the Columbia University Department of Industrial Engineering and Operations Research, Department of Computer Science, and Data Science Institute. During the COVID-19 pandemic, we worked largely from home. We thank our respective universities and colleagues for providing such supportive and stimulating environments. As we complete this book, those of us who are not retired are eager to return to our respective universities now that the pandemic seems to be abating.

Julie Sussman, P.P.A., came to our rescue once again with her technical copyediting under tremendous time pressure. If not for Julie, this book would be riddled with errors (or, let's say, many more errors than it has) and would be far less readable. Julie, we will be forever indebted to you. Errors that remain are the responsibility of the authors (and probably were inserted after Julie read the material).

Dozens of errors in previous editions were corrected in the process of creating this edition. We thank our readers—too many to list them all—who have reported errors and suggested improvements over the years.

We received considerable help in preparing some of the new material in this edition. Neville Campbell (unaffiliated), Bill Kuszmaul of MIT, and Chee Yap of NYU provided valuable advice regarding the treatment of recurrences in Chapter 4. Yan Gu of the University of California, Riverside, provided feedback on parallel algorithms in Chapter 26. Rob Shapire of Microsoft Research altered our approach to the material on machine learning with his detailed comments on Chapter 33. Qi Qi of MIT helped with the analysis of the Monty Hall problem (Problem C-1).

Molly Seaman and Mary Reilly of the MIT Press helped us select the color palette in the illustrations, and Wojciech Jarosz of Dartmouth College suggested design improvements to our newly colored figures. Yichen (Annie) Ke and Linda Xiao, who have since graduated from Dartmouth, aided in colorizing the illustrations, and Linda also produced many of the Python implementations that are available on the book's website.

Finally, we thank our wives—Wendy Leiserson, Gail Rivest, Rebecca Ivry, and the late Nicole Cormen—and our families. The patience and encouragement of those who love us 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*

June, 2021

Part I Foundations

Introduction

When you design and analyze algorithms, you need to be able to describe how they operate and how to design them. You also need some mathematical tools to show that your algorithms do the right thing and do it efficiently. This part will get you started. 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 for considering 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 "asymptotic" notation to express them.

Chapter 3 precisely defines asymptotic notation. We'll use asymptotic notation to bound the growth of functions—most often, functions that describe the running time of algorithms—from above and below. The chapter starts by informally defining the most commonly used asymptotic notations and giving an example of how to apply them. It then formally defines five asymptotic notations and presents conventions for how to put them together. 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 two additional examples of divide-and-conquer algorithms for multiplying square matrices, including Strassen's surprising method. Chapter 4 contains methods for solving recurrences, which are useful for describing the running times of recursive algorithms. In the substitution method, you guess an answer and prove it correct. Recursion trees provide one way to generate a guess. Chapter 4 also presents the powerful technique of the "master method," which you can often use to solve recurrences that arise from divide-and-conquer algorithms. Although the chapter provides a proof of a foundational theorem on which the master theorem depends, you should feel free to employ the master method without delving into the proof. Chapter 4 concludes with some advanced topics.

Chapter 5 introduces probabilistic analysis and randomized algorithms. You 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, you might assume that the inputs conform to a known probability distribution, so that you 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. You 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 might 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? This chapter 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*** in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output.

You 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 for problem instances, typically of arbitrarily large size. The algorithm describes a specific computational procedure for achieving that input/output relationship for all problem instances.

As an example, suppose that you need to sort a sequence of numbers into monotonically increasing 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 $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Thus, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a correct 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***[1] 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, you have a large number of good sorting algorithms at your 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—archaically—tapes.

An algorithm for a computational problem is ***correct*** if, for every problem instance provided as input, it ***halts***—finishes its computing in finite time—and outputs the correct solution to the problem instance. 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 you can control their error rate. We'll 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'll concern ourselves 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 roughly 30,000 genes in human DNA, determining the sequences of the roughly 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 presented here, enabling scientists to accomplish tasks while using resources efficiently. Dynamic programming, as

[1] Sometimes, when the problem context is known, problem instances are themselves simply called "problems."

in Chapter 14, is an important technique for solving several of these biological problems, particularly ones that involve determining similarity between DNA sequences. The savings realized are in time, both human and machine, and in money, as more information can be extracted by 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 travels (techniques for solving such problems appear in Chapter 22), 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 might wish to know where to place its wells in order to maximize its expected profit. A political candidate might want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline might 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 might 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 by modeling them as linear programs, which Chapter 29 explores.

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:

- You have a road map on which the distance between each pair of adjacent intersections is marked, and you wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if you disallow routes that cross over themselves. How can you choose which of all possible routes is the shortest? You can start by modeling 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). In this graph, you wish to find the shortest path from one vertex to another. Chapter 22 shows how to solve this problem efficiently.

- Given a mechanical design in terms of a library of parts, where each part may include instances of other parts, 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, you cannot feasibly generate each possible order and then verify that, within that order, each part appears before the parts using it (unless you have only a few parts). This problem is an instance of topological sorting, and Chapter 20 shows how to solve this problem efficiently.
- A doctor needs to determine whether an image represents a cancerous tumor or a benign one. The doctor has available images of many other tumors, some of which are known to be cancerous and some of which are known to be benign. A cancerous tumor is likely to be more similar to other cancerous tumors than to benign tumors, and a benign tumor is more likely to be similar to other benign tumors. By using a clustering algorithm, as in Chapter 33, the doctor can identify which outcome is more likely.
- You need to compress a large file containing text so that it occupies less space. Many ways to do so are known, including "LZW compression," which looks for repeating character sequences. Chapter 15 studies a different approach, "Huffman coding," which encodes characters by bit sequences of various lengths, with characters occurring more frequently encoded by shorter bit sequences.

These lists are far from exhaustive (as you again have probably surmised from this book's heft), but they exhibit two characteristics 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," without explicitly examining each possible solution, 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 might 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 might want to find driving directions using a navigation app.

Not every problem solved by algorithms has an easily identified set of candidate solutions. For example, given a set of numerical values representing samples of a signal taken at regular time intervals, the discrete Fourier transform converts the time domain to the frequency domain. That is, it approximates the signal as a weighted sum of sinusoids, producing the strength of various frequencies which, when summed, approximate 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. The chapter also sketches out the design of a hardware FFT circuit.

Data structures

This book also presents several data structures. A ***data structure*** is a way to store and organize data in order to facilitate access and modifications. Using the appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes, and so you should know the strengths and limitations of several of them.

Technique

Although you can use this book as a "cookbook" for algorithms, you might 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 analyze 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 21, and determining a maximum flow in a network in Chapter 24. Other chapters introduce techniques, such as divide-and-conquer in Chapters 2 and 4, dynamic programming in Chapter 14, and amortized analysis in Chapter 16.

Hard problems

Most of this book is about efficient algorithms. Our usual measure of efficiency is speed: how long does an algorithm take to produce its result? There are some problems, however, for which we know of no algorithm that runs in a reasonable amount of time. 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 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, instead, you can show that the problem is NP-complete, you can spend your time developing an efficient approximation algorithm, that is, an algorithm that gives a good, but not necessarily the best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each day, it loads up delivery trucks at the depot and sends them 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-salesperson problem," and it is NP-complete.[2] It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that compute overall distances close to the smallest possible. Chapter 35 discusses such "approximation algorithms."

Alternative computing models

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 26 presents a model for ”task-parallel” algorithms, which take advantage of multiple processing cores. This model has advantages from both theoretical and

[2] To be precise, only decision problems—those with a "yes/no" answer—can be NP-complete. The decision version of the traveling salesperson problem asks whether there exists an order of stops whose distance totals at most a given amount.

practical standpoints, and many modern parallel-programming platforms embrace something similar to this model of parallelism.

Most of the examples in this book assume that all of the input data are available when an algorithm begins running. Much of the work in algorithm design makes the same assumption. For many important real-world examples, however, the input actually arrives over time, and the algorithm must decide how to proceed without knowing what data will arrive in the future. In a data center, jobs are constantly arriving and departing, and a scheduling algorithm must decide when and where to run a job, without knowing what jobs will be arriving in the future. Traffic must be routed in the internet based on the current state, without knowing about where traffic will arrive in the future. Hospital emergency rooms make triage decisions about which patients to treat first without knowing when other patients will be arriving in the future and what treatments they will need. Algorithms that receive their input over time, rather than having all the input present at the start, are ***online algorithms***, which Chapter 27 examines.

Exercises

1.1-1
Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

1.1-2
Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

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

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

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

1.1-6
Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

1.2 Algorithms as a technology

If computers were infinitely fast and computer memory were 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 be certain 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. Computing time is therefore a bounded resource, which makes it precious. Although the saying goes, "Time is money," time is even more valuable than money: you can get back money after you spend it, but once time is spent, you can never get it back. Memory may be inexpensive, but it is neither infinite nor free. You should choose algorithms that use the resources of time and space efficiently.

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, Chapter 2 introduces 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'll 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 is 1000, $\lg n$ is approximately 10, and when n is 1,000,000, $\lg n$ 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 $\lg n$ versus n more than compensates for the difference in constant factors. No matter how much smaller c_1 is than c_2, there is always 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 (much slower than most contemporary computers), 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

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

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 sorting 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. Although 100 million might seem like a large number, there are more than 100 million web searches every half hour, more than 100 million emails sent every minute, and some of the smallest galaxies (known as ultra-compact dwarf galaxies) contain about 100 million stars. In general, as the problem size increases, so does the relative advantage of merge sort.

Algorithms and other technologies

The example above shows that you 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,
- fast networking, both wired and wireless,
- machine learning,
- and mobile devices.

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. It would also require algorithms for 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.

Machine learning can be thought of as a method for performing algorithmic tasks without explicitly designing an algorithm, but instead inferring patterns from data and thereby automatically learning a solution. At first glance, machine learning, which automates the process of algorithmic design, may seem to make learning about algorithms obsolete. The opposite is true, however. Machine learning is itself a collection of algorithms, just under a different name. Furthermore, it currently seems that the successes of machine learning are mainly for problems for which we, as humans, do not really understand what the right algorithm is. Prominent examples include computer vision and automatic language translation. For algorithmic problems that humans understand well, such as most of the problems in this book, efficient algorithms designed to solve a specific problem are typically more successful than machine-learning approaches.

Data science is an interdisciplinary field with the goal of extracting knowledge and insights from structured and unstructured data. Data science uses methods from statistics, computer science, and optimization. The design and analysis of algorithms is fundamental to the field. The core techniques of data science, which overlap significantly with those in machine learning, include many of the algorithms in this book.

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 defines the truly skilled programmer. 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 that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort runs in $64 n \lg n$ 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], Dasgupta, Papadimitriou, and Vazirani [107], Edmonds [133], Erickson [135], Goodrich and Tamassia [195, 196], Kleinberg and Tardos [257], Knuth [259, 260, 261, 262, 263], Levitin [298], Louridas [305], Mehlhorn and Sanders [325], Mitzenmacher and Upfal [331], Neapolitan [342], Roughgarden [385, 386, 387, 388], Sanders, Mehlhorn, Dietzfelbinger, and Dementiev [393], Sedgewick and Wayne [402], Skiena [414], Soltys-Kulinicz [419], Wilf [455], and Williamson and Shmoys [459]. Some of the more practical aspects of algorithm design are discussed by Bentley [49, 50, 51], Bhargava [54], Kochenderfer and Wheeler [268], and McGeoch [321]. Surveys of the field of algorithms can also be found in books by Atallah and Blanton [27, 28] and Mehta and Sahhi [326]. For less technical material, see the books by Christian and Griffiths [92], Cormen [104], Erwig [136], MacCormick [307], and Vöcking et al. [448]. Overviews of the algorithms used in computational biology can be found in books by Jones and Pevzner [240], Elloumi and Zomaya [134], and Marchisio [315].

2 Getting Started

This chapter will familiarize you with the framework we'll 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 will be introduced in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We'll begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We'll specify algorithms using a pseudocode that should be understandable to you if you have done computer programming. We'll see why insertion sort correctly sorts and analyze its running time. The analysis introduces a notation that describes how running time increases with the number of items to be sorted. Following a discussion of insertion sort, we'll use a method called divide-and-conquer to develop a sorting algorithm called merge sort. We'll 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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

The numbers to be sorted are also known as the ***keys***. Although the problem is conceptually about sorting a sequence, the input comes in the form of an array with n elements. When we want to sort numbers, it's often because they are the keys associated with other data, which we call ***satellite data***. Together, a key and satellite data form a ***record***. For example, consider a spreadsheet containing student records with many associated pieces of data such as age, grade-point average, and number of courses taken. Any one of these quantities could be a key, but when the spreadsheet sorts, it moves the associated record (the satellite data) with the key. When describing a sorting algorithm, we focus on the keys, but it is important to remember that there usually is associated satellite data.

In this book, we'll typically describe algorithms as procedures written in a ***pseudocode*** that is similar in many respects to C, C++, Java, Python,[1] or JavaScript. (Apologies if we've omitted your favorite programming language. We can't list them all.) If you have been introduced to any of these languages, you should have little trouble understanding algorithms "coded" in pseudocode. 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 that looks more like real code. Another difference between pseudocode and real code is that pseudocode often ignores aspects of software engineering—such as data abstraction, modularity, and error handling—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 you might sort a hand of playing cards. Start with an empty left hand and the cards in a pile on the table. Pick up the first card in the pile and hold it with your left hand. Then, with your right hand, remove one card at a time from the pile, and insert it into the correct position in your left hand. As Figure 2.1 illustrates, you find the correct position for a card by comparing it with each of the cards already in your left hand, starting at the right and moving left. As soon as you see a card in your left hand whose value is less than or equal to the card you're holding in your right hand, insert the card that you're holding in your right hand just to the right of this card in your left hand. If all the cards in your left hand have values greater than the card in your right hand, then place this card as the leftmost card in your left hand. At all times, the cards held in your left hand are sorted, and these cards were originally the top cards of the pile on the table.

The pseudocode for insertion sort is given as the procedure INSERTION-SORT on the facing page. It takes two parameters: an array A containing the values to be sorted and the number n of values of sort. The values occupy positions $A[1]$ through $A[n]$ of the array, which we denote by $A[1:n]$. When the INSERTION-SORT procedure is finished, array $A[1:n]$ contains the original values, but in sorted order.

[1] If you're familiar with only Python, you can think of arrays as similar to Python lists.

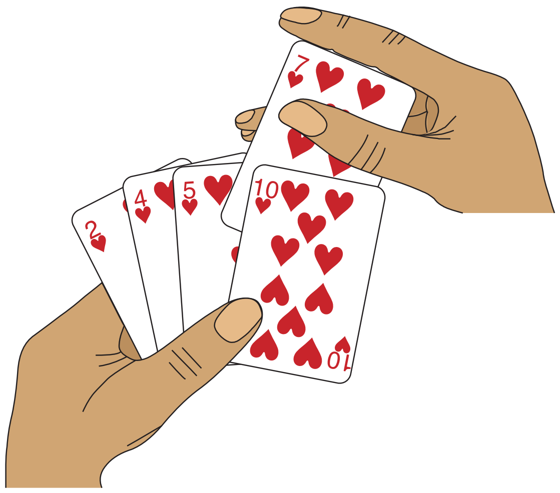


Figure 2.1 Sorting a hand of cards using insertion sort.

```
INSERTION-SORT($A, n$)
1  for $i = 2$ to $n$
2      $key = A[i]$
3      // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.
4      $j = i - 1$
5      while $j > 0$ and $A[j] > key$
6          $A[j+1] = A[j]$
7          $j = j - 1$
8      $A[j+1] = key$
```

Loop invariants and the correctness of insertion sort

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

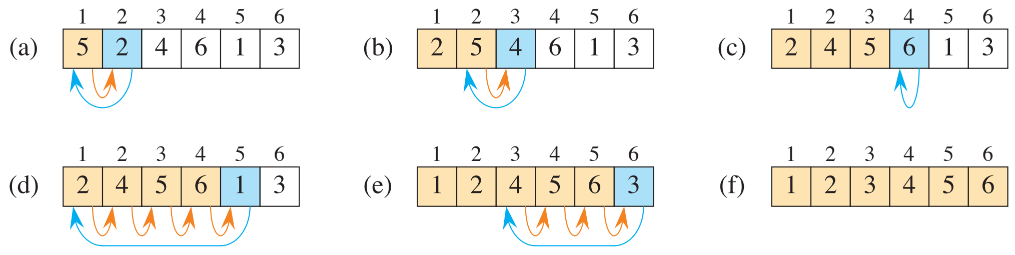


Figure 2.2 The operation of INSERTION-SORT(A, n), where A initially contains the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$ and $n = 6$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of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

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

Loop invariants help us understand why an algorithm is correct. When you're using a loop invariant, you need to show three things:

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: The loop terminates, and when it terminates, the invariant—usually along with the reason that the loop terminated—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, you are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before each iteration.) A loop-invariant proof is a form of 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 you are using the loop invariant to show correctness. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction typically applies the inductive step infinitely, but in a loop invariant the "induction" stops when the loop terminates.

Let's see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $i = 2$.[2] The subarray $A[1:i-1]$ consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (after all, how could a subarray with just one value not be sorted?), 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 the values in $A[i-1]$, $A[i-2]$, $A[i-3]$, and so on by one position to the right until it finds the proper position for $A[i]$ (lines 4–7), at which point it inserts the value of $A[i]$ (line 8). The subarray $A[1:i]$ then consists of the elements originally in $A[1:i]$, but in sorted order. ***Incrementing*** i (increasing its value by 1) 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. Let's not get bogged down in such formalism just yet. Instead, we'll rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine loop termination. The loop variable i starts at 2 and increases by 1 in each iteration. Once i's value exceeds n in line 1, the loop terminates. That is, the loop terminates once i equals $n+1$. Substituting $n+1$ for i in the wording of the loop invariant yields that the subarray $A[1:n]$ consists of the elements originally in $A[1:n]$, but in sorted order. Hence, the algorithm is correct.

This method of loop invariants is used to show correctness in various places throughout this book.

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

[2] When the loop is a **for** loop, the loop-invariant check just prior to the first iteration occurs 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 i but before the first test of whether $i \le n$.

begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if**-**else** statements[3] as well. Using indentation instead of textual indicators of block structure, such as **begin** and **end** statements or curly braces, reduces clutter while preserving, or even enhancing, clarity.[4]

- 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 JavaScript.[5] In this book, the loop counter retains its value after the loop is exited, unlike some situations that arise in C++ and Java. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound.[6] We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $i = 2$ **to** n, and so when this loop terminates, i equals $n+1$. We use the keyword **to** when a **for** loop increments its loop counter in each iteration, and we use the keyword **downto** when a **for** loop ***decrements*** its loop counter (reduces its value by 1 in each iteration). 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.
- Variables (such as i, j, and *key*) are local to the given procedure. We won't 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 ith element of the array A.

  Although many programming languages enforce 0-origin indexing for arrays (0 is the smallest valid index), we choose whichever indexing scheme is clearest for human readers to understand. Because people usually start counting at 1, not 0, most—but not all—of the arrays in this book use 1-origin indexing. To be

---

[3] In an **if**-**else** statement, we indent **else** at the same level as its matching **if**. The first executable line of an **else** clause appears on the same line as the keyword **else**. For multiway tests, we use **elseif** for tests after the first one. When it is the first line in an **else** clause, an **if** statement appears on the line following **else** so that you do not misconstrue it as **elseif**.

[4] Each pseudocode procedure in this book appears on one page so that you do not need to discern levels of indentation in pseudocode that is split across pages.

[5] Most block-structured languages have equivalent constructs, though the exact syntax may differ. Python lacks **repeat**-**until** loops, and its **for** loops operate differently from the **for** loops in this book. Think of the pseudocode line "**for** $i = 1$ **to** n" as equivalent to "for i in range(1, n+1)" in Python.

[6] In Python, the loop counter retains its value after the loop is exited, but the value it retains is the value it had during the final iteration of the **for** loop, rather than the value that exceeded the loop bound. That is because a Python **for** loop iterates through a list, which may contain nonnumeric values.

clear about whether a particular algorithm assumes 0-origin or 1-origin indexing, we'll specify the bounds of the arrays explicitly. If you are implementing an algorithm that we specify using 1-origin indexing, but you're writing in a programming language that enforces 0-origin indexing (such as C, C++, Java, Python, or JavaScript), then give yourself credit for being able to adjust. You can either always subtract 1 from each index or allocate each array with one extra position and just ignore position 0.

The notation ":" denotes a subarray. Thus, $A[i:j]$ indicates the subarray of A consisting of the elements $A[i], A[i+1], \ldots, A[j]$.[7] We also use this notation to indicate the bounds of an array, as we did earlier when discussing the array $A[1:n]$.

- 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, if an object x has attribute f, we denote this attribute by $x.f$.

  We treat a variable representing an array or object as a pointer (known as a reference in some programming languages) 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 $y = x$. This way of treating arrays and objects is consistent with most contemporary programming languages.

  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 refers 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

[7] If you're used to programming in Python, bear in mind that in this book, the subarray $A[i:j]$ includes the element $A[j]$. In Python, the last element of $A[i:j]$ is $A[j-1]$. Python allows negative indices, which count from the back end of the list. This book does not use negative array indices.

the called procedure is not visible to the calling procedure. The assignment $x.f = 3$, however, is visible if the calling procedure has a pointer to the same object as x. 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. Again, most contemporary programming languages work this way.

- 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 without having to create objects to package them together.[8]
- The boolean operators "and" and "or" are ***short circuiting***. That is, evaluate the expression "x and y" by first evaluating x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and therefore y is not evaluated. If, on the other hand, x evaluates to TRUE, y must be evaluated to determine the value of the entire expression. Similarly, in the expression "x or y" the expression y is evaluated 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 upon evaluating $x.f$ when x is NIL.
- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called, and the procedure immediately terminates. 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 an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1:n]$.

[8] Python's tuple notation allows **return** statements to return multiple values without creating objects from a programmer-defined class.

```
SUM-ARRAY(A, n)
1  sum = 0
2  for i = 1 to n
3      sum = sum + A[i]
4  return sum
```

2.1-3

Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

2.1-4

Consider the ***searching problem***:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1:n]$ and a value x.

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

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-5

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays $A[0:n-1]$ and $B[0:n-1]$, where each element is either 0 or 1, $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$, and $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. The sum $c = a + b$ of the two integers should be stored in binary form in an $(n+1)$-element array $C[0:n]$, where $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. You might consider resources such as memory, communication bandwidth, or energy consumption. Most often, however, you'll want to measure computational time. If you analyze several candidate algorithms for a problem, you can identify the most efficient one. There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology that it runs on, including the resources of that technology and a way to express their costs. Most of this book assumes a generic one-processor, ***random-access machine (RAM)*** model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs. In the RAM model, instructions execute one after another, with no concurrent operations. The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access—using the value of a variable or storing into a variable—takes the same amount of time as any other data access. In other words, in the RAM model each instruction or data access takes a constant amount of time—even indexing into an array.[9]

Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and 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 you could sort in just one step. Such a RAM would be unrealistic, since such instructions do not appear in real computers. 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).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character. Real computers do not usually have a separate data type for the boolean values TRUE and FALSE. Instead, they often test whether an integer value is 0 (FALSE) or nonzero (TRUE), as in C. Although we typically do not concern ourselves with precision for floating-point values in this book (many numbers cannot be represented exactly in floating point), precision is crucial for most applications. We also assume that each word of data has a limit on the number of bits. For example, when working with inputs of size n, we typically

[9] We assume that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations. For example, if array $A[1:n]$ starts at memory address 1000 and each element occupies four bytes, then element $A[i]$ is at address $1000 + 4(i - 1)$. In general, computing the address in memory of a particular array element requires at most one subtraction (no subtraction for a 0-origin array), one multiplication (often implemented as a shift operation if the element size is an exact power of 2), and one addition. Furthermore, for code that iterates through the elements of an array in order, an optimizing compiler can generate the address of each element using just one addition, by adding the element size to the address of the preceding element.

assume that integers are represented by $c \log_2 n$ 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—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: to compute x^n when x and n are general integers typically takes time logarithmic in n (see equation (31.34) on page 934), and you must worry about whether the result fits into a computer word. If n is an exact power of 2, however, exponentiation can usually be viewed as a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by n positions to the left. In most computers, shifting the bits of an integer by 1 position to the left is equivalent to multiplying by 2, so that shifting the bits by n positions to the left is equivalent to multiplying by 2^n. Therefore, such computers can compute 2^n in 1 constant-time instruction by shifting the integer 1 by n positions to the left, as long as n is no more than the number of bits in a computer word. We'll try to avoid such gray areas in the RAM model and treat computing 2^n and multiplying by 2^n as constant-time operations when the result is small enough to fit in a computer word.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. Several other computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. Section 11.5 and a handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book do 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.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Analysis of insertion sort

How long does the INSERTION-SORT procedure take? One way to tell would be for you to run it on your computer and time how long it takes to run. Of course, you'd first have to implement it in a real programming language, since you cannot run our pseudocode directly. What would such a timing test tell you? You would find out how long insertion sort takes to run on your particular computer, on that particular input, under the particular implementation that you created, with the particular compiler or interpreter that you ran, with the particular libraries that you linked in, and with the particular background tasks that were running on your computer concurrently with your timing test (such as checking for incoming information over a network). If you run insertion sort again on your computer with the same input, you might even get a different timing result. From running just one implementation of insertion sort on just one computer and on just one input, what would you be able to determine about insertion sort's running time if you were to give it a different input, if you were to run it on a different computer, or if you were to implement it in a different programming language? Not much. We need a way to predict, given a new input, how long insertion sort will take.

Instead of timing a run, or even several runs, of insertion sort, we can determine how long it takes by analyzing the algorithm itself. We'll examine how many times it executes each line of pseudocode and how long each line of pseudocode takes to run. We'll first come up with a precise but complicated formula for the running time. Then, we'll distill the important part of the formula using a convenient notation that can help us compare the running times of different algorithms for the same problem.

How do we analyze insertion sort? First, let's acknowledge that the running time depends on the input. You shouldn't be terribly surprised that sorting a thousand numbers takes longer than sorting three numbers. Moreover, insertion sort can take different amounts of time to sort two input arrays of the same size, depending on how nearly sorted they already are. Even though the running time can depend on many features of the input, we'll focus on the one that has been shown to have the greatest effect, namely the size of the input, and 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 "input size" more carefully. We also need to be clear about whether we are discussing the running time for an input that elicits the worst-case behavior, the best-case behavior, or some other case.

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 number n of items being sorted. 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 more than just one number. For example, if the input to an algorithm is a graph, we usually characterize the input size by both the number of vertices and the number of edges in the graph. We'll 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 instructions and data accesses executed. How we account for these costs should be independent of any particular computer, but within the framework of the RAM model. 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 might take more or less time than another line, but we'll assume that each execution of the kth line takes c_k time, where c_k 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.[10]

Let's analyze the INSERTION-SORT procedure. As promised, we'll start by devising a precise formula that uses the input size and all the statement costs c_k. This formula turns out to be messy, however. We'll then switch to a simpler notation that is more concise and easier to use. This simpler notation makes clear how to compare the running times of algorithms, especially as the size of the input increases.

To analyze the INSERTION-SORT procedure, let's view it on the following page with the time cost of each statement and the number of times each statement is executed. For each $i = 2, 3, \ldots, n$, let t_i denote the number of times the **while** loop test in line 5 is executed for that value of i. When a **for** or **while** loop exits in the usual way—because the test in the loop header comes up FALSE—the test is executed one time more than the loop body. Because comments are not executable statements, assume that they take no time.

The running time of the algorithm is the sum of running times for each statement executed. A statement that takes c_k steps to execute and executes m times contributes $c_k m$ to the total running time.[11] We usually denote the running time of an algorithm on an input of size n by $T(n)$. 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

[10] 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, in the RADIX-SORT procedure on page 213, one line reads "use a stable sort to sort array A on digit i," which, as we shall see, takes more than a constant amount of time. Also, although a statement that calls a subroutine takes only constant time, the subroutine itself, 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.

[11] 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.

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

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

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 when the array is already sorted. In this case, each time that line 5 executes, the value of *key*—the value originally in $A[i]$—is already greater than or equal to all values in $A[1:i-1]$, so that the **while** loop of lines 5–7 always exits upon the first test in line 5. Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned} \quad (2.1)$$

We can express this running time as $an + b$ for *constants* a and b that depend on the statement costs c_k (where $a = c_1 + c_2 + c_4 + c_5 + c_8$ and $b = c_2 + c_4 + c_5 + c_8$). The running time is thus a ***linear function*** of n.

The worst case arises when the array is in reverse sorted order—that is, it starts out in decreasing order. The procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1:i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. (The procedure finds that $A[j] > key$ every time in line 5, and the **while** loop exits only when j reaches 0.) Noting that

$$\begin{aligned} \sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i \right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \quad \text{(by equation (A.2) on page 1141)} \end{aligned}$$

and

$$\begin{aligned}\sum_{i=2}^{n}(i-1) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \quad \text{(again, by equation (A.2))} ,\end{aligned}$$

we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \qquad (2.2)\end{aligned}$$

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_k (now, $a = c_5/2 + c_6/2 + c_7/2$, $b = c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8$, and $c = -(c_2 + c_4 + c_5 + c_8)$). The running time 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 we'll also see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

Our analysis of insertion sort 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'll usually (but not always) concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. Why? Here are three reasons:

- The worst-case running time of an algorithm gives an upper bound on the running time for *any* input. If you know it, then you have a guarantee that the algorithm never takes any longer. You need not make some educated guess about the running time and hope that it never gets much worse. This feature is especially important for real-time computing, in which operations must complete by a deadline.
- 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 often occurs 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 you run insertion sort on an array of n randomly chosen numbers. How long does it take to determine where in subarray $A[1:i-1]$ to insert element $A[i]$? On average, half the elements in $A[1:i-1]$ are less than $A[i]$, and half the elements are greater. On average, therefore, $A[i]$ is compared with just half of the subarray $A[1:i-1]$, and so t_i is about $i/2$. 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'll be interested in the ***average-case*** running time of an algorithm. We'll 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'll 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

In order to ease our analysis of the INSERTION-SORT procedure, we used some simplifying abstractions. First, we ignored the actual cost of each statement, using the constants c_k to represent these costs. Still, the best-case and worst-case running times in equations (2.1) and (2.2) are rather unwieldy. The constants in these expressions give us more detail than we really need. That's why we also expressed the best-case running time as $an + b$ for constants a and b that depend on the statement costs c_k and why we expressed the worst-case running time as $an^2 + bn + c$ for constants a, b, and c that depend on the statement costs. We thus ignored not only the actual statement costs, but also the abstract costs c_k.

Let's 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's worst-case running time, when we ignore the lower-order terms and the leading term's constant coefficient, only the factor of n^2 from the leading term remains. That factor, n^2, is by far the most important part of the running time. For example, suppose that an algorithm implemented on a particular machine takes $n^2/100 + 100n + 17$ microseconds on an input of size n. Although the coefficients of $1/100$ for the n^2 term and 100 for the n term differ by four orders of magnitude, the $n^2/100$ term dominates the $100n$ term once n exceeds 10,000. Although 10,000 might seem large, it is smaller than the population of an average town. Many real-world problems have much larger input sizes.

To highlight the order of growth of the running time, we have a special notation that uses the Greek letter Θ (theta). We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared" or just "theta n-squared"). We also write that insertion sort has a best-case running time of $\Theta(n)$ ("theta of n" or "theta n"). For now, think of Θ-notation as saying "roughly proportional when n is large," so that $\Theta(n^2)$ means "roughly proportional to n^2 when n is large" and $\Theta(n)$ means "roughly proportional to n when n is large" We'll use Θ-notation informally in this chapter and 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 on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, for example, takes less time in the worst case than an algorithm whose worst-case running time is $\Theta(n^3)$. Regardless of the constants hidden by the Θ-notation, there is always some number, say n_0, such that for all input sizes $n \geq n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.

Exercises

2.2-1

Express the function $n^3/1000 + 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2

Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. 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 worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

2.2-3

Consider linear search again (see Exercise 2.1-4). How many elements of the input array 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?

Using Θ-notation, give the average-case and worst-case running times of linear search. Justify your answers.

2.2-4
How can you modify any sorting algorithm to have a good best-case running time?

2.3 Designing algorithms

You can choose from a wide range of algorithm design techniques. Insertion sort uses the ***incremental*** method: for each element $A[i]$, insert it into its proper place in the subarray $A[1:i]$, having already sorted the subarray $A[1:i-1]$.

This section examines another design method, known as "divide-and-conquer," which we 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 using an algorithm that follows the divide-and-conquer method is that analyzing its running time is often straightforward, using techniques that we'll explore in Chapter 4.

2.3.1 The divide-and-conquer method

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they ***recurse*** (call themselves) one or more times to handle closely related subproblems. These algorithms typically follow the ***divide-and-conquer*** method: 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.

In the divide-and-conquer method, if the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer method. In each step, it sorts a subarray $A[p:r]$, starting with the entire array $A[1:n]$ and recursing down to smaller and smaller subarrays. Here is how merge sort operates:

Divide the subarray $A[p:r]$ to be sorted into two adjacent subarrays, each of half the size. To do so, compute the midpoint q of $A[p:r]$ (taking the average of p and r), and divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$.

Conquer by sorting each of the two subarrays $A[p:q]$ and $A[q+1:r]$ recursively using merge sort.

Combine by merging the two sorted subarrays $A[p:q]$ and $A[q+1:r]$ back into $A[p:r]$, producing the sorted answer.

The recursion "bottoms out"—it reaches the base case—when the subarray $A[p:r]$ to be sorted has just 1 element, that is, when p equals r. As we noted in the initialization argument for INSERTION-SORT's loop invariant, a subarray comprising just a single element is always sorted.

The key operation of the merge sort algorithm occurs in the "combine" step, which merges two adjacent, sorted subarrays. The merge operation is performed by the auxiliary procedure MERGE(A, p, q, r) on the following page, where A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The procedure assumes that the adjacent subarrays $A[p:q]$ and $A[q+1:r]$ were already recursively sorted. It ***merges*** the two sorted subarrays to form a single sorted subarray that replaces the current subarray $A[p:r]$.

To understand how the MERGE procedure works, let's return to our card-playing motif. Suppose that you have two piles of cards face up on a table. Each pile is sorted, with the smallest-value cards on top. You wish to merge the two piles into a single sorted output pile, which is to be face down on the table. The 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. Repeat this step until one input pile is empty, at which time you can just take the remaining input pile and flip over the entire pile, placing it face down onto the output pile.

Let's think about how long it takes to merge two sorted piles of cards. Each basic step takes constant time, since you are comparing just the two top cards. If the two sorted piles that you start with each have $n/2$ cards, then the number of basic steps is at least $n/2$ (since in whichever pile was emptied, every card was found to be smaller than some card from the other pile) and at most n (actually, at most $n-1$, since after $n-1$ basic steps, one of the piles must be empty). With each basic step taking constant time and the total number of basic steps being between $n/2$ and n, we can say that merging takes time roughly proportional to n. That is, merging takes $\Theta(n)$ time.

In detail, the MERGE procedure works as follows. It copies the two subarrays $A[p:q]$ and $A[q+1:r]$ into temporary arrays L and R ("left" and "right"), and then it merges the values in L and R back into $A[p:r]$. Lines 1 and 2 compute the lengths n_L and n_R of the subarrays $A[p:q]$ and $A[q+1:r]$, respectively. Then

```
MERGE($A, p, q, r$)
 1  $n_L = q - p + 1$           // length of $A[p:q]$
 2  $n_R = r - q$               // length of $A[q+1:r]$
 3  let $L[0:n_L - 1]$ and $R[0:n_R - 1]$ be new arrays
 4  for $i = 0$ to $n_L - 1$    // copy $A[p:q]$ into $L[0:n_L - 1]$
 5      $L[i] = A[p + i]$
 6  for $j = 0$ to $n_R - 1$    // copy $A[q+1:r]$ into $R[0:n_R - 1]$
 7      $R[j] = A[q + j + 1]$
 8  $i = 0$                     // $i$ indexes the smallest remaining element in $L$
 9  $j = 0$                     // $j$ indexes the smallest remaining element in $R$
10  $k = p$                     // $k$ indexes the location in $A$ to fill
11  // As long as each of the arrays $L$ and $R$ contains an unmerged element,
    //        copy the smallest unmerged element back into $A[p:r]$.
12  while $i < n_L$ and $j < n_R$
13      if $L[i] \le R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16      else $A[k] = R[j]$
17          $j = j + 1$
18      $k = k + 1$
19  // Having gone through one of $L$ and $R$ entirely, copy the
    //        remainder of the other to the end of $A[p:r]$.
20  while $i < n_L$
21      $A[k] = L[i]$
22      $i = i + 1$
23      $k = k + 1$
24  while $j < n_R$
25      $A[k] = R[j]$
26      $j = j + 1$
27      $k = k + 1$
```

line 3 creates arrays $L[0:n_L - 1]$ and $R[0:n_R - 1]$ with respective lengths n_L and n_R.[12] The **for** loop of lines 4–5 copies the subarray $A[p:q]$ into L, and the **for** loop of lines 6–7 copies the subarray $A[q+1:r]$ into R.

Lines 8–18, illustrated in Figure 2.3, perform the basic steps. The **while** loop of lines 12–18 repeatedly identifies the smallest value in L and R that has yet to

[12] This procedure is the rare case that uses both 1-origin indexing (for array A) and 0-origin indexing (for arrays L and R). Using 0-origin indexing for L and R makes for a simpler loop invariant in Exercise 2.3-3.

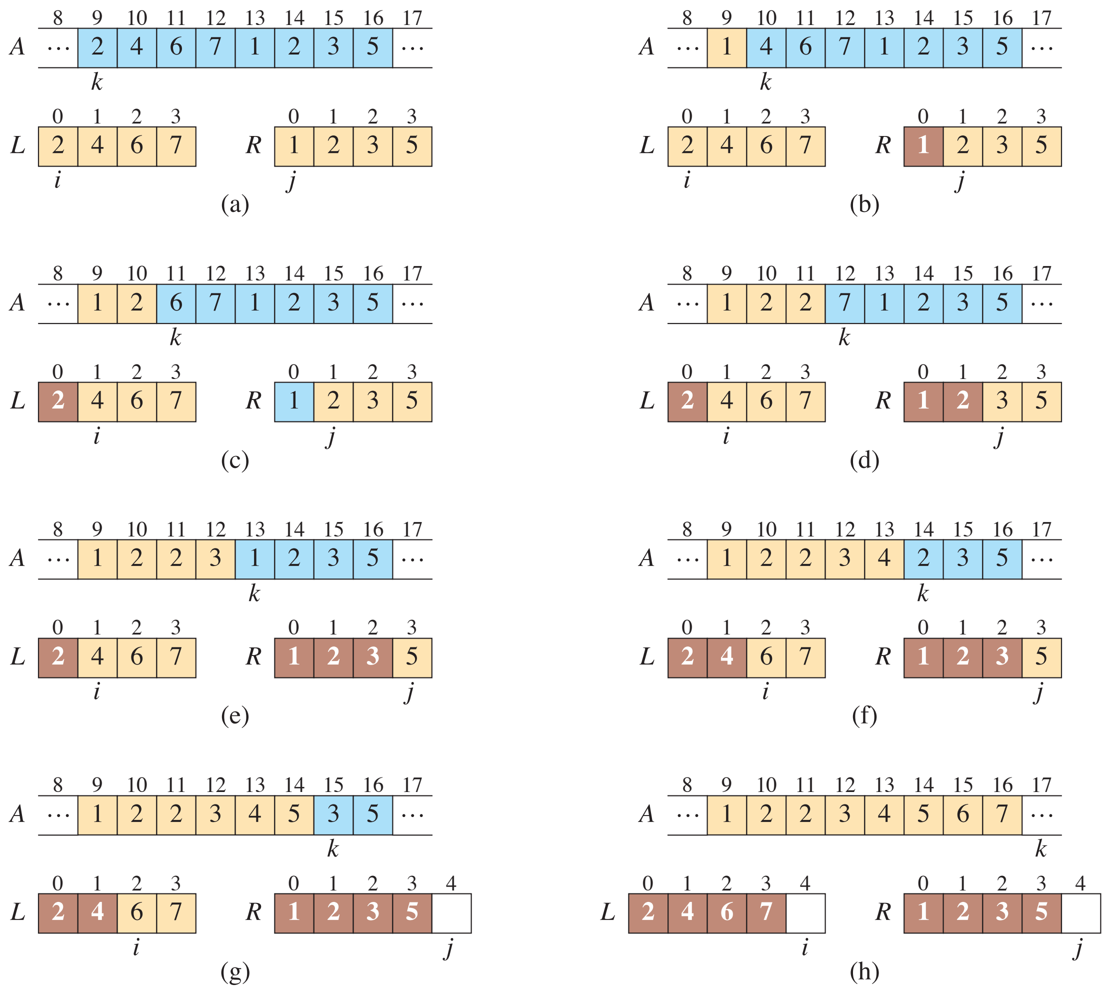


Figure 2.3 The operation of the **while** loop in lines 8–18 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9:16]$ contains the values $\langle 2, 4, 6, 7, 1, 2, 3, 5\rangle$. After allocating and copying into the arrays L and R, the array L contains $\langle 2, 4, 6, 7\rangle$, and the array R contains $\langle 1, 2, 3, 5\rangle$. Tan positions in A contain their final values, and tan positions in L and R contain values that have yet to be copied back into A. Taken together, the tan positions always comprise the values originally in $A[9:16]$. Blue positions in A contain values that will be copied over, and dark positions in L and R contain values that have already been copied back into A. **(a)–(g)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–18. At the point in part (g), all values in R have been copied back into A (indicated by j equaling the length of R), and so the **while** loop in lines 12–18 terminates. **(h)** The arrays and indices at termination. The **while** loops of lines 20–23 and 24–27 copied back into A the remaining values in L and R, which are the largest values originally in $A[9:16]$. Here, lines 20–23 copied $L[2:3]$ into $A[15:16]$, and because all values in R had already been copied back into A, the **while** loop of lines 24–27 iterated 0 times. At this point, the subarray in $A[9:16]$ is sorted.

be copied back into $A[p:r]$ and copies it back in. As the comments indicate, the index k gives the position of A that is being filled in, and the indices i and j give the positions in L and R, respectively, of the smallest remaining values. Eventually, either all of L or all of R is copied back into $A[p:r]$, and this loop terminates. If the loop terminates because all of R has been copied back, that is, because j equals n_R, then i is still less than n_L, so that some of L has yet to be copied back, and these values are the greatest in both L and R. In this case, the **while** loop of lines 20–23 copies these remaining values of L into the last few positions of $A[p:r]$. Because j equals n_R, the **while** loop of lines 24–27 iterates 0 times. If instead the **while** loop of lines 12–18 terminates because i equals n_L, then all of L has already been copied back into $A[p:r]$, and the **while** loop of lines 24–27 copies the remaining values of R back into the end of $A[p:r]$.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$,[13] observe that each of lines 1–3 and 8–10 takes constant time, and the **for** loops of lines 4–7 take $\Theta(n_L + n_R) = \Theta(n)$ time.[14] To account for the three **while** loops of lines 12–18, 20–23, and 24–27, observe that each iteration of these loops copies exactly one value from L or R back into A and that every value is copied back into A exactly once. Therefore, these three loops together make a total of n iterations. Since each iteration of each of the three loops takes constant time, the total time spent in these three loops is $\Theta(n)$.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) on the facing page sorts the elements in the subarray $A[p:r]$. If p equals r, the subarray has just 1 element and is therefore already sorted. Otherwise, we must have $p < r$, and MERGE-SORT runs the divide, conquer, and combine steps. The divide step simply computes an index q that partitions $A[p:r]$ into two adjacent subarrays: $A[p:q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1:r]$, containing $\lfloor n/2 \rfloor$ elements.[15] The initial call MERGE-SORT$(A, 1, n)$ sorts the entire array $A[1:n]$.

Figure 2.4 illustrates the operation of the procedure for $n = 8$, showing also the sequence of divide and merge steps. The algorithm recursively divides the array down to 1-element subarrays. The combine steps merge pairs of 1-element subar-

[13] If you're wondering where the "+1" comes from, imagine that $r = p + 1$. Then the subarray $A[p:r]$ consists of two elements, and $r - p + 1 = 2$.

[14] Chapter 3 shows how to formally interpret equations containing Θ-notation.

[15] 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 Section 3.3. 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.

```
MERGE-SORT($A, p, r$)
1  if $p \geq r$                              // zero or one element?
2      return
3  $q = \lfloor (p + r)/2 \rfloor$                     // midpoint of $A[p:r]$
4  MERGE-SORT($A, p, q$)                   // recursively sort $A[p:q]$
5  MERGE-SORT($A, q + 1, r$)               // recursively sort $A[q + 1:r]$
6  // Merge $A[p:q]$ and $A[q + 1:r]$ into $A[p:r]$.
7  MERGE($A, p, q, r$)
```

rays to form sorted subarrays of length 2, merges those to form sorted subarrays of length 4, and merges those to form the final sorted subarray of length 8. If n is not an exact power of 2, then some divide steps create subarrays whose lengths differ by 1. (For example, when dividing a subarray of length 7, one subarray has length 4 and the other has length 3.) Regardless of the lengths of the two subarrays being merged, the time to merge a total of n items is $\Theta(n)$.

2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call, you 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 of the same algorithm on smaller inputs. You can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic method. As we did for insertion sort, let $T(n)$ be the worst-case running time on a problem of size n. If the problem size is small enough, say $n < n_0$ for some constant $n_0 > 0$, the straightforward solution takes constant time, which we write as $\Theta(1)$.[16] Suppose that the division of the problem yields a subproblems, each with size n/b, that is, $1/b$ the size of the original. For merge sort, both a and b are 2, but we'll see other divide-and-conquer algorithms in which $a \neq b$. It takes $T(n/b)$ time to solve one subproblem of size n/b, and so it takes $aT(n/b)$ time to solve all a of them. If it takes $D(n)$ 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

[16] If you're wondering where $\Theta(1)$ comes from, think of it this way. When we say that $n^2/100$ is $\Theta(n^2)$, we are ignoring the coefficient $1/100$ of the factor n^2. Likewise, when we say that a constant c is $\Theta(1)$, we are ignoring the coefficient c of the factor 1 (which you can also think of as n^0).

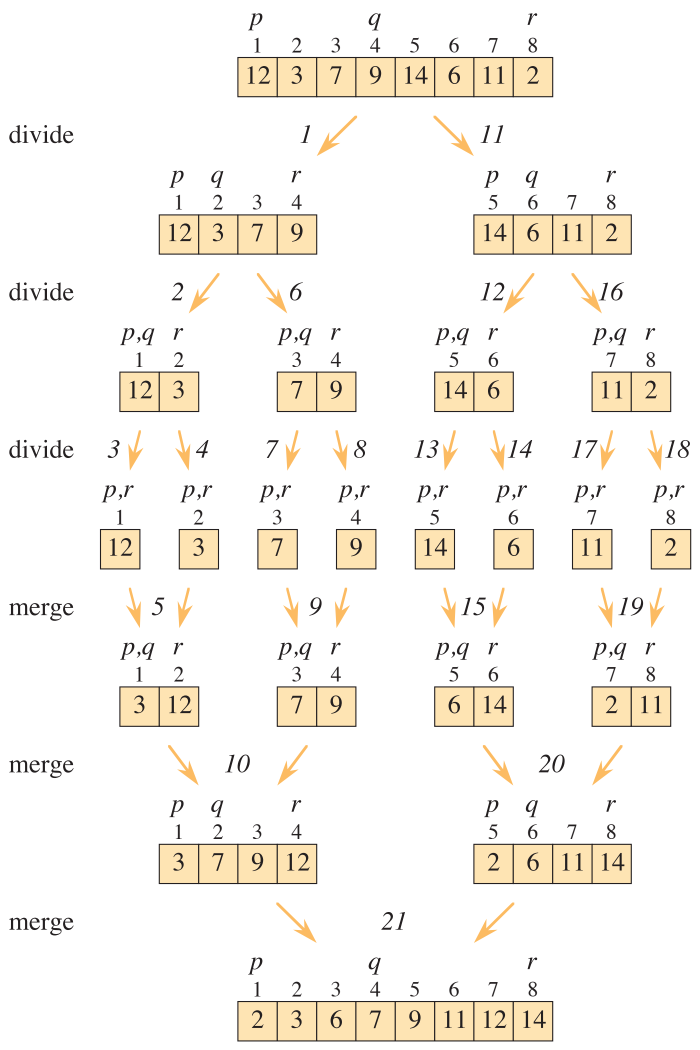


Figure 2.4 The operation of merge sort on the array A with length 8 that initially contains the sequence $\langle 12, 3, 7, 9, 14, 6, 11, 2 \rangle$. The indices p, q, and r into each subarray appear above their values. Numbers in italics indicate the order in which the MERGE-SORT and MERGE procedures are called following the initial call of MERGE-SORT$(A, 1, 8)$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 , \\ D(n) + aT(n/b) + C(n) & \text{otherwise} . \end{cases}$$

Chapter 4 shows how to solve common recurrences of this form.

Sometimes, the n/b size of the divide step isn't an integer. For example, the MERGE-SORT procedure divides a problem of size n into subproblems of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Since the difference between $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ is at most 1, which for large n is much smaller than the effect of dividing n by 2, we'll squint a little and just call them both size $n/2$. As Chapter 4 will discuss, this simplification of ignoring floors and ceilings does not generally affect the order of growth of a solution to a divide-and-conquer recurrence.

Another convention we'll adopt is to omit a statement of the base cases of the recurrence, which we'll also discuss in more detail in Chapter 4. The reason is that the base cases are pretty much always $T(n) = \Theta(1)$ if $n < n_0$ for some constant $n_0 > 0$. That's because the running time of an algorithm on an input of constant size is constant. We save ourselves a lot of extra writing by adopting this convention.

Analysis of merge sort

Here's how to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

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

Conquer: Recursively solving two subproblems, each of size $n/2$, contributes $2T(n/2)$ to the running time (ignoring the floors and ceilings, as we discussed).

Combine: Since the MERGE procedure on an n-element subarray takes $\Theta(n)$ time, we have $C(n) = \Theta(n)$.

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, it is roughly proportional to n when n is large, and so merge sort's dividing and combining times together are $\Theta(n)$. Adding $\Theta(n)$ to the $2T(n/2)$ term from the conquer step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = 2T(n/2) + \Theta(n) . \tag{2.3}$$

Chapter 4 presents the "master theorem," which shows that $T(n) = \Theta(n \lg n)$.[17] Compared with insertion sort, whose worst-case running time is $\Theta(n^2)$, merge sort trades away a factor of n for a factor of $\lg n$. Because the logarithm function grows more slowly than any linear function, that's a good trade. For large enough inputs, merge sort, with its $\Theta(n \lg n)$ worst-case running time, outperforms insertion sort, whose worst-case running time is $\Theta(n^2)$.

[17] The notation $\lg n$ stands for $\log_2 n$, although the base of the logarithm doesn't matter here, but as computer scientists, we like logarithms base 2. Section 3.3 discusses other standard notation.

We do not need the master theorem, however, to understand intuitively why the solution to recurrence (2.3) is $T(n) = \Theta(n \lg n)$. For simplicity, assume that n is an exact power of 2 and that the implicit base case is $n = 1$. Then recurrence (2.3) is essentially

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 , \end{cases} \tag{2.4}$$

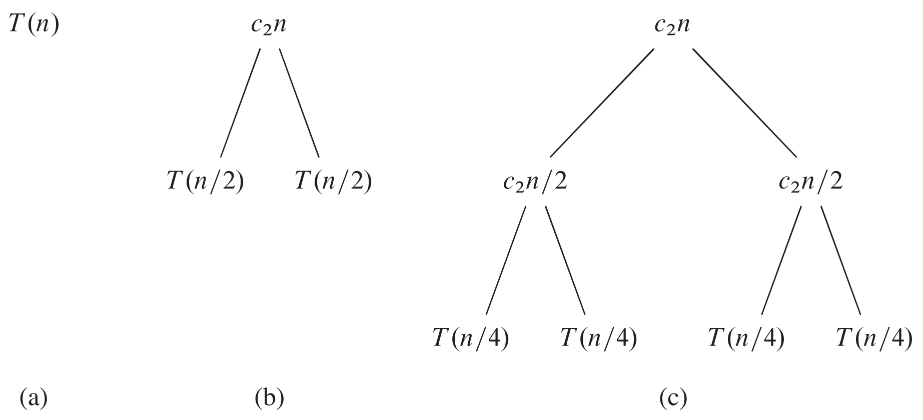
where the constant $c_1 > 0$ represents the time required to solve a problem of size 1, and $c_2 > 0$ is the time per array element of the divide and combine steps.[18]

Figure 2.5 illustrates one way of figuring out the solution to recurrence (2.4). Part (a) of the figure shows $T(n)$, which part (b) expands into an equivalent tree representing the recurrence. The $c_2 n$ term denotes the cost of dividing and combining 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 for dividing and combining at each of the two nodes at the second level of recursion is $c_2 n/2$. Continue to expand 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_1. Part (d) shows the resulting ***recursion tree***.

Next, add the costs across each level of the tree. The top level has total cost $c_2 n$, the next level down has total cost $c_2(n/2) + c_2(n/2) = c_2 n$, the level after that has total cost $c_2(n/4) + c_2(n/4) + c_2(n/4) + c_2(n/4) = c_2 n$, and so on. Each level has twice as many nodes as the level above, but each node contributes only half the cost of a node from the level above. From one level to the next, doubling and halving cancel each other out, so that the cost across each level is the same: $c_2 n$. In general, the level that is i levels below the top has 2^i nodes, each contributing a cost of $c_2(n/2^i)$, so that the ith level below the top has total cost $2^i \cdot c_2(n/2^i) = c_2 n$. The bottom level has n nodes, each contributing a cost of c_1, for a total cost of $c_1 n$.

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 1 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 assume that the input size is an exact power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has 1 more

[18] It is unlikely that c_1 is exactly the time to solve problems of size 1 and that $c_2 n$ is exactly the time of the divide and combine steps. We'll look more closely at bounding recurrences in Chapter 4, where we'll be more careful about this kind of detail.



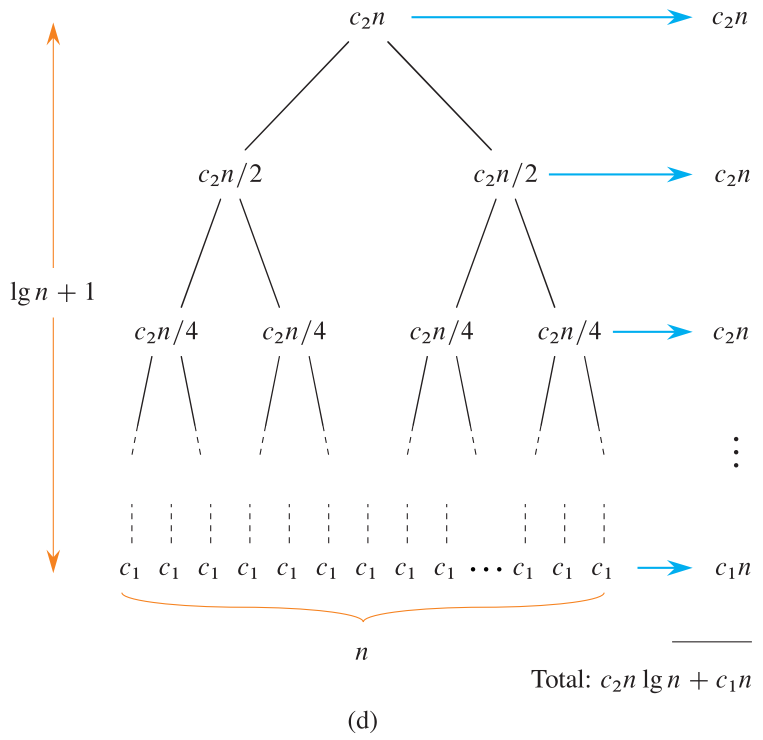


Figure 2.5 How to construct a recursion tree for the recurrence (2.4). Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels. Each level above the leaves contributes a total cost of c_2n, and the leaf level contributes c_1n. The total cost, therefore, is $c_2n \lg n + c_1n = \Theta(n \lg n)$.

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 (2.4), simply add up the costs of all the levels. The recursion tree has $\lg n + 1$ levels. The levels above the leaves each cost $c_2 n$, and the leaf level costs $c_1 n$, for a total cost of $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.

Exercises

2.3-1
Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence $\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3-2
The test in line 1 of the MERGE-SORT procedure reads "**if** $p \geq r$" rather than "**if** $p \neq r$." If MERGE-SORT is called with $p > r$, then the subarray $A[p:r]$ is empty. Argue that as long as the initial call of MERGE-SORT$(A, 1, n)$ has $n \geq 1$, the test "**if** $p \neq r$" suffices to ensure that no recursive call has $p > r$.

2.3-3
State a loop invariant for the **while** loop of lines 12–18 of the MERGE procedure. Show how to use it, along with the **while** loops of lines 20–23 and 24–27, to prove that the MERGE procedure is correct.

2.3-4
Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

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

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

2.3-5
You can also think of insertion sort as a recursive algorithm. In order to sort $A[1:n]$, recursively sort the subarray $A[1:n-1]$ and then insert $A[n]$ into the sorted subarray $A[1:n-1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

2.3-6
Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further

consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

2.3-7
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 $A[1 : j - 1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

2.3-8
Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

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 Θ-notation?

d. How should you 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. The procedure BUBBLESORT sorts array $A[1:n]$.

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

a. Let A' denote the array A after BUBBLESORT(A, n) is executed. To prove that BUBBLESORT is correct, you need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n] . \tag{2.5}$$

In order to show that BUBBLESORT actually sorts, what else do you need to prove?

The next two parts prove inequality (2.5).

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 allows you to prove inequality (2.5). 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 with the running time of INSERTION-SORT?

2-3 *Correctness of Horner's rule*

You are given the coefficents $a_0, a_1, a_2, \ldots, a_n$ of a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n , \end{aligned}$$

and you want to evaluate this polynomial for a given value of x. ***Horner's rule*** says to evaluate the polynomial according to this parenthesization:

$$P(x) = a_0 + x\Big(a_1 + x\left(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots\right)\Big) .$$

The procedure HORNER implements Horner's rule to evaluate $P(x)$, given the coefficients $a_0, a_1, a_2, \ldots, a_n$ in an array $A[0:n]$ and the value of x.

```
HORNER(A, n, x)
1  p = 0
2  for i = n downto 0
3      p = A[i] + x · p
4  return p
```

a. In terms of Θ-notation, what is the running time of this procedure?

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 with HORNER?

c. Consider the following loop invariant for the procedure HORNER:

At the start of each iteration of the **for** loop of lines 2–3,

$$p = \sum_{k=0}^{n-(i+1)} A[k+i+1] \cdot x^k .$$

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, $p = \sum_{k=0}^{n} A[k] \cdot x^k$.

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* [259, 260, 261]. The first volume ushered in the modern study of computer algorithms with a focus on the analysis of running time. 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 Θ-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 [261] 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 subarrays 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 [200], who credits P. Naur with the first article in this field. Gries attributes loop invariants to R. W. Floyd. The textbook by Mitchell [329] is a good reference on how to prove programs correct.

3 Characterizing Running Times

The order of growth of the running time of an algorithm, defined in Chapter 2, gives a simple way to characterize the algorithm's efficiency and also allows us to compare it with 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 rarely 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 relevant only the order of growth of the running time, 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 is 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 presents informally the three most commonly used types of "asymptotic notation," of which we have already seen an example in Θ-notation. It also shows one way to use these asymptotic notations to reason about the worst-case running time of insertion sort. Then we look at asymptotic notations more formally and present several notational conventions used throughout this book. The last section reviews the behavior of functions that commonly arise when analyzing algorithms.

3.1 O-notation, Ω-notation, and Θ-notation

When we analyzed the worst-case running time of insertion sort in Chapter 2, we started with the complicated expression

$$\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n$$
$$-(c_2+c_4+c_5+c_8)\ .$$

We then discarded the lower-order terms $(c_1+c_2+c_4+c_5/2-c_6/2-c_7/2+c_8)n$ and $c_2+c_4+c_5+c_8$, and we also ignored the coefficient $c_5/2+c_6/2+c_7/2$ of n^2. That left just the factor n^2, which we put into Θ-notation as $\Theta(n^2)$. We use this style to characterize running times of algorithms: discard the lower-order terms and the coefficient of the leading term, and use a notation that focuses on the rate of growth of the running time.

Θ-notation is not the only such "asymptotic notation." In this section, we'll see other forms of asymptotic notation as well. We start with intuitive looks at these notations, revisiting insertion sort to see how we can apply them. In the next section, we'll see the formal definitions of our asymptotic notations, along with conventions for using them.

Before we get into specifics, bear in mind that the asymptotic notations we'll see are designed so that they characterize functions in general. It so happens that the functions we are most interested in denote 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.

O-notation

O-notation characterizes an *upper bound* on the asymptotic behavior of a function. In other words, it says that a function grows *no faster* than a certain rate, based on the highest-order term. Consider, for example, the function $7n^3+100n^2-20n+6$. Its highest-order term is $7n^3$, and so we say that this function's rate of growth is n^3. Because this function grows no faster than n^3, we can write that it is $O(n^3)$. You might be surprised that we can also write that the function $7n^3+100n^2-20n+6$ is $O(n^4)$. Why? Because the function grows more slowly than n^4, we are correct in saying that it grows no faster. As you might have guessed, this function is also $O(n^5)$, $O(n^6)$, and so on. More generally, it is $O(n^c)$ for any constant $c \geq 3$.

Ω-notation

Ω-notation characterizes a *lower bound* on the asymptotic behavior of a function. In other words, it says that a function grows *at least as fast* as a certain rate, based—as in O-notation—on the highest-order term. Because the highest-order term in the function $7n^3 + 100n^2 - 20n + 6$ grows at least as fast as n^3, this function is $\Omega(n^3)$. This function is also $\Omega(n^2)$ and $\Omega(n)$. More generally, it is $\Omega(n^c)$ for any constant $c \le 3$.

Θ-notation

Θ-notation characterizes a *tight bound* on the asymptotic behavior of a function. It says that a function grows *precisely* at a certain rate, based—once again—on the highest-order term. Put another way, Θ-notation characterizes the rate of growth of the function to within a constant factor from above and to within a constant factor from below. These two constant factors need not be equal.

If you can show that a function is both $O(f(n))$ and $\Omega(fn))$ for some function $f(n)$, then you have shown that the function is $\Theta(f(n))$. (The next section states this fact as a theorem.) For example, since the function $7n^3+100n^2-20n+6$ is both $O(n^3)$ and $\Omega(n^3)$, it is also $\Theta(n^3)$.

Example: Insertion sort

Let's revisit insertion sort and see how to work with asymptotic notation to characterize its $\Theta(n^2)$ worst-case running time without evaluating summations as we did in Chapter 2. Here is the INSERTION-SORT procedure once again:

```
INSERTION-SORT($A, n$)
for $i = 2$ to $n$
    $key = A[i]$
    // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.
    $j = i - 1$
    while $j > 0$ and $A[j] > key$
        $A[j+1] = A[j]$
        $j = j - 1$
    $A[j+1] = key$
```

What can we observe about how the pseudocode operates? The procedure has nested loops. The outer loop is a **for** loop that runs $n - 1$ times, regardless of the values being sorted. The inner loop is a **while** loop, but the number of iterations it makes depends on the values being sorted. The loop variable j starts at $i - 1$

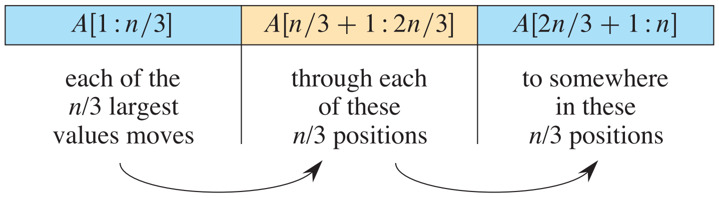

$A[1:n/3]$
$A[n/3+1:2n/3]$
$A[2n/3+1:n]$
each of the $n/3$ largest values moves
through each of these $n/3$ positions
to somewhere in these $n/3$ positions

Figure 3.1 The $\Omega(n^2)$ lower bound for insertion sort. If the first $n/3$ positions contain the $n/3$ largest values, each of these values must move through each of the middle $n/3$ positions, one position at a time, to end up somewhere in the last $n/3$ positions. Since each of $n/3$ values moves through at least each of $n/3$ positions, the time taken in this case is at least proportional to $(n/3)(n/3) = n^2/9$, or $\Omega(n^2)$.

and decreases by 1 in each iteration until either it reaches 0 or $A[j] \le key$. For a given value of i, the **while** loop might iterate 0 times, $i-1$ times, or anywhere in between. The body of the **while** loop (lines 6–7) takes constant time per iteration of the **while** loop.

These observations suffice to deduce an $O(n^2)$ running time for any case of INSERTION-SORT, giving us a blanket statement that covers all inputs. The running time is dominated by the inner loop. Because each of the $n-1$ iterations of the outer loop causes the inner loop to iterate at most $i-1$ times, and because i is at most n, the total number of iterations of the inner loop is at most $(n-1)(n-1)$, which is less than n^2. Since each iteration of the inner loop takes constant time, the total time spent in the inner loop is at most a constant times n^2, or $O(n^2)$.

With a little creativity, we can also see that the worst-case running time of INSERTION-SORT is $\Omega(n^2)$. By saying that the worst-case running time of an algorithm is $\Omega(n^2)$, we mean that for every input size n above a certain threshold, there is at least one input of size n for which the algorithm takes at least cn^2 time, for some positive constant c. It does not necessarily mean that the algorithm takes at least cn^2 time for all inputs.

Let's now see why the worst-case running time of INSERTION-SORT is $\Omega(n^2)$. For a value to end up to the right of where it started, it must have been moved in line 6. In fact, for a value to end up k positions to the right of where it started, line 6 must have executed k times. As Figure 3.1 shows, let's assume that n is a multiple of 3 so that we can divide the array A into groups of $n/3$ positions. Suppose that in the input to INSERTION-SORT, the $n/3$ largest values occupy the first $n/3$ array positions $A[1:n/3]$. (It does not matter what relative order they have within the first $n/3$ positions.) Once the array has been sorted, each of these $n/3$ values ends up somewhere in the last $n/3$ positions $A[2n/3+1:n]$. For that to happen, each of these $n/3$ values must pass through each of the middle $n/3$ positions $A[n/3+1:2n/3]$. Each of these $n/3$ values passes through these middle

$n/3$ positions one position at a time, by at least $n/3$ executions of line 6. Because at least $n/3$ values have to pass through at least $n/3$ positions, the time taken by INSERTION-SORT in the worst case is at least proportional to $(n/3)(n/3) = n^2/9$, which is $\Omega(n^2)$.

Because we have shown that INSERTION-SORT runs in $O(n^2)$ time in all cases and that there is an input that makes it take $\Omega(n^2)$ time, we can conclude that the worst-case running time of INSERTION-SORT is $\Theta(n^2)$. It does not matter that the constant factors for upper and lower bounds might differ. What matters is that we have characterized the worst-case running time to within constant factors (discounting lower-order terms). This argument does not show that INSERTION-SORT runs in $\Theta(n^2)$ time in *all* cases. Indeed, we saw in Chapter 2 that the best-case running time is $\Theta(n)$.

Exercises

3.1-1
Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

3.1-2
Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

3.1-3
Suppose that α is a fraction in the range $0 < \alpha < 1$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the αn largest values must pass through each of the middle $(1 - 2\alpha)n$ array positions?

3.2 Asymptotic notation: formal definitions

Having seen asymptotic notation informally, let's get more formal. The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are typically the set $\mathbb{N}$ of natural numbers or the set $\mathbb{R}$ of real numbers. Such notations are convenient for describing a running-time function $T(n)$. This section defines the basic asymptotic notations and also introduces some common "proper" notational abuses.

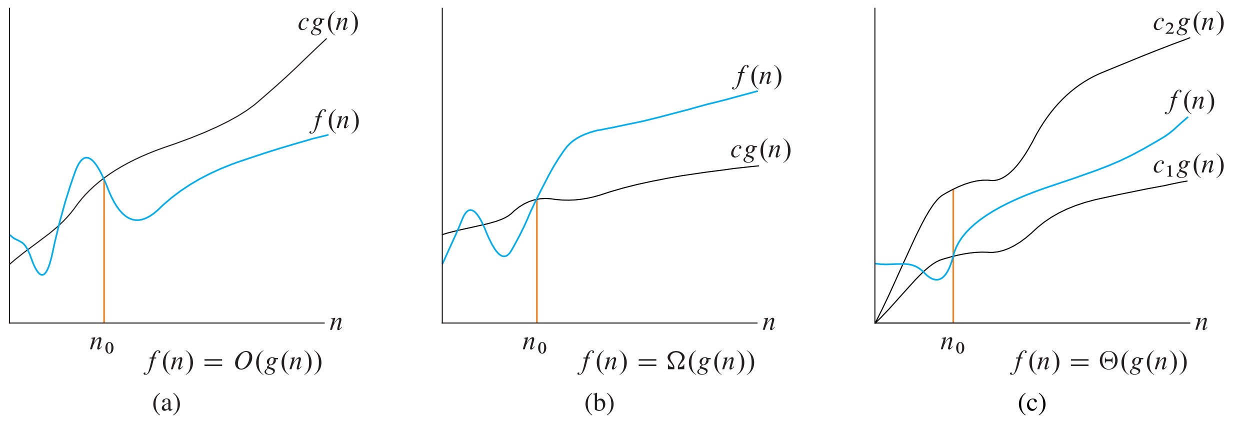


Figure 3.2 Graphic examples of the O, Ω, and Θ notations. In each part, the value of n_0 shown is the minimum possible value, but any greater value also works. **(a)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(b)** Ω-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 $cg(n)$. **(c)** Θ-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 $c_1g(n)$ and $c_2g(n)$ inclusive.

O-notation

As we saw in Section 3.1, O-notation describes an ***asymptotic upper bound***. We use O-notation to give an upper bound on a function, to within a constant factor.

Here is the formal definition of 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*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$[1]

A function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that $f(n) \le cg(n)$ for sufficiently large n. Figure 3.2(a) 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 $cg(n)$.

The definition of $O(g(n))$ requires that every function $f(n)$ in the set $O(g(n))$ be ***asymptotically nonnegative***: $f(n)$ must be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all

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

sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $O(g(n))$ is empty. We therefore assume that every function used within O-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

You might be surprised that we define O-notation in terms of sets. Indeed, you might expect that we would write "$f(n) \in O(g(n))$" to indicate that $f(n)$ belongs to the set $O(g(n))$. Instead, we usually write "$f(n) = O(g(n))$" and say "$f(n)$ is big-oh of $g(n)$" to express the same notion. Although it may seem confusing at first to abuse equality in this way, we'll see later in this section that doing so has its advantages.

Let's explore an example of how to use the formal definition of O-notation to justify our practice of discarding lower-order terms and ignoring the constant coefficient of the highest-order term. We'll show that $4n^2 + 100n + 500 = O(n^2)$, even though the lower-order terms have much larger coefficients than the leading term. We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \le cn^2$ for all $n \ge n_0$. Dividing both sides by n^2 gives $4 + 100/n + 500/n^2 \le c$. This inequality is satisfied for many choices of c and n_0. For example, if we choose $n_0 = 1$, then this inequality holds for $c = 604$. If we choose $n_0 = 10$, then $c = 19$ works, and choosing $n_0 = 100$ allows us to use $c = 5.05$.

We can also use the formal definition of O-notation to show that the function $n^3 - 100n^2$ does not belong to the set $O(n^2)$, even though the coefficient of n^2 is a large negative number. If we had $n^3 - 100n^2 = O(n^2)$, then there would be positive constants c and n_0 such that $n^3 - 100n^2 \le cn^2$ for all $n \ge n_0$. Again, we divide both sides by n^2, giving $n - 100 \le c$. Regardless of what value we choose for the constant c, this inequality does not hold for any value of $n > c + 100$.

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-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

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

Figure 3.2(b) shows the intuition behind Ω-notation. For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

We've already shown that $4n^2 + 100n + 500 = O(n^2)$. Now let's show that $4n^2 + 100n + 500 = \Omega(n^2)$. We need to find positive constants c and n_0 such that $4n^2 + 100n + 500 \ge cn^2$ for all $n \ge n_0$. As before, we divide both sides by n^2, giving $4 + 100/n + 500/n^2 \geq c$. This inequality holds when n_0 is any positive integer and $c = 4$.

What if we had subtracted the lower-order terms from the $4n^2$ term instead of adding them? What if we had a small coefficient for the n^2 term? The function would still be $\Omega(n^2)$. For example, let's show that $n^2/100 - 100n - 500 = \Omega(n^2)$. Dividing by n^2 gives $1/100 - 100/n - 500/n^2 \geq c$. We can choose any value for n_0 that is at least 10,005 and find a positive value for c. For example, when $n_0 = 10{,}005$, we can choose $c = 2.49 \times 10^{-9}$. Yes, that's a tiny value for c, but it is positive. If we select a larger value for n_0, we can also increase c. For example, if $n_0 = 100{,}000$, then we can choose $c = 0.0089$. The higher the value of n_0, the closer to the coefficient $1/100$ we can choose c.

Θ-notation

We use Θ-notation for ***asymptotically tight bounds***. For a given function $g(n)$, we denote by $\Theta(g(n))$ ("theta of g of n") the set of functions

$$\begin{aligned}\Theta(g(n)) = \{f(n) :\ & \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that} \\ & 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\} .\end{aligned}$$

Figure 3.2(c) shows the intuition behind Θ-notation. 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 constant factors.

The definitions of O-, Ω-, and Θ-notations lead to the following theorem, whose proof we leave as Exercise 3.2-4.

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 $f(n) = \Omega(g(n))$. ■

We typically apply Theorem 3.1 to prove asymptotically tight bounds from asymptotic upper and lower bounds.

Asymptotic notation and running times

When you use asymptotic notation to characterize an algorithm's running time, make sure that the asymptotic notation you use is as precise as possible without overstating which running time it applies to. Here are some examples of using asymptotic notation properly and improperly to characterize running times.

Let's start with insertion sort. We can correctly say that insertion sort's worst-case running time is $O(n^2)$, $\Omega(n^2)$, and—due to Theorem 3.1—$\Theta(n^2)$. Although all three ways to characterize the worst-case running times are correct, the $\Theta(n^2)$ bound is the most precise and hence the most preferred. We can also correctly say that insertion sort's best-case running time is $O(n)$, $\Omega(n)$, and $\Theta(n)$, again with $\Theta(n)$ the most precise and therefore the most preferred.

Here is what we *cannot* correctly say: insertion sort's running time is $\Theta(n^2)$. That is an overstatement because by omitting "worst-case" from the statement, we're left with a blanket statement covering all cases. The error here is that insertion sort does not run in $\Theta(n^2)$ time in all cases since, as we've seen, it runs in $\Theta(n)$ time in the best case. We can correctly say that insertion sort's running time is $O(n^2)$, however, because in all cases, its running time grows no faster than n^2. When we say $O(n^2)$ instead of $\Theta(n^2)$, there is no problem in having cases whose running time grows more slowly than n^2. Likewise, we cannot correctly say that insertion sort's running time is $\Theta(n)$, but we can say that its running time is $\Omega(n)$.

How about merge sort? Since merge sort runs in $\Theta(n \lg n)$ time in all cases, we can just say that its running time is $\Theta(n \lg n)$ without specifying worst-case, best-case, or any other case.

People occasionally conflate O-notation with Θ-notation by mistakenly using O-notation to indicate an asymptotically tight bound. They say things like "an $O(n \lg n)$-time algorithm runs faster than an $O(n^2)$-time algorithm." Maybe it does, maybe it doesn't. Since O-notation denotes only an asymptotic upper bound, that so-called $O(n^2)$-time algorithm might actually run in $\Theta(n)$ time. You should be careful to choose the appropriate asymptotic notation. If you want to indicate an asymptotically tight bound, use Θ-notation.

We typically use asymptotic notation to provide the simplest and most precise bounds possible. For example, if an algorithm has a running time of $3n^2 + 20n$ in all cases, we use asymptotic notation to write that its running time is $\Theta(n^2)$. Strictly speaking, we are also correct in writing that the running time is $O(n^3)$ or $\Theta(3n^2 + 20n)$. Neither of these expressions is as useful as writing $\Theta(n^2)$ in this case, however: $O(n^3)$ is less precise than $\Theta(n^2)$ if the running time is $3n^2 + 20n$, and $\Theta(3n^2 + 20n)$ introduces complexity that obscures the order of growth. By writing the simplest and most precise bound, such as $\Theta(n^2)$, we can categorize and compare different algorithms. Throughout the book, you will see asymptotic running times that are almost always based on polynomials and logarithms: functions such as n, $n \lg^2 n$, $n^2 \lg n$, or $n^{1/2}$. You will also see some other functions, such as exponentials, $\lg \lg n$, and $\lg^* n$ (see Section 3.3). It is usually fairly easy to compare the rates of growth of these functions. Problem 3-3 gives you good practice.

Asymptotic notation in equations and inequalities

Although we formally define asymptotic notation in terms of sets, we use the equal sign ($=$) instead of the set membership sign ($\in$) within formulas. For example, we wrote that $4n^2 + 100n + 500 = O(n^2)$. 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 $4n^2 + 100n + 500 = O(n^2)$, the equal sign means set membership: $4n^2 + 100n + 500 \in O(n^2)$. 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 $f(n) \in \Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed belongs to $\Theta(n)$.

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, because 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

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

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

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

By the rules above, interpret each equation separately. 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. This interpretation implies that $2n^2 + 3n + 1 = \Theta(n^2)$, which is what the chaining of equations intuitively says.

Proper abuses of asymptotic notation

Besides the abuse of equality to mean set membership, which we now see has a precise mathematical interpretation, another abuse of asymptotic notation occurs when the variable tending toward ∞ must be inferred from context. For example, when we say $O(g(n))$, we can assume that we're interested in the growth of $g(n)$ as n grows, and if we say $O(g(m))$ we're talking about the growth of $g(m)$ as m grows. The free variable in the expression indicates what variable is going to ∞.

The most common situation requiring contextual knowledge of which variable tends to ∞ occurs when the function inside the asymptotic notation is a constant, as in the expression $O(1)$. We cannot infer from the expression which variable is going to ∞, because no variable appears there. The context must disambiguate. For example, if the equation using asymptotic notation is $f(n) = O(1)$, it's apparent that the variable we're interested in is n. Knowing from context that the variable of interest is n, however, allows us to make perfect sense of the expression by using the formal definition of O-notation: the expression $f(n) = O(1)$ means that the function $f(n)$ is bounded from above by a constant as n goes to ∞. Technically, it might be less ambiguous if we explicitly indicated the variable tending to ∞ in the asymptotic notation itself, but that would clutter the notation. Instead, we simply ensure that the context makes it clear which variable (or variables) tend to ∞.

When the function inside the asymptotic notation is bounded by a positive constant, as in $T(n) = O(1)$, we often abuse asymptotic notation in yet another way, especially when stating recurrences. We may write something like $T(n) = O(1)$ for $n < 3$. According to the formal definition of O-notation, this statement is meaningless, because the definition only says that $T(n)$ is bounded above by a positive constant c for $n \geq n_0$ for some $n_0 > 0$. The value of $T(n)$ for $n < n_0$ need not be so bounded. Thus, in the example $T(n) = O(1)$ for $n < 3$, we cannot infer any constraint on $T(n)$ when $n < 3$, because it might be that $n_0 > 3$.

What is conventionally meant when we say $T(n) = O(1)$ for $n < 3$ is that there exists a positive constant c such that $T(n) \leq c$ for $n < 3$. This convention saves us the trouble of naming the bounding constant, allowing it to remain anonymous while we focus on more important variables in an analysis. Similar abuses occur with the other asymptotic notations. For example, $T(n) = \Theta(1)$ for $n < 3$ means that $T(n)$ is bounded above and below by positive constants when $n < 3$.

Occasionally, the function describing an algorithm's running time may not be defined for certain input sizes, for example, when an algorithm assumes that the input size is an exact power of 2. We still use asymptotic notation to describe the growth of the running time, understanding that any constraints apply only when the function is defined. For example, suppose that $f(n)$ is defined only on a subset of the natural or nonnegative real numbers. Then $f(n) = O(g(n))$ means that the bound $0 \le T(n) \le cg(n)$ in the definition of O-notation holds for all $n \ge n_0$ over the domain of $f(n)$, that is, where $f(n)$ is defined. This abuse is rarely pointed out, since what is meant is generally clear from context.

In mathematics, it's okay—and often desirable—to abuse a notation, as long as we don't misuse it. If we understand precisely what is meant by the abuse and don't draw incorrect conclusions, it can simplify our mathematical language, contribute to our higher-level understanding, and help us focus on what really matters.

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

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

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

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 gets large:

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

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

ω-notation

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

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

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

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

Where the definition of o-notation says that $f(n) < cg(n)$, the definition of ω-notation says the opposite: that $cg(n) < f(n)$. For examples of ω-notation, we have $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$. 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 gets large.

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:

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

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:

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

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:

$$\begin{aligned}
f(n) &= O(g(n)) && \text{is like} \quad a \le b\,, \\
f(n) &= \Omega(g(n)) && \text{is like} \quad a \ge b\,, \\
f(n) &= \Theta(g(n)) && \text{is like} \quad a = b\,, \\
f(n) &= o(g(n)) && \text{is like} \quad a < b\,, \\
f(n) &= \omega(g(n)) && \text{is like} \quad a > b\,.
\end{aligned}$$

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

3.2-1

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

3.2-2

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

3.2-3

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

3.2-4

Prove Theorem 3.1.

3.2-5
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 $\Omega(g(n))$.

3.2-6
Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

3.2-7
We can extend our notation to the case of two parameters n and m that can go to ∞ independently at different rates. For a given function $g(n,m)$, we denote by $O(g(n,m))$ the set of functions

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

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

3.3 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"). The floor function is monotonically increasing, as is the ceiling function.

Floors and ceilings obey the following properties. For any integer n, we have

$$\lfloor n \rfloor = n = \lceil n \rceil . \tag{3.1}$$

For all real x, we have

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1 . \tag{3.2}$$

We also have

$$-\lfloor x \rfloor = \lceil -x \rceil , \tag{3.3}$$

or equivalently,

$$-\lceil x \rceil = \lfloor -x \rfloor . \tag{3.4}$$

For any real number $x \ge 0$ and integers $a, b > 0$, we have

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

$$\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor , \tag{3.6}$$

$$\left\lceil \frac{a}{b} \right\rceil \le \frac{a + (b-1)}{b} , \tag{3.7}$$

$$\left\lfloor \frac{a}{b} \right\rfloor \ge \frac{a - (b-1)}{b} . \tag{3.8}$$

For any integer n and real number x, we have

$$\lfloor n + x \rfloor = n + \lfloor x \rfloor , \tag{3.9}$$

$$\lceil n + x \rceil = n + \lceil x \rceil . \tag{3.10}$$

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:

$$a \bmod n = a - n \lfloor a/n \rfloor . \tag{3.11}$$

It follows that

$$0 \le a \bmod n < n , \tag{3.12}$$

even when a is negative.

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 $a = b \pmod{n}$ and say that a is ***equivalent*** to b, modulo n. In other words, $a = b \pmod{n}$ if a and b have the same remainder when divided by n. Equivalently, $a = b \pmod{n}$ if and only if n is a divisor of $b - a$. We write $a \ne b \pmod{n}$ 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 assume that $0^0 = 1$.

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

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

from which we can conclude that

$$n^b = o(a^n) . \tag{3.13}$$

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,

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

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

$$1 + x \le e^x , \tag{3.14}$$

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

$$1 + x \le e^x \le 1 + x + x^2 . \tag{3.15}$$

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,

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x . \tag{3.16}$$

Logarithms

We use the following notations:

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

We adopt the following notational convention: in the absence of parentheses, *a logarithm function applies only to the next term in the formula*, so that $\lg n + 1$ means $(\lg n) + 1$ and not $\lg(n + 1)$.

For any constant $b > 1$, the function $\log_b n$ is undefined if $n \le 0$, strictly increasing if $n > 0$, negative if $0 < n < 1$, positive if $n > 1$, and 0 if $n = 1$. For all real $a > 0$, $b > 0$, $c > 0$, and n, we have

$$\begin{aligned} a &= b^{\log_b a} , && (3.17) \\ \log_c(ab) &= \log_c a + \log_c b , && (3.18) \\ \log_b a^n &= n \log_b a , \\ \log_b a &= \frac{\log_c a}{\log_c b} , && (3.19) \\ \log_b(1/a) &= -\log_b a , && (3.20) \\ \log_b a &= \frac{1}{\log_a b} , \\ a^{\log_b c} &= c^{\log_b a} , && (3.21) \end{aligned}$$

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

By equation (3.19), changing the base of a logarithm from one constant to another changes the value of the logarithm by only a constant factor. Consequently, we 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 $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots . \tag{3.22}$$

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

$$\frac{x}{1+x} \le \ln(1+x) \le x , \tag{3.23}$$

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 (3.13). For all real constants $a > 0$ and b, we have

$$\lg^b n = o(n^a) . \tag{3.24}$$

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***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) , \tag{3.25}$$

where e is the base of the natural logarithm, gives us a tighter upper bound, and a lower bound as well. Exercise 3.3-4 asks you to prove the three facts

$$n! = o(n^n) , \tag{3.26}$$

$$n! = \omega(2^n) , \tag{3.27}$$

$$\lg(n!) = \Theta(n \lg n) , \tag{3.28}$$

where Stirling's approximation is helpful in proving equation (3.28). The following equation also holds for all $n \geq 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n} \tag{3.29}$$

where

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

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

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases} \tag{3.30}$$

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 ith 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 $2^{65536} = 10^{65536/\lg 10} \approx 10^{19,728}$, we rarely encounter an input size n for which $\lg^* n > 5$.

Fibonacci numbers

We define the ***Fibonacci numbers*** F_i, for $i \geq 0$, as follows:

$$F_i = \begin{cases} 0 & \text{if } i = 0\,, \\ 1 & \text{if } i = 1\,, \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2\,. \end{cases} \tag{3.31}$$

Thus, after the first two, 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,. . ..

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

$$x^2 = x + 1\,.$$

As Exercise 3.3-7 asks you to prove, the golden ratio is given by

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots, \end{aligned} \tag{3.32}$$

and its conjugate, by

$$\begin{aligned} \hat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots\,. \end{aligned} \tag{3.33}$$

Specifically, we have

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

which can be proved by induction (Exercise 3.3-8). Since $\left|\hat{\phi}\right| < 1$, we have

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

which implies that

$$F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor\,, \tag{3.34}$$

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

Exercises

3.3-1

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.

3.3-2

Prove that $\lfloor \alpha n \rfloor + \lceil (1-\alpha)n \rceil = n$ for any integer n and real number α in the range $0 \le \alpha \le 1$.

3.3-3

Use equation (3.14) or other means to show that $(n + o(n))^k = \Theta(n^k)$ for any real constant k. Conclude that $\lceil n \rceil^k = \Theta(n^k)$ and $\lfloor n \rfloor^k = \Theta(n^k)$.

3.3-4

Prove the following:

a. Equation (3.21).

b. Equations (3.26)–(3.28).

c. $\lg(\Theta(n)) = \Theta(\lg n)$.

★ ***3.3-5***

Is the function $\lceil \lg n \rceil!$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil!$ polynomially bounded?

★ ***3.3-6***

Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

3.3-7

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

3.3-8

Prove by induction that the ith Fibonacci number satisfies the equation

$$F_i = (\phi^i - \widehat{\phi}^i)/\sqrt{5} ,$$

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

3.3-9

Show that $k \lg k = \Theta(n)$ implies $k = \Theta(n/\lg n)$.

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 $k \ge d$, then $p(n) = O(n^k)$.

b. If $k \le d$, 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, Ω, ω, or Θ of B. Assume that $k \ge 1$, $\epsilon > 0$, and $c > 1$ are constants. Write your answer in the form of the table with "yes" or "no" written in each box.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
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 $g(n)$ belong to the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	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 $g(n) = \Omega(fn))$.

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

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

3-5 Manipulating asymptotic notation

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove the following identities:

a. $\Theta(\Theta(f(n))) = \Theta(f(n))$.

b. $\Theta(f(n)) + O(f(n)) = \Theta(f(n))$.

c. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$.

d. $\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$.

e. Argue that for any real constants $a_1, b_1 > 0$ and integer constants k_1, k_2, the following asymptotic bound holds:

$$(a_1 n)^{k_1} \lg^{k_2}(a_2 n) = \Theta(n^{k_1} \lg^{k_2} n) .$$

★ ***f.*** Prove that for $S \subseteq \mathbb{Z}$, we have

$$\sum_{k \in S} \Theta(f(k)) = \Theta\left(\sum_{k \in S} f(k)\right) ,$$

assuming that both sums converge.

★ ***g.*** Show that for $S \subseteq \mathbb{Z}$, the following asymptotic bound does not necessarily hold, even assuming that both products converge, by giving a counterexample:

$$\prod_{k \in S} \Theta(f(k)) = \Theta\left(\prod_{k \in S} f(k)\right) .$$

3-6 Variations on O and Ω

Some authors define Ω-notation in a slightly different way than this textbook does. We'll use the nomenclature $\overset{\infty}{\Omega}$ (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 asymptotically nonnegative functions $f(n)$ and $g(n)$, we have $f(n) = O(g(n))$ or $f(n) = \overset{\infty}{\Omega}(g(n))$ (or both).

b. Show that there exist two asymptotically nonnegative functions $f(n)$ and $g(n)$ for which neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.

c. Describe the potential advantages and disadvantages of using $\overset{\infty}{\Omega}$-notation instead of Ω-notation to characterize the running times of programs.

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

d. What happens to each direction of the "if and only if" in Theorem 3.1 on page 56 if we substitute O' for O but still use Ω?

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

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

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

3-7 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 minimum number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the functions $f(n)$ and constants c in the table below, give as tight a bound as possible on $f_c^*(n)$. If there is no i such that $f^{(i)}(n) \le c$, write "undefined" as your answer.

	$f(n)$	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	

Chapter notes

Knuth [259] 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 Ω and Θ notations were advocated by Knuth [265] to correct the popular, but technically sloppy, practice in the literature of using O-notation for both upper and lower bounds. As noted earlier in this chapter, many people continue to use the O-notation where the Θ-notation is more technically precise. The soft-oh notation $\widetilde{O}$ in Problem 3-6 was introduced by Babai, Luks, and Seress [31], although it was originally written as $O\sim$. Some authors now define $\widetilde{O}(g(n))$ as ignoring factors that are logarithmic in $g(n)$, rather than in n. With this definition, we can say that $n2^n = \widetilde{O}(2^n)$, but with the definition in Problem 3-6, this statement is not true. Further discussion of the history and development of asymptotic notations appears in works by Knuth [259, 265] and Brassard and Bratley [70].

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 (3.29) is due to Robbins [381]. Other properties of elementary mathematical functions can be found in any good mathematical reference, such as Abramowitz and Stegun [1] or Zwillinger [468], or in a calculus book, such as Apostol [19] or Thomas et al. [433]. Knuth [259] and Graham, Knuth, and Patashnik [199] contain a wealth of material on discrete mathematics as used in computer science.

4 Divide-and-Conquer

The divide-and-conquer method is a powerful strategy for designing asymptotically efficient algorithms. We saw an example of divide-and-conquer in Section 2.3.1 when learning about merge sort. In this chapter, we'll explore applications of the divide-and-conquer method and acquire valuable mathematical tools that you can use to solve the recurrences that arise when analyzing divide-and-conquer algorithms.

Recall that for divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough—the ***base case***—you just solve it directly without recursing. Otherwise—the ***recursive case***—you perform three characteristic steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion ***bottoms out*** when it reaches a base case and the subproblem is small enough to solve directly without further recursing.

Recurrences

To analyze recursive divide-and-conquer algorithms, we'll need some mathematical tools. A ***recurrence*** is an equation that describes a function in terms of its value on other, typically smaller, arguments. Recurrences go hand in hand with the divide-and-conquer method because they give us a natural way to characterize the running times of recursive algorithms mathematically. You saw an example of a recurrence in Section 2.3.2 when we analyzed the worst-case running time of merge sort.

For the divide-and-conquer matrix-multiplication algorithms presented in Sections 4.1 and 4.2, we'll derive recurrences that describe their worst-case running times. To understand why these two divide-and-conquer algorithms perform the way they do, you'll need to learn how to solve the recurrences that describe their running times. Sections 4.3–4.7 teach several methods for solving recurrences. These sections also explore the mathematics behind recurrences, which can give you stronger intuition for designing your own divide-and-conquer algorithms.

We want to get to the algorithms as soon as possible. So, let's just cover a few recurrence basics now, and then we'll look more deeply at recurrences, especially how to solve them, after we see the matrix-multiplication examples.

The general form of a recurrence is an equation or inequality that describes a function over the integers or reals using the function itself. It contains two or more cases, depending on the argument. If a case involves the recursive invocation of the function on different (usually smaller) inputs, it is a ***recursive case***. If a case does not involve a recursive invocation, it is a ***base case***. There may be zero, one, or many functions that satisfy the statement of the recurrence. The recurrence is ***well defined*** if there is at least one function that satisfies it, and ***ill defined*** otherwise.

Algorithmic recurrences

We'll be particularly interested in recurrences that describe the running times of divide-and-conquer algorithms. A recurrence $T(n)$ is ***algorithmic*** if, for every sufficiently large ***threshold*** constant $n_0 > 0$, the following two properties hold:

1. For all $n < n_0$, we have $T(n) = \Theta(1)$.
2. For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

Similar to how we sometimes abuse asymptotic notation (see page 60), when a function is not defined for all arguments, we understand that this definition is constrained to values of n for which $T(n)$ is defined.

Why would a recurrence $T(n)$ that represents a (correct) divide-and-conquer algorithm's worst-case running time satisfy these properties for all sufficiently large threshold constants? The first property says that there exist constants c_1, c_2 such that $0 < c_1 \leq T(n) \leq c_2$ for $n < n_0$. For every legal input, the algorithm must output the solution to the problem it's solving in finite time (see Section 1.1). Thus we can let c_1 be the minimum amount of time to call and return from a procedure, which must be positive, because machine instructions need to be executed to invoke a procedure. The running time of the algorithm may not be defined for some values of n if there are no legal inputs of that size, but it must be defined for at least one, or else the "algorithm" doesn't solve any problem. Thus we can let c_2 be the algorithm's maximum running time on any input of size $n < n_0$, where n_0 is sufficiently large that the algorithm solves at least one problem of size less than n_0. The maximum is well defined, since there are at most a finite number of inputs of size less than n_0, and there is at least one if n_0 is sufficiently large. Consequently, $T(n)$ satisfies the first property. If the second property fails to hold for $T(n)$, then the algorithm isn't correct, because it would end up in an infinite recursive loop or otherwise fail to compute a solution. Thus, it stands to reason that a recurrence for the worst-case running time of a correct divide-and-conquer algorithm would be algorithmic.

Conventions for recurrences

We adopt the following convention:

> *Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic.*

That means you're free to pick any sufficiently large threshold constant n_0 for the range of base cases where $T(n) = \Theta(1)$. Interestingly, the asymptotic solutions of most algorithmic recurrences you're likely to see when analyzing algorithms don't depend on the choice of threshold constant, as long as it's large enough to make the recurrence well defined.

Asymptotic solutions of algorithmic divide-and-conquer recurrences also don't tend to change when we drop any floors or ceilings in a recurrence defined on the integers to convert it to a recurrence defined on the reals. Section 4.7 gives a sufficient condition for ignoring floors and ceilings that applies to most of the divide-and-conquer recurrences you're likely to see. Consequently, we'll frequently state algorithmic recurrences without floors and ceilings. Doing so generally simplifies the statement of the recurrences, as well as any math that we do with them.

You may sometimes see recurrences that are not equations, but rather inequalities, such as $T(n) \le 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we express its solution using O-notation rather than Θ-notation. Similarly, if the inequality is reversed to $T(n) \ge 2T(n/2) + \Theta(n)$, then, because the recurrence gives only a lower bound on $T(n)$, we use Ω-notation in its solution.

Divide-and-conquer and recurrences

This chapter illustrates the divide-and-conquer method by presenting and using recurrences to analyze two divide-and-conquer algorithms for multiplying $n \times n$ matrices. Section 4.1 presents a simple divide-and-conquer algorithm that solves a matrix-multiplication problem of size n by breaking it into four subproblems of size $n/2$, which it then solves recursively. The running time of the algorithm can be characterized by the recurrence

$$T(n) = 8T(n/2) + \Theta(1) ,$$

which turns out to have the solution $T(n) = \Theta(n^3)$. Although this divide-and-conquer algorithm is no faster than the straightforward method that uses a triply nested loop, it leads to an asymptotically faster divide-and-conquer algorithm due to V. Strassen, which we'll explore in Section 4.2. Strassen's remarkable algorithm divides a problem of size n into seven subproblems of size $n/2$ which it solves recursively. The running time of Strassen's algorithm can be described by the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2) ,$$

which has the solution $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$. Strassen's algorithm beats the straightforward looping method asymptotically.

These two divide-and-conquer algorithms both break a problem of size n into several subproblems of size $n/2$. Although it is common when using divide-and-conquer for all the subproblems to have the same size, that isn't always the case. Sometimes it's productive to divide a problem of size n into subproblems of different sizes, and then the recurrence describing the running time reflects the irregularity. For example, consider a divide-and-conquer algorithm that divides a problem of size n into one subproblem of size $n/3$ and another of size $2n/3$, taking $\Theta(n)$ time to divide the problem and combine the solutions to the subproblems. Then the algorithm's running time can be described by the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) ,$$

which turns out to have solution $T(n) = \Theta(n \lg n)$. We'll even see an algorithm in Chapter 9 that solves a problem of size n by recursively solving a subproblem of size $n/5$ and another of size $7n/10$, taking $\Theta(n)$ time for the divide and combine steps. Its performance satisfies the recurrence

$$T(n) = T(n/5) + T(7n/10) + \Theta(n) ,$$

which has solution $T(n) = \Theta(n)$.

Although divide-and-conquer algorithms usually create subproblems with sizes a constant fraction of the original problem size, that's not always the case. For example, a recursive version of linear search (see Exercise 2.1-4) creates just one subproblem, with one element less than the original problem. Each recursive call takes constant time plus the time to recursively solve a subproblem with one less element, leading to the recurrence

$$T(n) = T(n-1) + \Theta(1) ,$$

which has solution $T(n) = \Theta(n)$. Nevertheless, the vast majority of efficient divide-and-conquer algorithms solve subproblems that are a constant fraction of the size of the original problem, which is where we'll focus our efforts.

Solving recurrences

After learning about divide-and-conquer algorithms for matrix multiplication in Sections 4.1 and 4.2, we'll explore several mathematical tools for solving recurrences—that is, for obtaining asymptotic Θ-, O-, or Ω-bounds on their solutions. We want simple-to-use tools that can handle the most commonly occurring situations. But we also want general tools that work, perhaps with a little more effort, for less common cases. This chapter offers four methods for solving recurrences:

- In the ***substitution method*** (Section 4.3), you guess the form of a bound and then use mathematical induction to prove your guess correct and solve for constants. This method is perhaps the most robust method for solving recurrences, but it also requires you to make a good guess and to produce an inductive proof.
- The ***recursion-tree method*** (Section 4.4) models the recurrence as a tree whose nodes represent the costs incurred at various levels of the recursion. To solve the recurrence, you determine the costs at each level and add them up, perhaps using techniques for bounding summations from Section A.2. Even if you don't use this method to formally prove a bound, it can be helpful in guessing the form of the bound for use in the substitution method.
- The ***master method*** (Sections 4.5 and 4.6) is the easiest method, when it applies. It provides bounds for recurrences of the form

  $$T(n) = aT(n/b) + f(n) ,$$

  where $a > 0$ and $b > 1$ are constants and $f(n)$ is a given "driving" function. This type of recurrence tends to arise more frequently in the study of algorithms than any other. It characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ times the size of the original problem, using $f(n)$ time for the divide and combine steps. To apply the master method, you need to memorize three cases, but once you do, you can easily determine asymptotic bounds on running times for many divide-and-conquer algorithms.
- The ***Akra-Bazzi method*** (Section 4.7) is a general method for solving divide-and-conquer recurrences. Although it involves calculus, it can be used to attack more complicated recurrences than those addressed by the master method.

4.1 Multiplying square matrices

We can use the divide-and-conquer method to multiply square matrices. If you've seen matrices before, then you probably know how to multiply them. (Otherwise, you should read Section D.1.) Let $A = (a_{ik})$ and $B = (b_{jk})$ be square $n \times n$ matrices. The matrix product $C = A \cdot B$ is also an $n \times n$ matrix, where for $i, j = 1, 2, \ldots, n$, the (i, j) entry of C is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{4.1}$$

Generally, we'll assume that the matrices are ***dense***, meaning that most of the n^2 entries are not 0, as opposed to ***sparse***, where most of the n^2 entries are 0 and the nonzero entries can be stored more compactly than in an $n \times n$ array.

Computing the matrix C requires computing n^2 matrix entries, each of which is the sum of n pairwise products of input elements from A and B. The MATRIX-MULTIPLY procedure implements this strategy in a straightforward manner, and it generalizes the problem slightly. It takes as input three $n \times n$ matrices A, B, and C, and it adds the matrix product $A \cdot B$ to C, storing the result in C. Thus, it computes $C = C + A \cdot B$, instead of just $C = A \cdot B$. If only the product $A \cdot B$ is needed, just initialize all n^2 entries of C to 0 before calling the procedure, which takes an additional $\Theta(n^2)$ time. We'll see that the cost of matrix multiplication asymptotically dominates this initialization cost.

```
MATRIX-MULTIPLY($A, B, C, n$)
1  for $i = 1$ to $n$                          // compute entries in each of $n$ rows
2      for $j = 1$ to $n$                      // compute $n$ entries in row $i$
3          for $k = 1$ to $n$
4              $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$    // add in another term of equation (4.1)
```

The pseudocode for MATRIX-MULTIPLY works as follows. The **for** loop of lines 1–4 computes the entries of each row i, and within a given row i, the **for** loop of lines 2–4 computes each of the entries c_{ij} for each column j. Each iteration of the **for** loop of lines 3–4 adds in one more term of equation (4.1).

Because each of the triply nested **for** loops runs for exactly n iterations, and each execution of line 4 takes constant time, the MATRIX-MULTIPLY procedure operates in $\Theta(n^3)$ time. Even if we add in the $\Theta(n^2)$ time for initializing C to 0, the running time is still $\Theta(n^3)$.

A simple divide-and-conquer algorithm

Let's see how to compute the matrix product $A \cdot B$ using divide-and-conquer. For $n > 1$, the divide step partitions the $n \times n$ matrices into four $n/2 \times n/2$ submatrices. We'll assume that n is an exact power of 2, so that as the algorithm recurses, we are guaranteed that the submatrix dimensions are integer. (Exercise 4.1-1 asks you to relax this assumption.) As with MATRIX-MULTIPLY, we'll actually compute $C = C + A \cdot B$. But to simplify the math behind the algorithm, let's assume that C has been initialized to the zero matrix, so that we are indeed computing $C = A \cdot B$.

The divide step views each of the $n \times n$ matrices A, B, and C as four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. \tag{4.2}$$

Then we can write the matrix product as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{4.3}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}, \tag{4.4}$$

which corresponds to the equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.5}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.6}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.7}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.8}$$

Equations (4.5)–(4.8) involve eight $n/2 \times n/2$ multiplications and four additions of $n/2 \times n/2$ submatrices.

As we look to transform these equations to an algorithm that can be described with pseudocode, or even implemented for real, there are two common approaches for implementing the matrix partitioning.

One strategy is to allocate temporary storage to hold A's four submatrices A_{11}, A_{12}, A_{21}, and A_{22} and B's four submatrices B_{11}, B_{12}, B_{21}, and B_{22}. Then copy each element in A and B to its corresponding location in the appropriate submatrix. After the recursive conquer step, copy the elements in each of C's four submatrices C_{11}, C_{12}, C_{21}, and C_{22} to their corresponding locations in C. This approach takes $\Theta(n^2)$ time, since $3n^2$ elements are copied.

The second approach uses index calculations and is faster and more practical. A submatrix can be specified within a matrix by indicating where within the matrix the submatrix lies without touching any matrix elements. Partitioning a matrix (or recursively, a submatrix) only involves arithmetic on this location information, which has constant size independent of the size of the matrix. Changes to the submatrix elements update the original matrix, since they occupy the same storage.

Going forward, we'll assume that index calculations are used and that partitioning can be performed in $\Theta(1)$ time. Exercise 4.1-3 asks you to show that it makes no difference to the overall asymptotic running time of matrix multiplication, however, whether the partitioning of matrices uses the first method of copying or the second method of index calculation. But for other divide-and-conquer matrix calculations, such as matrix addition, it can make a difference, as Exercise 4.1-4 asks you to show.

The procedure MATRIX-MULTIPLY-RECURSIVE uses equations (4.5)–(4.8) to implement a divide-and-conquer strategy for square-matrix multiplication. Like MATRIX-MULTIPLY, the procedure MATRIX-MULTIPLY-RECURSIVE computes $C = C + A \cdot B$ since, if necessary, C can be initialized to 0 before the procedure is called in order to compute only $C = A \cdot B$.

```
MATRIX-MULTIPLY-RECURSIVE($A, B, C, n$)
 1  if $n == 1$
 2  // Base case.
 3      $c_{11} = c_{11} + a_{11} \cdot b_{11}$
 4      return
 5  // Divide.
 6  partition $A$, $B$, and $C$ into $n/2 \times n/2$ submatrices
        $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$;
        and $C_{11}, C_{12}, C_{21}, C_{22}$; respectively
 7  // Conquer.
 8  MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{11}, C_{11}, n/2$)
 9  MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{12}, C_{12}, n/2$)
10  MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{11}, C_{21}, n/2$)
11  MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{12}, C_{22}, n/2$)
12  MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{21}, C_{11}, n/2$)
13  MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{22}, C_{12}, n/2$)
14  MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{21}, C_{21}, n/2$)
15  MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{22}, C_{22}, n/2$)
```

As we walk through the pseudocode, we'll derive a recurrence to characterize its running time. Let $T(n)$ be the worst-case time to multiply two $n \times n$ matrices using this procedure.

In the base case, when $n = 1$, line 3 performs just the one scalar multiplication and one addition, which means that $T(1) = \Theta(1)$. As is our convention for constant base cases, we can omit this base case in the statement of the recurrence.

The recursive case occurs when $n > 1$. As discussed, we'll use index calculations to partition the matrices in line 6, taking $\Theta(1)$ time. Lines 8–15 recursively call MATRIX-MULTIPLY-RECURSIVE a total of eight times. The first four recursive calls compute the first terms of equations (4.5)–(4.8), and the subsequent four recursive calls compute and add in the second terms. Each recursive call adds the product of a submatrix of A and a submatrix of B to the appropriate submatrix of C in place, thanks to index calculations. Because each recursive call multiplies two $n/2 \times n/2$ matrices, thereby contributing $T(n/2)$ to the overall running time, the time taken by all eight recursive calls is $8T(n/2)$. There is no combine step, because the matrix C is updated in place. The total time for the recursive case, therefore, is the sum of the partitioning time and the time for all the recursive calls, or $\Theta(1) + 8T(n/2)$.

Thus, omitting the statement of the base case, our recurrence for the running time of MATRIX-MULTIPLY-RECURSIVE is

$$T(n) = 8T(n/2) + \Theta(1) . \tag{4.9}$$

As we'll see from the master method in Section 4.5, recurrence (4.9) has the solution $T(n) = \Theta(n^3)$, which means that it has the same asymptotic running time as the straightforward MATRIX-MULTIPLY procedure.

Why is the $\Theta(n^3)$ solution to this recurrence so much larger than the $\Theta(n \lg n)$ solution to the merge-sort recurrence (2.3) on page 41? After all, the recurrence for merge sort contains a $\Theta(n)$ term, whereas the recurrence for recursive matrix multiplication contains only a $\Theta(1)$ term.

Let's think about what the recursion tree for recurrence (4.9) would look like as compared with the recursion tree for merge sort, illustrated in Figure 2.5 on page 43. The factor of 2 in the merge-sort recurrence determines how many children each tree node has, which in turn determines how many terms contribute to the sum at each level of the tree. In comparison, for the recurrence (4.9) for MATRIX-MULTIPLY-RECURSIVE, each internal node in the recursion tree has eight children, not two, leading to a "bushier" recursion tree with many more leaves, despite the fact that the internal nodes are each much smaller. Consequently, the solution to recurrence (4.9) grows much more quickly than the solution to recurrence (2.3), which is borne out in the actual solutions: $\Theta(n^3)$ versus $\Theta(n \lg n)$.

Exercises

Note: You may wish to read Section 4.5 before attempting some of these exercises.

4.1-1
Generalize MATRIX-MULTIPLY-RECURSIVE to multiply $n \times n$ matrices for which n is not necessarily an exact power of 2. Give a recurrence describing its running time. Argue that it runs in $\Theta(n^3)$ time in the worst case.

4.1-2
How quickly can you multiply a $kn \times n$ matrix (kn rows and n columns) by an $n \times kn$ matrix, where $k \geq 1$, using MATRIX-MULTIPLY-RECURSIVE as a subroutine? Answer the same question for multiplying an $n \times kn$ matrix by a $kn \times n$ matrix. Which is asymptotically faster, and by how much?

4.1-3

Suppose that instead of partitioning matrices by index calculation in MATRIX-MULTIPLY-RECURSIVE, you copy the appropriate elements of A, B, and C into separate $n/2 \times n/2$ submatrices A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; and C_{11}, C_{12}, C_{21}, C_{22}, respectively. After the recursive calls, you copy the results from C_{11}, C_{12}, C_{21}, and C_{22} back into the appropriate places in C. How does recurrence (4.9) change, and what is its solution?

4.1-4

Write pseudocode for a divide-and-conquer algorithm MATRIX-ADD-RECURSIVE that sums two $n \times n$ matrices A and B by partitioning each of them into four $n/2 \times n/2$ submatrices and then recursively summing corresponding pairs of submatrices. Assume that matrix partitioning uses $\Theta(1)$-time index calculations. Write a recurrence for the worst-case running time of MATRIX-ADD-RECURSIVE, and solve your recurrence. What happens if you use $\Theta(n^2)$-time copying to implement the partitioning instead of index calculations?

4.2 Strassen's algorithm for matrix multiplication

You might find it hard to imagine that any matrix multiplication algorithm could take less than $\Theta(n^3)$ time, since the natural definition of matrix multiplication requires n^3 scalar multiplications. Indeed, many mathematicians presumed that it was not possible to multiply matrices in $o(n^3)$ time until 1969, when V. Strassen [424] published a remarkable recursive algorithm for multiplying $n \times n$ matrices. Strassen's algorithm runs in $\Theta(n^{\lg 7})$ time. Since $\lg 7 = 2.8073549\ldots$, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the $\Theta(n^3)$ MATRIX-MULTIPLY and MATRIX-MULTIPLY-RECURSIVE procedures.

The key to Strassen's method is to use the divide-and-conquer idea from the MATRIX-MULTIPLY-RECURSIVE procedure, but make the recursion tree less bushy. We'll actually increase the work for each divide and combine step by a constant factor, but the reduction in bushiness will pay off. We won't reduce the bushiness from the eight-way branching of recurrence (4.9) all the way down to the two-way branching of recurrence (2.3), but we'll improve it just a little, and that will make a big difference. Instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, Strassen's algorithm performs only seven. The cost of eliminating one matrix multiplication is several new additions and subtractions of $n/2 \times n/2$ matrices, but still only a constant number. Rather than saying "additions and subtractions" everywhere, we'll adopt the common terminology of calling them both "additions" because subtraction is structurally the same computation as addition, except for a change of sign.

To get an inkling how the number of multiplications might be reduced, as well as why reducing the number of multiplications might be desirable for matrix calculations, suppose that you have two numbers x and y, and you want to calculate the quantity $x^2 - y^2$. The straightforward calculation requires two multiplications to square x and y, followed by one subtraction (which you can think of as a "negative addition"). But let's recall the old algebra trick $x^2 - y^2 = x^2 - xy + xy - y^2 = x(x - y) + y(x - y) = (x + y)(x - y)$. Using this formulation of the desired quantity, you could instead compute the sum $x + y$ and the difference $x - y$ and then multiply them, requiring only a single multiplication and two additions. At the cost of an extra addition, only one multiplication is needed to compute an expression that looks as if it requires two. If x and y are scalars, there's not much difference: both approaches require three scalar operations. If x and y are large matrices, however, the cost of multiplying outweighs the cost of adding, in which case the second method outperforms the first, although not asymptotically.

Strassen's strategy for reducing the number of matrix multiplications at the expense of more matrix additions is not at all obvious—perhaps the biggest understatement in this book! As with MATRIX-MULTIPLY-RECURSIVE, Strassen's algorithm uses the divide-and-conquer method to compute $C = C + A \cdot B$, where A, B, and C are all $n \times n$ matrices and n is an exact power of 2. Strassen's algorithm computes the four submatrices C_{11}, C_{12}, C_{21}, and C_{22} of C from equations (4.5)–(4.8) on page 82 in four steps. We'll analyze costs as we go along to develop a recurrence $T(n)$ for the overall running time. Let's see how it works:

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, as in line 3 of MATRIX-MULTIPLY-RECURSIVE, taking $\Theta(1)$ time, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2). This step takes $\Theta(1)$ time by index calculation, just as in MATRIX-MULTIPLY-RECURSIVE.

2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All 17 matrices can be created, and the P_i initialized, in $\Theta(n^2)$ time.

3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively compute each of the seven matrix products $P_1, P_2, \ldots, P_7$, taking $7T(n/2)$ time.

4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices, which takes $\Theta(n^2)$ time.

We'll see the details of steps 2–4 in a moment, but we already have enough information to set up a recurrence for the running time of Strassen's method. As is common, the base case in step 1 takes $\Theta(1)$ time, which we'll omit when stating the recurrence. When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires seven multiplications of $n/2 \times n/2$ matrices. Hence, we obtain the following recurrence for the running time of Strassen's algorithm:

$$T(n) = 7T(n/2) + \Theta(n^2) . \tag{4.10}$$

Compared with MATRIX-MULTIPLY-RECURSIVE, we have traded off one recursive submatrix multiplication for a constant number of submatrix additions. Once you understand recurrences and their solutions, you'll be able to see why this trade-off actually leads to a lower asymptotic running time. By the master method in Section 4.5, recurrence (4.10) has the solution $T(n) = \Theta(n^{\lg 7}) = O(n^{2.81})$, beating the $\Theta(n^3)$-time algorithms.

Now, let's delve into the details. Step 2 creates the following 10 matrices:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}$$

This step adds or subtracts $n/2 \times n/2$ matrices 10 times, taking $\Theta(n^2)$ time.

Step 3 recursively multiplies $n/2 \times n/2$ matrices 7 times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$\begin{aligned}
P_1 &= A_{11} \cdot S_1 && (= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}) , \\
P_2 &= S_2 \cdot B_{22} && (= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}) , \\
P_3 &= S_3 \cdot B_{11} && (= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}) , \\
P_4 &= A_{22} \cdot S_4 && (= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}) , \\
P_5 &= S_5 \cdot S_6 && (= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}) , \\
P_6 &= S_7 \cdot S_8 && (= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}) , \\
P_7 &= S_9 \cdot S_{10} && (= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}) .
\end{aligned}$$

The only multiplications that the algorithm performs are those in the middle column of these equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1, but the terms are never explicitly calculated by the algorithm.

Step 4 adds to and subtracts from the four $n/2 \times n/2$ submatrices of the product C the various P_i matrices created in step 3. We start with

$$C_{11} = C_{11} + P_5 + P_4 - P_2 + P_6 .$$

Expanding the calculation on the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that the update to C_{11} equals

$$\begin{array}{lllllll}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & & - A_{22} \cdot B_{11} & & + A_{22} \cdot B_{21} & & \\
 & - A_{11} \cdot B_{22} & & & & - A_{12} \cdot B_{22} & \\
 & & & - A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & & & & + A_{12} \cdot B_{21} ,
\end{array}$$

which corresponds to equation (4.5). Similarly, setting

$$C_{12} = C_{12} + P_1 + P_2$$

means that the update to C_{12} equals

$$\begin{array}{lll}
A_{11} \cdot B_{12} & - A_{11} \cdot B_{22} & \\
 & + A_{11} \cdot B_{22} & + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} & & + A_{12} \cdot B_{22} ,
\end{array}$$

corresponding to equation (4.6). Setting

$$C_{21} = C_{21} + P_3 + P_4$$

means that the update to C_{21} equals

$$\begin{array}{lll}
A_{21} \cdot B_{11} & + A_{22} \cdot B_{11} & \\
 & - A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} \\
\hline
A_{21} \cdot B_{11} & & + A_{22} \cdot B_{21} ,
\end{array}$$

corresponding to equation (4.7). Finally, setting

$$C_{22} = C_{22} + P_5 + P_1 - P_3 - P_7$$

means that the update to C_{22} equals

$$
\begin{array}{rrrrrrr}
A_{11} \cdot B_{11} & + A_{11} \cdot B_{22} & + A_{22} \cdot B_{11} & + A_{22} \cdot B_{22} & & & \\
 & - A_{11} \cdot B_{22} & & & + A_{11} \cdot B_{12} & & \\
 & & - A_{22} \cdot B_{11} & & & - A_{21} \cdot B_{11} & \\
- A_{11} \cdot B_{11} & & & & - A_{11} \cdot B_{12} & + A_{21} \cdot B_{11} & + A_{21} \cdot B_{12} \\
\hline
 & & & A_{22} \cdot B_{22} & & & + A_{21} \cdot B_{12} ,
\end{array}
$$

which corresponds to equation (4.8). Altogether, since we add or subtract $n/2 \times n/2$ matrices 12 times in step 4, this step indeed takes $\Theta(n^2)$ time.

We can see that Strassen's remarkable algorithm, comprising steps 1–4, produces the correct matrix product using 7 submatrix multiplications and 18 submatrix additions. We can also see that recurrence (4.10) characterizes its running time. Since Section 4.5 shows that this recurrence has the solution $T(n) = \Theta(n^{\lg 7}) = o(n^3)$, Strassen's method asymptotically beats the $\Theta(n^3)$ MATRIX-MULTIPLY and MATRIX-MULTIPLY-RECURSIVE procedures.

Exercises

Note: You may wish to read Section 4.5 before attempting some of these exercises.

4.2-1

Use Strassen's algorithm to compute the matrix product

$$
\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix} .
$$

Show your work.

4.2-2

Write pseudocode for Strassen's algorithm.

4.2-3

What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in $o(n^{\lg 7})$ time? What is the running time of this algorithm?

4.2-4

V. Pan discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare with Strassen's algorithm?

4.2-5

Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

4.2-6

Suppose that you have a $\Theta(n^{\alpha})$-time algorithm for squaring $n \times n$ matrices, where $\alpha \geq 2$. Show how to use that algorithm to multiply two different $n \times n$ matrices in $\Theta(n^{\alpha})$ time.

4.3 The substitution method for solving recurrences

Now that you have seen how recurrences characterize the running times of divide-and-conquer algorithms, let's learn how to solve them. We start in this section with the ***substitution method***, which is the most general of the four methods in this chapter. The substitution method comprises two steps:

1. Guess the form of the solution using symbolic constants.
2. Use mathematical induction to show that the solution works, and find the constants.

To apply the inductive hypothesis, you substitute the guessed solution for the function on smaller values—hence the name "substitution method." This method is powerful, but you must guess the form of the answer. Although generating a good guess might seem difficult, a little practice can quickly improve your intuition.

You can use the substitution method to establish either an upper or a lower bound on a recurrence. It's usually best not to try to do both at the same time. That is, rather than trying to prove a Θ-bound directly, first prove an O-bound, and then prove an Ω-bound. Together, they give you a Θ-bound (Theorem 3.1 on page 56).

As an example of the substitution method, let's determine an asymptotic upper bound on the recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) . \tag{4.11}$$

This recurrence is similar to recurrence (2.3) on page 41 for merge sort, except for the floor function, which ensures that $T(n)$ is defined over the integers. Let's guess that the asymptotic upper bound is the same—$T(n) = O(n \lg n)$—and use the substitution method to prove it.

We'll adopt the inductive hypothesis that $T(n) \leq cn \lg n$ for all $n \geq n_0$, where we'll choose the specific constants $c > 0$ and $n_0 > 0$ later, after we see what constraints they need to obey. If we can establish this inductive hypothesis, we can conclude that $T(n) = O(n \lg n)$. It would be dangerous to use $T(n) = O(n \lg n)$ as the inductive hypothesis because the constants matter, as we'll see in a moment in our discussion of pitfalls.

Assume by induction that this bound holds for all numbers at least as big as n_0 and less than n. In particular, therefore, if $n \geq 2n_0$, it holds for $\lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into recurrence (4.11)—hence the name "substitution" method—yields

$$\begin{aligned}
T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + \Theta(n) \\
&\leq 2(c(n/2) \lg(n/2)) + \Theta(n) \\
&= cn \lg(n/2) + \Theta(n) \\
&= cn \lg n - cn \lg 2 + \Theta(n) \\
&= cn \lg n - cn + \Theta(n) \\
&\leq cn \lg n \,,
\end{aligned}$$

where the last step holds if we constrain the constants n_0 and c to be sufficiently large that for $n \geq 2n_0$, the quantity cn dominates the anonymous function hidden by the $\Theta(n)$ term.

We've shown that the inductive hypothesis holds for the inductive case, but we also need to prove that the inductive hypothesis holds for the base cases of the induction, that is, that $T(n) \leq cn \lg n$ when $n_0 \leq n < 2n_0$. As long as $n_0 > 1$ (a new constraint on n_0), we have $\lg n > 0$, which implies that $n \lg n > 0$. So let's pick $n_0 = 2$. Since the base case of recurrence (4.11) is not stated explicitly, by our convention, $T(n)$ is algorithmic, which means that $T(2)$ and $T(3)$ are constant (as they should be if they describe the worst-case running time of any real program on inputs of size 2 or 3). Picking $c = \max\{T(2), T(3)\}$ yields $T(2) \leq c < (2 \lg 2)c$ and $T(3) \leq c < (3 \lg 3)c$, establishing the inductive hypothesis for the base cases.

Thus, we have $T(n) \leq cn \lg n$ for all $n \geq 2$, which implies that the solution to recurrence (4.11) is $T(n) = O(n \lg n)$.

In the algorithms literature, people rarely carry out their substitution proofs to this level of detail, especially in their treatment of base cases. The reason is that for most algorithmic divide-and-conquer recurrences, the base cases are all handled in pretty much the same way. You ground the induction on a range of values from a convenient positive constant n_0 up to some constant $n'_0 > n_0$ such that for $n \geq n'_0$, the recurrence always bottoms out in a constant-sized base case between n_0 and n'_0. (This example used $n'_0 = 2n_0$.) Then, it's usually apparent, without spelling out the details, that with a suitably large choice of the leading constant (such as c for this example), the inductive hypothesis can be made to hold for all the values in the range from n_0 to n'_0.

Making a good guess

Unfortunately, there is no general way to correctly guess the tightest asymptotic solution to an arbitrary recurrence. Making a good guess takes experience and, occasionally, creativity. Fortunately, learning some recurrence-solving heuristics, as well as playing around with recurrences to gain experience, can help you become a good guesser. You can also use recursion trees, which we'll see in Section 4.4, to help generate good guesses.

If a recurrence is similar to one you've seen before, then guessing a similar solution is reasonable. As an example, consider the recurrence

$$T(n) = 2T(n/2 + 17) + \Theta(n) ,$$

defined on the reals. This recurrence looks somewhat like the merge-sort recurrence (2.3), but it's more complicated because of the added "17" in the argument to T on the right-hand side. Intuitively, however, this additional term shouldn't substantially affect the solution to the recurrence. When n is large, the relative difference between $n/2$ and $n/2 + 17$ is not that large: both cut n nearly in half. Consequently, it makes sense to guess that $T(n) = O(n \lg n)$, which you can verify is correct using the substitution method (see Exercise 4.3-1).

Another way to make a good guess is to determine loose upper and lower bounds on the recurrence and then reduce your range of uncertainty. For example, you might start with a lower bound of $T(n) = \Omega(n)$ for recurrence (4.11), since the recurrence includes the term $\Theta(n)$, and you can prove an initial upper bound of $T(n) = O(n^2)$. Then split your time between trying to lower the upper bound and trying to raise the lower bound until you converge on the correct, asymptotically tight solution, which in this case is $T(n) = \Theta(n \lg n)$.

A trick of the trade: subtracting a low-order term

Sometimes, you might correctly guess a tight asymptotic bound on the solution of a recurrence, but somehow the math fails to work out in the induction proof. The problem frequently turns out to be that the inductive assumption is not strong enough. The trick to resolving this problem is to revise your guess by *subtracting* a lower-order term when you hit such a snag. The math then often goes through.

Consider the recurrence

$$T(n) = 2T(n/2) + \Theta(1) \tag{4.12}$$

defined on the reals. Let's guess that the solution is $T(n) = O(n)$ and try to show that $T(n) \le cn$ for $n \ge n_0$, where we choose the constants $c, n_0 > 0$ suitably. Substituting our guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\le 2(c(n/2)) + \Theta(1) \\ &= cn + \Theta(1) , \end{aligned}$$

which, unfortunately, does not imply that $T(n) \le cn$ for *any* choice of c. We might be tempted to try a larger guess, say $T(n) = O(n^2)$. Although this larger guess works, it provides only a loose upper bound. It turns out that our original guess of $T(n) = O(n)$ is correct and tight. In order to show that it is correct, however, we must strengthen our inductive hypothesis.

Intuitively, our guess is nearly right: we are off only by $\Theta(1)$, a lower-order term. Nevertheless, mathematical induction requires us to prove the *exact* form of the inductive hypothesis. Let's try our trick of subtracting a lower-order term from our previous guess: $T(n) \le cn - d$, where $d \ge 0$ is a constant. We now have

$$\begin{aligned} T(n) &\le 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\le cn - d - (d - \Theta(1)) \\ &\le cn - d \end{aligned}$$

as long as we choose d to be larger than the anonymous upper-bound constant hidden by the Θ-notation. Subtracting a lower-order term works! Of course, we must not forget to handle the base case, which is to choose the constant c large enough that $cn - d$ dominates the implicit base cases.

You might find the idea of subtracting a lower-order term to be counterintuitive. After all, if the math doesn't work out, shouldn't you increase your guess? Not necessarily! When the recurrence contains more than one recursive invocation (recurrence (4.12) contains two), if you add a lower-order term to the guess, then you end up adding it once for each of the recursive invocations. Doing so takes you even further away from the inductive hypothesis. On the other hand, if you subtract a lower-order term from the guess, then you get to subtract it once for each of the recursive invocations. In the above example, we subtracted the constant d twice because the coefficient of $T(n/2)$ is 2. We ended up with the inequality $T(n) \le cn - d - (d - \Theta(1))$, and we readily found a suitable value for d.

Avoiding pitfalls

Avoid using asymptotic notation in the inductive hypothesis for the substitution method because it's error prone. For example, for recurrence (4.11), we can falsely "prove" that $T(n) = O(n)$ if we unwisely adopt $T(n) = O(n)$ as our inductive hypothesis:

$$\begin{aligned} T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ &= 2 \cdot O(n) + \Theta(n) \\ &= O(n) \,. \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

The problem with this reasoning is that the constant hidden by the O-notation changes. We can expose the fallacy by repeating the "proof" using an explicit constant. For the inductive hypothesis, assume that $T(n) \le cn$ for all $n \ge n_0$, where $c, n_0 > 0$ are constants. Repeating the first two steps in the inequality chain yields

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) . \end{aligned}$$

Now, indeed $cn + \Theta(n) = O(n)$, but the constant hidden by the O-notation must be larger than c because the anonymous function hidden by the $\Theta(n)$ is asymptotically positive. We cannot take the third step to conclude that $cn + \Theta(n) \le cn$, thus exposing the fallacy.

When using the substitution method, or more generally mathematical induction, you must be careful that the constants hidden by any asymptotic notation are the same constants throughout the proof. Consequently, it's best to avoid asymptotic notation in your inductive hypothesis and to name constants explicitly.

Here's another fallacious use of the substitution method to show that the solution to recurrence (4.11) is $T(n) = O(n)$. We guess $T(n) \le cn$ and then argue

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ &\le cn + \Theta(n) \\ &= O(n) , \qquad \Longleftarrow \textit{wrong!} \end{aligned}$$

since c is a positive constant. The mistake stems from the difference between our goal—to prove that $T(n) = O(n)$—and our inductive hypothesis—to prove that $T(n) \le cn$. When using the substitution method, or in any inductive proof, you must prove the *exact* statement of the inductive hypothesis. In this case, we must explicitly prove that $T(n) \le cn$ to show that $T(n) = O(n)$.

Exercises

4.3-1

Use the substitution method to show that each of the following recurrences defined on the reals has the asymptotic solution specified:

a. $T(n) = T(n-1) + n$ has solution $T(n) = O(n^2)$.

b. $T(n) = T(n/2) + \Theta(1)$ has solution $T(n) = O(\lg n)$.

c. $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n \lg n)$.

d. $T(n) = 2T(n/2 + 17) + n$ has solution $T(n) = O(n \lg n)$.

e. $T(n) = 2T(n/3) + \Theta(n)$ has solution $T(n) = \Theta(n)$.

f. $T(n) = 4T(n/2) + \Theta(n)$ has solution $T(n) = \Theta(n^2)$.

4.3-2

The solution to the recurrence $T(n) = 4T(n/2) + n$ turns out to be $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \le cn^2$ fails. Then show how to subtract a lower-order term to make a substitution proof work.

4.3-3

The recurrence $T(n) = 2T(n-1) + 1$ has the solution $T(n) = O(2^n)$. Show that a substitution proof fails with the assumption $T(n) \le c2^n$, where $c > 0$ is constant. Then show how to subtract a lower-order term to make a substitution proof work.

4.4 The recursion-tree method for solving recurrences

Although you can use the substitution method to prove that a solution to a recurrence is correct, you might have trouble coming up with a good guess. Drawing out a recursion tree, as we did in our analysis of the merge-sort recurrence in Section 2.3.2, can help. In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. You typically sum the costs within each level of the tree to obtain the per-level costs, and then you sum all the per-level costs to determine the total cost of all levels of the recursion. Sometimes, however, adding up the total cost takes more creativity.

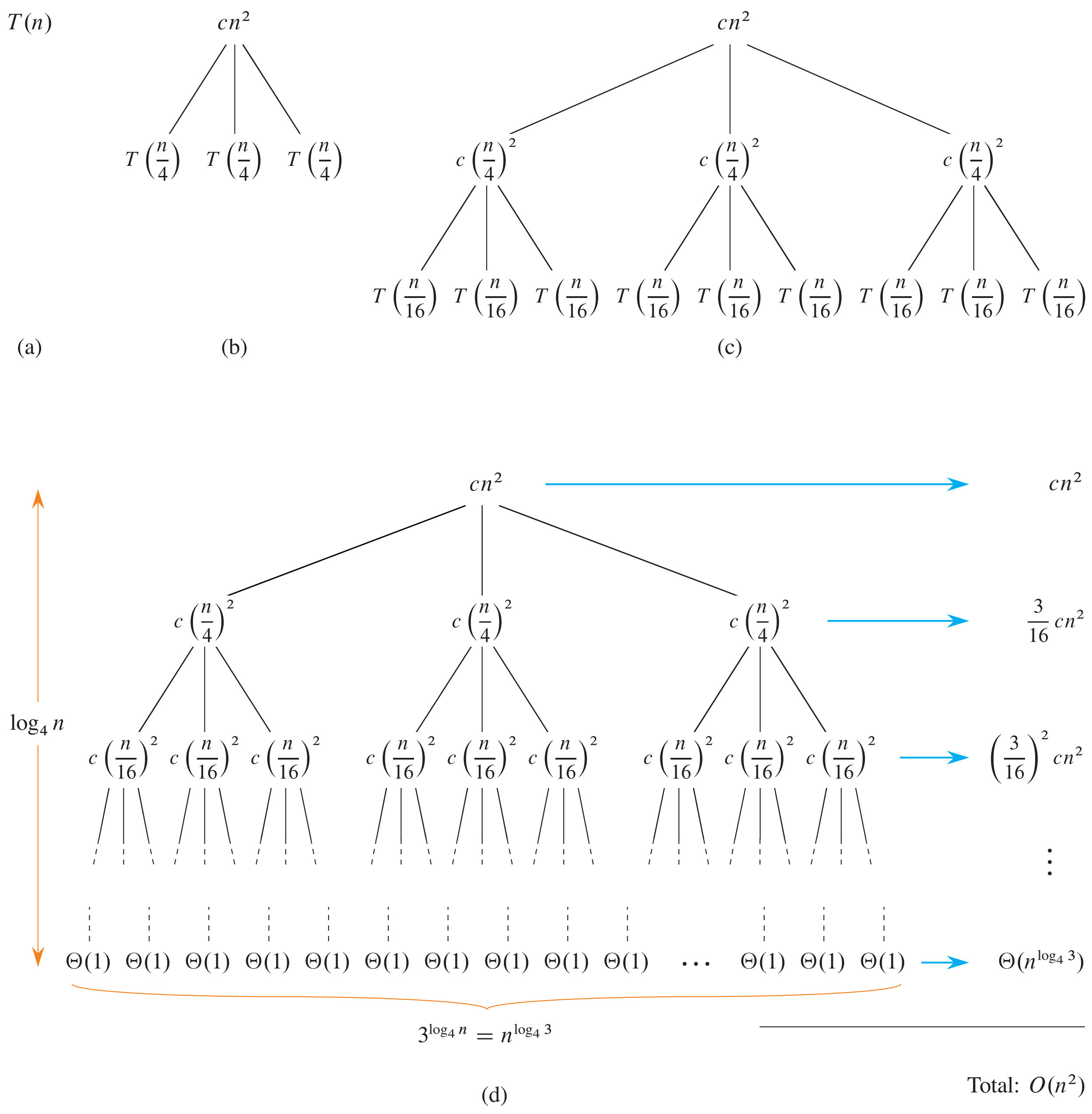
A recursion tree is best used to generate intuition for a good guess, which you can then verify by the substitution method. If you are meticulous when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence. But if you use it only to generate a good guess, you can often tolerate a small amount of "sloppiness," which can simplify the math. When you verify your guess with the substitution method later on, your math should be precise. This section demonstrates how you can use recursion trees to solve recurrences, generate good guesses, and gain intuition for recurrences.

An illustrative example

Let's see how a recursion tree can provide a good guess for an upper-bound solution to the recurrence

$$T(n) = 3T(n/4) + \Theta(n^2) . \tag{4.13}$$

Figure 4.1 shows how to derive the recursion tree for $T(n) = 3T(n/4) + cn^2$, where the constant $c > 0$ is the upper-bound constant in the $\Theta(n^2)$ term. Part (a) of the figure shows $T(n)$, which part (b) expands into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
$c\left(\frac{n}{16}\right)^2$
$\Theta(1)$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$\log_4 n$
$3^{\log_4 n} = n^{\log_4 3}$
Total: $O(n^2)$
(a)
(b)
(c)
(d)

Figure 4.1 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in **(d)** has height $\log_4 n$.

subproblems of size $n/4$. Part (c) shows this process carried one step further by expanding each node with cost $T(n/4)$ from part (b). The cost for each of the three children of the root is $c(n/4)^2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

Because subproblem sizes decrease by a factor of 4 every time we go down one level, the recursion must eventually bottom out in a base case where $n < n_0$. By convention, the base case is $T(n) = \Theta(1)$ for $n < n_0$, where $n_0 > 0$ is any threshold constant sufficiently large that the recurrence is well defined. For the purpose of intuition, however, let's simplify the math a little. Let's assume that n is an exact power of 4 and that the base case is $T(1) = \Theta(1)$. As it turns out, these assumptions don't affect the asymptotic solution.

What's the height of the recursion tree? The subproblem size for a node at depth i is $n/4^i$. As we descend the tree from the root, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has internal nodes at depths $0, 1, 2, \ldots, \log_4 n - 1$ and leaves at depth $\log_4 n$.

Part (d) of Figure 4.1 shows the cost at each level of the tree. Each level has three times as many nodes as the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level further from the root, each internal node at depth $i = 0, 1, 2, \ldots, \log_4 n - 1$ has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost of all nodes at a given depth i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, contains $3^{\log_4 n} = n^{\log_4 3}$ leaves (using equation (3.21) on page 66). Each leaf contributes $\Theta(1)$, leading to a total leaf cost of $\Theta(n^{\log_4 3})$.

Now we add up the costs over all levels to determine the cost for the entire tree:

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) && \text{(by equation (A.7) on page 1142)} \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) && (\Theta(n^{\log_4 3}) = O(n^{0.8}) = O(n^2))\ .
\end{aligned}$$

We've derived the guess of $T(n) = O(n^2)$ for the original recurrence. In this example, the coefficients of cn^2 form a decreasing geometric series. By equation (A.7), the sum of these coefficients is bounded from above by the constant $16/13$. Since the root's contribution to the total cost is cn^2, the cost of the root dominates the total cost of the tree.

In fact, if $O(n^2)$ is indeed an upper bound for the recurrence (as we'll verify in a moment), then it must be a tight bound. Why? The first recursive call contributes a cost of $\Theta(n^2)$, and so $\Omega(n^2)$ must be a lower bound for the recurrence.

Let's now use the substitution method to verify that our guess is correct, namely, that $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(n/4) + \Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\le 3T(n/4) + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

where the last step holds if we choose $d \ge (16/13)c$.

For the base case of the induction, let $n_0 > 0$ be a sufficiently large threshold constant that the recurrence is well defined when $T(n) = \Theta(1)$ for $n < n_0$. We can pick d large enough that d dominates the constant hidden by the Θ, in which case $dn^2 \ge d \ge T(n)$ for $1 \le n < n_0$, completing the proof of the base case.

The substitution proof we just saw involves two named constants, c and d. We named c and used it to stand for the upper-bound constant hidden and guaranteed to exist by the Θ-notation. We cannot pick c arbitrarily—it's given to us—although, for any such c, any constant $c' \ge c$ also suffices. We also named d, but we were free to choose any value for it that fit our needs. In this example, the value of d happened to depend on the value of c, which is fine, since d is constant if c is constant.

An irregular example

Let's find an asymptotic upper bound for another, more irregular, example. Figure 4.2 shows the recursion tree for the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) . \tag{4.14}$$

This recursion tree is unbalanced, with different root-to-leaf paths having different lengths. Going left at any node produces a subproblem of one-third the size, and going right produces a subproblem of two-thirds the size. Let $n_0 > 0$ be the implicit threshold constant such that $T(n) = \Theta(1)$ for $0 < n < n_0$, and let c represent the upper-bound constant hidden by the $\Theta(n)$ term for $n \ge n_0$. There are actually two n_0 constants here—one for the threshold in the recurrence, and the other for the threshold in the Θ-notation, so we'll let n_0 be the larger of the two constants.

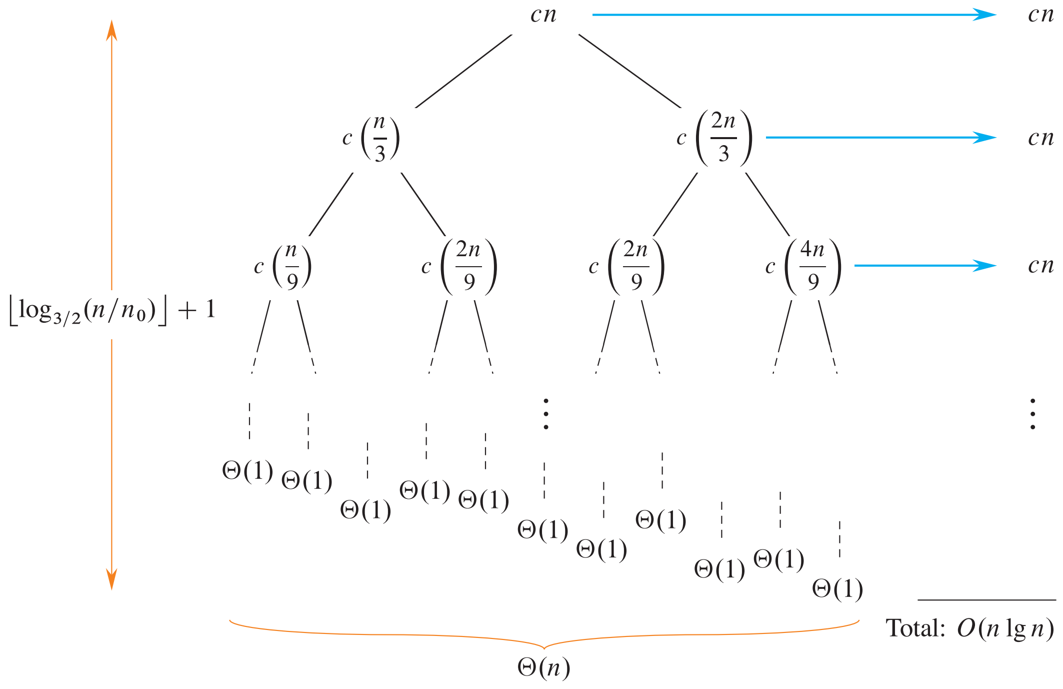

cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
cn
cn
$\lfloor \log_{3/2}(n/n_0) \rfloor + 1$
$\Theta(1)$
$\Theta(n)$
Total: $O(n \lg n)$

Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

The height of the tree runs down the right edge of the tree, corresponding to subproblems of sizes $n, (2/3)n, (4/9)n, \ldots, \Theta(1)$ with costs bounded by $cn, c(2n/3), c(4n/9), \ldots, \Theta(1)$, respectively. We hit the rightmost leaf when $(2/3)^h n < n_0 \le (2/3)^{h-1} n$, which happens when $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$ since, applying the floor bounds in equation (3.2) on page 64 with $x = \log_{3/2}(n/n_0)$, we have $(2/3)^h n = (2/3)^{\lfloor x \rfloor + 1} n < (2/3)^x n = (n_0/n)n = n_0$ and $(2/3)^{h-1} n = (2/3)^{\lfloor x \rfloor} n > (2/3)^x n = (n_0/n)n = n_0$. Thus, the height of the tree is $h = \Theta(\lg n)$.

We're now in a position to understand the upper bound. Let's postpone dealing with the leaves for a moment. Summing the costs of internal nodes across each level, we have at most cn per level times the $\Theta(\lg n)$ tree height for a total cost of $O(n \lg n)$ for all internal nodes.

It remains to deal with the leaves of the recursion tree, which represent base cases, each costing $\Theta(1)$. How many leaves are there? It's tempting to upper-bound their number by the number of leaves in a complete binary tree of height $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$, since the recursion tree is contained within such a complete binary tree. But this approach turns out to give us a poor bound. The complete binary tree has 1 node at the root, 2 nodes at depth 1, and generally 2^k nodes at depth k. Since the height is $h = \lfloor \log_{3/2} n \rfloor + 1$, there are

$2^h = 2^{\lfloor \log_{3/2} n \rfloor + 1} \le 2n^{\log_{3/2} 2}$ leaves in the complete binary tree, which is an upper bound on the number of leaves in the recursion tree. Because the cost of each leaf is $\Theta(1)$, this analysis says that the total cost of all leaves in the recursion tree is $O(n^{\log_{3/2} 2}) = O(n^{1.71})$, which is an asymptotically greater bound than the $O(n \lg n)$ cost of all internal nodes. In fact, as we're about to see, this bound is not tight. The cost of all leaves in the recursion tree is $O(n)$—asymptotically *less* than $O(n \lg n)$. In other words, the cost of the internal nodes dominates the cost of the leaves, not vice versa.

Rather than analyzing the leaves, we could quit right now and prove by substitution that $T(n) = \Theta(n \lg n)$. This approach works (see Exercise 4.4-3), but it's instructive to understand how many leaves this recursion tree has. You may see recurrences for which the cost of leaves dominates the cost of internal nodes, and then you'll be in better shape if you've had some experience analyzing the number of leaves.

To figure out how many leaves there really are, let's write a recurrence $L(n)$ for the number of leaves in the recursion tree for $T(n)$. Since all the leaves in $T(n)$ belong either to the left subtree or the right subtree of the root, we have

$$L(n) = \begin{cases} 1 & \text{if } n < n_0 , \\ L(n/3) + L(2n/3) & \text{if } n \ge n_0 . \end{cases} \tag{4.15}$$

This recurrence is similar to recurrence (4.14), but it's missing the $\Theta(n)$ term, and it contains an explicit base case. Because this recurrence omits the $\Theta(n)$ term, it is much easier to solve. Let's apply the substitution method to show that it has solution $L(n) = O(n)$. Using the inductive hypothesis $L(n) \le dn$ for some constant $d > 0$, and assuming that the inductive hypothesis holds for all values less than n, we have

$$\begin{aligned} L(n) &= L(n/3) + L(2n/3) \\ &\le dn/3 + 2(dn)/3 \\ &\le dn , \end{aligned}$$

which holds for any $d > 0$. We can now choose d large enough to handle the base case $L(n) = 1$ for $0 < n < n_0$, for which $d = 1$ suffices, thereby completing the substitution method for the upper bound on leaves. (Exercise 4.4-2 asks you to prove that $L(n) = \Theta(n)$.)

Returning to recurrence (4.14) for $T(n)$, it now becomes apparent that the total cost of leaves over all levels must be $L(n) \cdot \Theta(1) = \Theta(n)$. Since we have derived the bound of $O(n \lg n)$ on the cost of the internal nodes, it follows that the solution to recurrence (4.14) is $T(n) = O(n \lg n) + \Theta(n) = O(n \lg n)$. (Exercise 4.4-3 asks you to prove that $T(n) = \Theta(n \lg n)$.)

It's wise to verify any bound obtained with a recursion tree by using the substitution method, especially if you've made simplifying assumptions. But another strategy altogether is to use more-powerful mathematics, typically in the form of the master method in the next section (which unfortunately doesn't apply to recurrence (4.14)) or the Akra-Bazzi method (which does, but requires calculus). Even if you use a powerful method, a recursion tree can improve your intuition for what's going on beneath the heavy math.

Exercises

4.4-1

For each of the following recurrences, sketch its recursion tree, and guess a good asymptotic upper bound on its solution. Then use the substitution method to verify your answer.

a. $T(n) = T(n/2) + n^3$.

b. $T(n) = 4T(n/3) + n$.

c. $T(n) = 4T(n/2) + n$.

d. $T(n) = 3T(n-1) + 1$.

4.4-2

Use the substitution method to prove that recurrence (4.15) has the asymptotic lower bound $L(n) = \Omega(n)$. Conclude that $L(n) = \Theta(n)$.

4.4-3

Use the substitution method to prove that recurrence (4.14) has the solution $T(n) = \Omega(n \lg n)$. Conclude that $T(n) = \Theta(n \lg n)$.

4.4-4

Use a recursion tree to justify a good guess for the solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(n)$, where α is a constant in the range $0 < \alpha < 1$.

4.5 The master method for solving recurrences

The master method provides a "cookbook" method for solving algorithmic recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.16}$$

where $a > 0$ and $b > 1$ are constants. We call $f(n)$ a ***driving function***, and we call a recurrence of this general form a ***master recurrence***. To use the master method, you need to memorize three cases, but then you'll be able to solve many master recurrences quite easily.

A master recurrence describes the running time of a divide-and-conquer algorithm that divides a problem of size n into a subproblems, each of size $n/b<n$. The algorithm solves the a subproblems recursively, each in $T(n/b)$ time. The driving function $f(n)$ encompasses the cost of dividing the problem before the recursion, as well as the cost of combining the results of the recursive solutions to subproblems. For example, the recurrence arising from Strassen's algorithm is a master recurrence with $a = 7, b = 2$, and driving function $f(n) = \Theta(n^2)$.

As we have mentioned, in solving a recurrence that describes the running time of an algorithm, one technicality that we'd often prefer to ignore is the requirement that the input size n be an integer. For example, we saw that the running time of merge sort can be described by recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41. But if n is an odd number, we really don't have two problems of exactly half the size. Rather, to ensure that the problem sizes are integers, we round one subproblem down to size $\lfloor n/2 \rfloor$ and the other up to size $\lceil n/2 \rceil$, so the true recurrence is $T(n) = T(\lceil n/2 \rceil + T(\lfloor n/2 \rfloor) + \Theta(n)$. But this floors-and-ceilings recurrence is longer to write and messier to deal with than recurrence (2.3), which is defined on the reals. We'd rather not worry about floors and ceilings, if we don't have to, especially since the two recurrences have the same $\Theta(n \lg n)$ solution.

The master method allows you to state a master recurrence without floors and ceilings and implicitly infer them. No matter how the arguments are rounded up or down to the nearest integer, the asymptotic bounds that it provides remain the same. Moreover, as we'll see in Section 4.6, if you define your master recurrence on the reals, without implicit floors and ceilings, the asymptotic bounds still don't change. Thus you can ignore floors and ceilings for master recurrences. Section 4.7 gives sufficient conditions for ignoring floors and ceilings in more general divide-and-conquer recurrences.

The master theorem

The master method depends upon the following theorem.

Theorem 4.1 (Master theorem)

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n) , \tag{4.17}$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Before applying the master theorem to some examples, let's spend a few moments to understand broadly what it says. The function $n^{\log_b a}$ is called the ***watershed function***. In each of the three cases, we compare the driving function $f(n)$ to the watershed function $n^{\log_b a}$. Intuitively, if the watershed function grows asymptotically faster than the driving function, then case 1 applies. Case 2 applies if the two functions grow at nearly the same asymptotic rate. Case 3 is the "opposite" of case 1, where the driving function grows asymptotically faster than the watershed function. But the technical details matter.

In case 1, not only must the watershed function grow asymptotically faster than the driving function, it must grow *polynomially* faster. That is, the watershed function $n^{\log_b a}$ must be asymptotically larger than the driving function $f(n)$ by at least a factor of $\Theta(n^\epsilon)$ for some constant $\epsilon > 0$. The master theorem then says that the solution is $T(n) = \Theta(n^{\log_b a})$. In this case, if we look at the recursion tree for the recurrence, the cost per level grows at least geometrically from root to leaves, and the total cost of leaves dominates the total cost of the internal nodes.

In case 2, the watershed and driving functions grow at nearly the same asymptotic rate. But more specifically, the driving function grows faster than the watershed function by a factor of $\Theta(\lg^k n)$, where $k \geq 0$. The master theorem says that we tack on an extra $\lg n$ factor to $f(n)$, yielding the solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. In this case, each level of the recursion tree costs approximately the same—$\Theta(n^{\log_b a} \lg^k n)$—and there are $\Theta(\lg n)$ levels. In practice, the most common situation for case 2 occurs when $k = 0$, in which case the watershed and driving functions have the same asymptotic growth, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3 mirrors case 1. Not only must the driving function grow asymptotically faster than the watershed function, it must grow *polynomially* faster. That is, the driving function $f(n)$ must be asymptotically larger than the watershed function $n^{\log_b a}$ by at least a factor of $\Theta(n^\epsilon)$ for some constant $\epsilon > 0$. Moreover, the driving function must satisfy the regularity condition that $af(n/b) \leq cf(n)$. This condition is satisfied by most of the polynomially bounded functions that you're likely to encounter when applying case 3. The regularity condition might not be satisfied if the driving function grows slowly in local areas, yet relatively quickly overall. (Exercise 4.5-5 gives an example of such a function.) For case 3, the master theorem says that the solution is $T(n) = \Theta(f(n))$. If we look at the recursion tree, the cost per level drops at least geometrically from the root to the leaves, and the root cost dominates the cost of all other nodes.

It's worth looking again at the requirement that there be polynomial separation between the watershed function and the driving function for either case 1 or case 3 to apply. The separation doesn't need to be much, but it must be there, and it must grow polynomially. For example, for the recurrence $T(n) = 4T(n/2) + n^{1.99}$ (admittedly not a recurrence you're likely to see when analyzing an algorithm), the watershed function is $n^{\log_b a} = n^2$. Hence the driving function $f(n) = n^{1.99}$ is polynomially smaller by a factor of $n^{0.01}$. Thus case 1 applies with $\epsilon = 0.01$.

Using the master method

To use the master method, you determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider the recurrence $T(n) = 9T(n/3) + n$. For this recurrence, we have $a = 9$ and $b = 3$, which implies that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = n = O(n^{2-\epsilon})$ for any constant $\epsilon \le 1$, we can apply case 1 of the master theorem to conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider the recurrence $T(n) = T(2n/3) + 1$, which has $a = 1$ and $b = 3/2$, which means that the watershed function is $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies since $f(n) = 1 = \Theta(n^{\log_b a} \lg^0 n) = \Theta(1)$. The solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence $T(n) = 3T(n/4) + n \lg n$, we have $a = 3$ and $b = 4$, which means that $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = n \lg n = \Omega(n^{\log_4 3+\epsilon})$, where ϵ can be as large as approximately 0.2, case 3 applies as long as the regularity condition holds for $f(n)$. It does, because for sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. By case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Next, let's look at the recurrence $T(n) = 2T(n/2) + n \lg n$, where we have $a = 2$, $b = 2$, and $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies since $f(n) = n \lg n = \Theta(n^{\log_b a} \lg^1 n)$. We conclude that the solution is $T(n) = \Theta(n \lg^2 n)$.

We can use the master method to solve the recurrences we saw in Sections 2.3.2, 4.1, and 4.2.

Recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41, characterizes the running time of merge sort. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies because $f(n) = \Theta(n)$, and the solution is $T(n) = \Theta(n \lg n)$.

Recurrence (4.9), $T(n) = 8T(n/2) + \Theta(1)$, on page 84, describes the running time of the simple recursive algorithm for matrix multiplication. We have $a = 8$ and $b = 2$, which means that the watershed function is $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than the driving function $f(n) = \Theta(1)$—indeed, we have $f(n) = O(n^{3-\epsilon})$ for any positive $\epsilon < 3$—case 1 applies. We conclude that $T(n) = \Theta(n^3)$.

Finally, recurrence (4.10), $T(n) = 7T(n/2) + \Theta(n^2)$, on page 87, arose from the analysis of Strassen's algorithm for matrix multiplication. For this recurrence, we have $a = 7$ and $b = 2$, and the watershed function is $n^{\log_b a} = n^{\lg 7}$. Observing that $\lg 7 = 2.807355\ldots$, we can let $\epsilon = 0.8$ and bound the driving function $f(n) = \Theta(n^2) = O(n^{\lg 7-\epsilon})$. Case 1 applies with solution $T(n) = \Theta(n^{\lg 7})$.

When the master method doesn't apply

There are situations where you can't use the master theorem. For example, it can be that the watershed function and the driving function cannot be asymptotically compared. We might have that $f(n) \gg n^{\log_b a}$ for an infinite number of values of n but also that $f(n) \ll n^{\log_b a}$ for an infinite number of different values of n. As a practical matter, however, most of the driving functions that arise in the study of algorithms can be meaningfully compared with the watershed function. If you encounter a master recurrence for which that's not the case, you'll have to resort to substitution or other methods.

Even when the relative growths of the driving and watershed functions can be compared, the master theorem does not cover all the possibilities. There is a gap between cases 1 and 2 when $f(n) = o(n^{\log_b a})$, yet the watershed function does not grow polynomially faster than the driving function. Similarly, there is a gap between cases 2 and 3 when $f(n) = \omega(n^{\log_b a})$ and the driving function grows more than polylogarithmically faster than the watershed function, but it does not grow polynomially faster. If the driving function falls into one of these gaps, or if the regularity condition in case 3 fails to hold, you'll need to use something other than the master method to solve the recurrence.

As an example of a driving function falling into a gap, consider the recurrence $T(n) = 2T(n/2) + n/\lg n$. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. The driving function is $n/\lg n = o(n)$, which means that it grows asymptotically more slowly than the watershed function n. But $n/\lg n$ grows only *logarithmically* slower than n, not *polynomially* slower. More precisely, equation (3.24) on page 67 says that $\lg n = o(n^\epsilon)$ for any constant $\epsilon > 0$, which means that $1/\lg n = \omega(n^{-\epsilon})$ and $n/\lg n = \omega(n^{1-\epsilon}) = \omega(n^{\log_b a-\epsilon})$. Thus no constant $\epsilon > 0$ exists such that $n/\lg n = O(n^{\log_b a-\epsilon})$, which is required for case 1 to apply. Case 2 fails to apply as well, since $n/\lg n = \Theta(n^{\log_b a}\lg^k n)$, where $k = -1$, but k must be nonnegative for case 2 to apply.

To solve this kind of recurrence, you must use another method, such as the substitution method (Section 4.3) or the Akra-Bazzi method (Section 4.7). (Exercise 4.6-3 asks you to show that the answer is $\Theta(n \lg \lg n)$.) Although the master theorem doesn't handle this particular recurrence, it does handle the overwhelming majority of recurrences that tend to arise in practice.

Exercises

4.5-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1$.

b. $T(n) = 2T(n/4) + \sqrt{n}$.

c. $T(n) = 2T(n/4) + \sqrt{n} \lg^2 n$.

d. $T(n) = 2T(n/4) + n$.

e. $T(n) = 2T(n/4) + n^2$.

4.5-2

Professor Caesar wants to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into $n/4 \times n/4$ submatrices, and the divide and combine steps together will take $\Theta(n^2)$ time. Suppose that the professor's algorithm creates a recursive subproblems of size $n/4$. What is the largest integer value of a for which his algorithm could possibly run asymptotically faster than Strassen's?

4.5-3

Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-6 for a description of binary search.)

4.5-4

Consider the function $f(n) = \lg n$. Argue that although $f(n/2) < f(n)$, the regularity condition $af(n/b) \le cf(n)$ with $a = 1$ and $b = 2$ does not hold for any constant $c < 1$. Argue further that for any $\epsilon > 0$, the condition in case 3 that $f(n) = \Omega(n^{\log_b a + \epsilon})$ does not hold.

4.5-5

Show that for suitable constants a, b, and ϵ, the function $f(n) = 2^{\lceil \lg n \rceil}$ satisfies all the conditions in case 3 of the master theorem except the regularity condition.

★ 4.6 Proof of the continuous master theorem

Proving the master theorem (Theorem 4.1) in its full generality, especially dealing with the knotty technical issue of floors and ceilings, is beyond the scope of this book. This section, however, states and proves a variant of the master theorem, called the ***continuous master theorem***[1] in which the master recurrence (4.17) is defined over sufficiently large positive real numbers. The proof of this version, uncomplicated by floors and ceilings, contains the main ideas needed to understand how master recurrences behave. Section 4.7 discusses floors and ceilings in divide-and-conquer recurrences at greater length, presenting sufficient conditions for them not to affect the asymptotic solutions.

Of course, since you need not understand the proof of the master theorem in order to apply the master method, you may choose to skip this section. But if you wish to study more-advanced algorithms beyond the scope of this textbook, you may appreciate a better understanding of the underlying mathematics, which the proof of the continuous master theorem provides.

Although we usually assume that recurrences are algorithmic and don't require an explicit statement of a base case, we must be much more careful for proofs that justify the practice. The lemmas and theorem in this section explicitly state the base cases, because the inductive proofs require mathematical grounding. It is common in the world of mathematics to be extraordinarily careful proving theorems that justify acting more casually in practice.

The proof of the continuous master theorem involves two lemmas. Lemma 4.2 uses a slightly simplified master recurrence with a threshold constant of $n_0 = 1$, rather than the more general $n_0 > 0$ threshold constant implied by the unstated base case. The lemma employs a recursion tree to reduce the solution of the simplified master recurrence to that of evaluating a summation. Lemma 4.3 then provides asymptotic bounds for the summation, mirroring the three cases of the master theorem. Finally, the continuous master theorem itself (Theorem 4.4) gives asymptotic bounds for master recurrences, while generalizing to an arbitrary threshold constant $n_0 > 0$ as implied by the unstated base case.

[1] This terminology does not mean that either $T(n)$ or $f(n)$ need be continuous, only that the domain of $T(n)$ is the real numbers, as opposed to integers.

Some of the proofs use the properties described in Problem 3-5 on pages 72–73 to combine and simplify complicated asymptotic expressions. Although Problem 3-5 addresses only Θ-notation, the properties enumerated there can be extended to O-notation and Ω-notation as well.

Here's the first lemma.

Lemma 4.2
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } 0 \leq n < 1 , \\ aT(n/b) + f(n) & \text{if } n \geq 1 \end{cases}$$

has solution

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) . \tag{4.18}$$

Proof Consider the recursion tree in Figure 4.3. Let's look first at its internal nodes. The root of the tree has cost $f(n)$, and it has a children, each with cost $f(n/b)$. (It is convenient to think of a as being an integer, especially when visualizing the recursion tree, but the mathematics does not require it.) Each of these children has a children, making a^2 nodes at depth 2, and each of the a children has cost $f(n/b^2)$. In general, there are a^j nodes at depth j, and each node has cost $f(n/b^j)$.

Now, let's move on to understanding the leaves. The tree grows downward until n/b^j becomes less than 1. Thus, the tree has height $\lfloor \log_b n \rfloor + 1$, because $n/b^{\lfloor \log_b n \rfloor} \geq n/b^{\log_b n} = 1$ and $n/b^{\lfloor \log_b n \rfloor + 1} < n/b^{\log_b n} = 1$. Since, as we have observed, the number of nodes at depth j is a^j and all the leaves are at depth $\lfloor \log_b n \rfloor + 1$, the tree contains $a^{\lfloor \log_b n \rfloor + 1}$ leaves. Using the identity (3.21) on page 66, we have $a^{\lfloor \log_b n \rfloor + 1} \leq a^{\log_b n + 1} = an^{\log_b a} = O(n^{\log_b a})$, since a is constant, and $a^{\lfloor \log_b n \rfloor + 1} \geq a^{\log_b n} = n^{\log_b a} = \Omega(n^{\log_b a})$. Consequently, the total number of leaves is $\Theta(n^{\log_b a})$—asymptotically, the watershed function.

We are now in a position to derive equation (4.18) by summing the costs of the nodes at each depth in the tree, as shown in the figure. The first term in the equation is the total costs of the leaves. Since each leaf is at depth $\lfloor \log_b n \rfloor + 1$ and $n/b^{\lfloor \log_b n \rfloor + 1} < 1$, the base case of the recurrence gives the cost of a leaf: $T(n/b^{\lfloor \log_b n \rfloor + 1}) = \Theta(1)$. Hence the cost of all $\Theta(n^{\log_b a})$ leaves is $\Theta(n^{\log_b a}) \cdot \Theta(1) = \Theta(n^{\log_b a})$ by Problem 3-5(d). The second term in equation (4.18) is the cost of the internal nodes, which, in the underlying divide-and-conquer algorithm, represents the costs of dividing problems into subproblems and

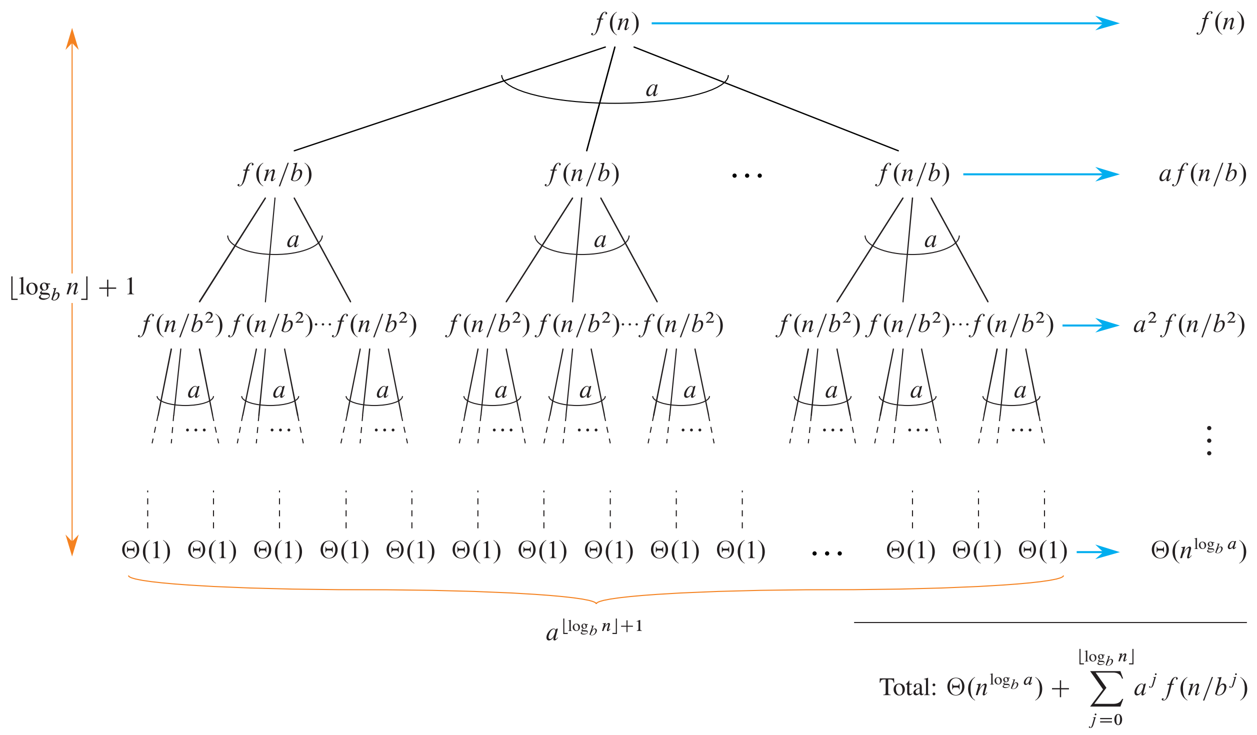

$f(n)$
a
$f(n/b)$
$f(n/b)$
$\cdots$
$f(n/b)$
$af(n/b)$
$\lfloor \log_b n \rfloor + 1$
$f(n/b^2)$
$a^2 f(n/b^2)$
$\Theta(1)$
$\Theta(n^{\log_b a})$
$a^{\lfloor \log_b n \rfloor + 1}$
Total: $\Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j)$

Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $a^{\lfloor \log_b n \rfloor + 1}$ leaves and height $\lfloor \log_b n \rfloor + 1$. The cost of the nodes at each depth is shown at the right, and their sum is given in equation (4.18).

then recombining the subproblems. Since the cost for all the internal nodes at depth j is $a^j f(n/b^j)$, the total cost of all internal nodes is

$$\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) . \qquad \blacksquare$$

As we'll see, the three cases of the master theorem depend on the distribution of the total cost across levels of the recursion tree:

Case 1: The costs increase geometrically from the root to the leaves, growing by a constant factor with each level.

Case 2: The costs depend on the value of k in the theorem. With $k = 0$, the costs are equal for each level; with $k = 1$, the costs grow linearly from the root to the leaves; with $k = 2$, the growth is quadratic; and in general, the costs grow polynomially in k.

Case 3: The costs decrease geometrically from the root to the leaves, shrinking by a constant factor with each level.

The summation in equation (4.18) describes the cost of the dividing and combining steps in the underlying divide-and-conquer algorithm. The next lemma provides asymptotic bounds on the summation's growth.

Lemma 4.3
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the asymptotic behavior of the function

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \,, \tag{4.19}$$

defined for $n \geq 1$, can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $g(n) = O(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $g(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant c in the range $0 < c < 1$ such that $0 < af(n/b) \leq cf(n)$ for all $n \geq 1$, then $g(n) = \Theta(f(n))$.

Proof For case 1, we have $f(n) = O(n^{\log_b a - \epsilon})$, which implies that $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$. Substituting into equation (4.19) yields

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) \\
&= O\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) && \text{(by Problem 3-5(c), repeatedly)} \\
&= O\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j\right) \\
&= O\left(n^{\log_b a - \epsilon} \sum_{j=0}^{\lfloor \log_b n \rfloor} (b^\epsilon)^j\right) && \text{(by equation (3.17) on page 66)} \\
&= O\left(n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon(\lfloor \log_b n \rfloor + 1)} - 1}{b^\epsilon - 1}\right)\right) && \text{(by equation (A.6) on page 1142)} \,,
\end{aligned}$$

the last series being geometric. Since b and ϵ are constants, the $b^\epsilon - 1$ denominator doesn't affect the asymptotic growth of $g(n)$, and neither does the -1 in the numerator. Since $b^{\epsilon(\lfloor \log_b n \rfloor + 1)} \le (b^{\log_b n + 1})^{\epsilon} = b^{\epsilon} n^{\epsilon} = O(n^{\epsilon})$, we obtain $g(n) = O(n^{\log_b a - \epsilon} \cdot O(n^{\epsilon})) = O(n^{\log_b a})$, thereby proving case 1.

Case 2 assumes that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, from which we can conclude that $f(n/b^j) = \Theta((n/b^j)^{\log_b a} \lg^k (n/b^j))$. Substituting into equation (4.19) and repeatedly applying Problem 3-5(c) yields

$$\begin{aligned}
g(n) &= \Theta\left(\sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \frac{a^j}{b^{j \log_b a}} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \lg^k \left(\frac{n}{b^j}\right)\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b (n/b^j)}{\log_b 2}\right)^k\right) && \text{(by equation (3.19) on page 66)} \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} \left(\frac{\log_b n - j}{\log_b 2}\right)^k\right) && \text{(by equations (3.17), (3.18), and (3.20))} \\
&= \Theta\left(\frac{n^{\log_b a}}{\log_b^k 2} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k\right) && \text{($b > 1$ and k are constants)} .
\end{aligned}$$

The summation within the Θ-notation can be bounded from above as follows:

$$\begin{aligned}
\sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k &\le \sum_{j=0}^{\lfloor \log_b n \rfloor} (\lfloor \log_b n \rfloor + 1 - j)^k \\
&= \sum_{j=1}^{\lfloor \log_b n \rfloor + 1} j^k && \text{(reindexing—pages 1143–1144)} \\
&= O((\lfloor \log_b n \rfloor + 1)^{k+1}) && \text{(by Exercise A.1-5 on page 1144)} \\
&= O(\log_b^{k+1} n) && \text{(by Exercise 3.3-3 on page 70)} .
\end{aligned}$$

Exercise 4.6-1 asks you to show that the summation can similarly be bounded from below by $\Omega(\log_b^{k+1} n)$. Since we have tight upper and lower bounds, the summation is $\Theta(\log_b^{k+1} n)$, from which we can conclude that $g(n) = \Theta\left(n^{\log_b a} \log_b^{k+1} n\right)$, thereby completing the proof of case 2.

For case 3, observe that $f(n)$ appears in the definition (4.19) of $g(n)$ (when $j = 0$) and that all terms of $g(n)$ are positive. Therefore, we must have $g(n) = \Omega(f(n))$, and it only remains to prove that $g(n) = O(f(n))$. Performing j iterations of the inequality $af(n/b) \le cf(n)$ yields $a^j f(n/b^j) \le c^j f(n)$. Substituting into equation (4.19), we obtain

$$\begin{aligned} g(n) &= \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\lfloor \log_b n \rfloor} c^j f(n) \\ &\le f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n)\left(\frac{1}{1-c}\right) \qquad \text{(by equation (A.7) on page 1142 since } |c| < 1) \\ &= O(f(n)) . \end{aligned}$$

Thus, we can conclude that $g(n) = \Theta(f(n))$. With case 3 proved, the entire proof of the lemma is complete. ■

We can now state and prove the continuous master theorem.

Theorem 4.4 (Continuous master theorem)
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the algorithmic recurrence $T(n)$ on the positive real numbers by

$$T(n) = aT(n/b) + f(n) .$$

Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \ge 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Proof The idea is to bound the summation (4.18) from Lemma 4.2 by applying Lemma 4.3. But we must account for Lemma 4.2 using a base case for $0 < n < 1$, whereas this theorem uses an implicit base case for $0 < n < n_0$, where $n_0 > 0$ is an arbitrary threshold constant. Since the recurrence is algorithmic, we can assume that $f(n)$ is defined for $n \geq n_0$.

For $n > 0$, let us define two auxiliary functions $T'(n) = T(n_0 n)$ and $f'(n) = f(n_0 n)$. We have

$$\begin{aligned} T'(n) &= T(n_0 n) \\ &= \begin{cases} \Theta(1) & \text{if } n_0 n < n_0 , \\ aT(n_0 n/b) + f(n_0 n) & \text{if } n_0 n \geq n_0 \end{cases} \\ &= \begin{cases} \Theta(1) & \text{if } n < 1 , \\ aT'(n/b) + f'(n) & \text{if } n \geq 1 . \end{cases} \end{aligned} \tag{4.20}$$

We have obtained a recurrence for $T'(n)$ that satisfies the conditions of Lemma 4.2, and by that lemma, the solution is

$$T'(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j f'(n/b^j) . \tag{4.21}$$

To solve $T'(n)$, we first need to bound $f'(n)$. Let's examine the individual cases in the theorem.

The condition for case 1 is $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. We have

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= O((n_0 n)^{\log_b a - \epsilon}) \\ &= O(n^{\log_b a - \epsilon}) , \end{aligned}$$

since a, b, n_0, and ϵ are all constant. The function $f'(n)$ satisfies the conditions of case 1 of Lemma 4.3, and the summation in equation (4.18) of Lemma 4.2 evaluates to $O(n^{\log_b a})$. Because a, b and n_0 are all constants, we have

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + O((n/n_0)^{\log_b a}) \\ &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \qquad \text{(by Problem 3-5(b))} , \end{aligned}$$

thereby completing case 1 of the theorem.

The condition for case 2 is $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$. We have

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Theta((n_0 n)^{\log_b a} \lg^k (n_0 n)) \\ &= \Theta(n^{\log_b a} \lg^k n) \qquad \text{(by eliminating the constant terms)} . \end{aligned}$$

Similar to the proof of case 1, the function $f'(n)$ satisfies the conditions of case 2 of Lemma 4.3. The summation in equation (4.18) of Lemma 4.2 is therefore $\Theta(n^{\log_b a} \lg^{k+1} n)$, which implies that

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta((n/n_0)^{\log_b a} \lg^{k+1}(n/n_0)) \\ &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg^{k+1} n) \\ &= \Theta(n^{\log_b a} \lg^{k+1} n) \qquad \text{(by Problem 3-5(c))} , \end{aligned}$$

which proves case 2 of the theorem.

Finally, the condition for case 3 is $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ additionally satisfies the regularity condition $af(n/b) \le cf(n)$ for all $n \ge n_0$ and some constants $c < 1$ and $n_0 > 1$. The first part of case 3 is like case 1:

$$\begin{aligned} f'(n) &= f(n_0 n) \\ &= \Omega((n_0 n)^{\log_b a + \epsilon}) \\ &= \Omega(n^{\log_b a + \epsilon}) . \end{aligned}$$

Using the definition of $f'(n)$ and the fact that $n_0 n \ge n_0$ for all $n \ge 1$, we have for $n \ge 1$ that

$$\begin{aligned} af'(n/b) &= af(n_0 n/b) \\ &\le cf(n_0 n) \\ &= cf'(n) . \end{aligned}$$

Thus $f'(n)$ satisfies the requirements for case 3 of Lemma 4.3, and the summation in equation (4.18) of Lemma 4.2 evaluates to $\Theta(f'(n))$, yielding

$$\begin{aligned} T(n) &= T'(n/n_0) \\ &= \Theta((n/n_0)^{\log_b a}) + \Theta(f'(n/n_0)) \\ &= \Theta(f'(n/n_0)) \\ &= \Theta(f(n)) , \end{aligned}$$

which completes the proof of case 3 of the theorem and thus the whole theorem. ■

Exercises

4.6-1

Show that $\sum_{j=0}^{\lfloor \log_b n \rfloor} (\log_b n - j)^k = \Omega(\log_b^{k+1} n)$.

★ ***4.6-2***

Show that case 3 of the master theorem is overstated (which is also why case 3 of Lemma 4.3 does not require that $f(n) = \Omega(n^{\log_b a + \epsilon})$) in the sense that the regularity condition $af(n/b) \le cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

★ ***4.6-3***

For $f(n) = \Theta(n^{\log_b a}/\lg n)$, prove that the summation in equation (4.19) has solution $g(n) = \Theta(n^{\log_b a} \lg \lg n)$. Conclude that a master recurrence $T(n)$ using $f(n)$ as its driving function has solution $T(n) = \Theta(n^{\log_b a} \lg \lg n)$.

★ 4.7 Akra-Bazzi recurrences

This section provides an overview of two advanced topics related to divide-and-conquer recurrences. The first deals with technicalities arising from the use of floors and ceilings, and the second discusses the Akra-Bazzi method, which involves a little calculus, for solving complicated divide-and-conquer recurrences.

In particular, we'll look at the class of algorithmic divide-and-conquer recurrences originally studied by M. Akra and L. Bazzi [13]. These ***Akra-Bazzi*** recurrences take the form

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(n/b_i) , \tag{4.22}$$

where k is a positive integer; all the constants $a_1, a_2, \ldots, a_k \in \mathbb{R}$ are strictly positive; all the constants $b_1, b_2, \ldots, b_k \in \mathbb{R}$ are strictly greater than 1; and the driving function $f(n)$ is defined on sufficiently large nonnegative reals and is itself nonnegative.

Akra-Bazzi recurrences generalize the class of recurrences addressed by the master theorem. Whereas master recurrences characterize the running times of divide-and-conquer algorithms that break a problem into equal-sized subproblems (modulo floors and ceilings), Akra-Bazzi recurrences can describe the running time of divide-and-conquer algorithms that break a problem into different-sized subproblems. The master theorem, however, allows you to ignore floors and ceilings, but the Akra-Bazzi method for solving Akra-Bazzi recurrences needs an additional requirement to deal with floors and ceilings.

But before diving into the Akra-Bazzi method itself, let's understand the limitations involved in ignoring floors and ceilings in Akra-Bazzi recurrences. As you're aware, algorithms generally deal with integer-sized inputs. The mathematics for recurrences is often easier with real numbers, however, than with integers, where we must cope with floors and ceilings to ensure that terms are well defined. The difference may not seem to be much—especially because that's often the truth with recurrences—but to be mathematically correct, we must be careful with our assumptions. Since our end goal is to understand algorithms and not the vagaries of mathematical corner cases, we'd like to be casual yet rigorous. How can we treat floors and ceilings casually while still ensuring rigor?

From a mathematical point of view, the difficulty in dealing with floors and ceilings is that some driving functions can be really, really weird. So it's not okay in general to ignore floors and ceilings in Akra-Bazzi recurrences. Fortunately, most of the driving functions we encounter in the study of algorithms behave nicely, and floors and ceilings don't make a difference.

The polynomial-growth condition

If the driving function $f(n)$ in equation (4.22) is well behaved in the following sense, it's okay to drop floors and ceilings.

> A function $f(n)$ defined on all sufficiently large positive reals satisfies the ***polynomial-growth condition*** if there exists a constant $\hat{n} > 0$ such that the following holds: for every constant $\phi \geq 1$, there exists a constant $d > 1$ (depending on ϕ) such that $f(n)/d \leq f(\psi n) \leq df(n)$ for all $1 \leq \psi \leq \phi$ and $n \geq \hat{n}$.

This definition may be one of the hardest in this textbook to get your head around. To a first order, it says that $f(n)$ satisfies the property that $f(\Theta(n)) = \Theta(f(n))$, although the polynomial-growth condition is actually somewhat stronger (see Exercise 4.7-4). The definition also implies that $f(n)$ is asymptotically positive (see Exercise 4.7-3).

Examples of functions that satisfy the polynomial-growth condition include any function of the form $f(n) = \Theta(n^\alpha \lg^\beta n \lg\lg^\gamma n)$, where α, β, and γ are constants. Most of the polynomially bounded functions used in this book satisfy the condition. Exponentials and superexponentials do not (see Exercise 4.7-2, for example), and there also exist polynomially bounded functions that do not.

Floors and ceilings in "nice" recurrences

When the driving function in an Akra-Bazzi recurrence satisfies the polynomial-growth condition, floors and ceilings don't change the asymptotic behavior of the solution. The following theorem, which is presented without proof, formalizes this notion.

Theorem 4.5

Let $T(n)$ be a function defined on the nonnegative reals that satisfies recurrence (4.22), where $f(n)$ satisfies the polynomial-growth condition. Let $T'(n)$ be another function defined on the natural numbers also satisfying recurrence (4.22), except that each $T(n/b_i)$ is replaced either with $T(\lceil n/b_i \rceil)$ or with $T(\lfloor n/b_i \rfloor)$. Then we have $T'(n) = \Theta(T(n))$. ■

Floors and ceilings represent a minor perturbation to the arguments in the recursion. By inequality (3.2) on page 64, they perturb an argument by at most 1. But much larger perturbations are tolerable. As long as the driving function $f(n)$ in recurrence (4.22) satisfies the polynomial-growth condition, it turns out that replacing any term $T(n/b_i)$ with $T(n/b_i + h_i(n))$, where $|h_i(n)| = O(n/\lg^{1+\epsilon} n)$ for some constant $\epsilon > 0$ and sufficiently large n, leaves the asymptotic solution unaffected. Thus, the divide step in a divide-and-conquer algorithm can be moderately coarse without affecting the solution to its running-time recurrence.

The Akra-Bazzi method

The Akra-Bazzi method, not surprisingly, was developed to solve Akra-Bazzi recurrences (4.22), which by dint of Theorem 4.5, applies in the presence of floors and ceilings or even larger perturbations, as just discussed. The method involves first determining the unique real number p such that $\sum_{i=1}^{k} a_i/b_i^p = 1$. Such a p always exists, because when $p \to -\infty$, the sum goes to ∞; it decreases as p increases; and when $p \to \infty$, it goes to 0. The Akra-Bazzi method then gives the solution to the recurrence as

$$T(n) = \Theta\left(n^p\left(1 + \int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) . \tag{4.23}$$

As an example, consider the recurrence

$$T(n) = T(n/5) + T(7n/10) + n . \tag{4.24}$$

We'll see the similar recurrence (9.1) on page 240 when we study an algorithm for selecting the ith smallest element from a set of n numbers. This recurrence has the form of equation (4.22), where $a_1 = a_2 = 1$, $b_1 = 5$, $b_2 = 10/7$, and $f(n) = n$. To solve it, the Akra-Bazzi method says that we should determine the unique p satisfying

$$\left(\frac{1}{5}\right)^p + \left(\frac{7}{10}\right)^p = 1 .$$

Solving for p is kind of messy—it turns out that $p = 0.83978\ldots$—but we can solve the recurrence without actually knowing the exact value for p. Observe that $(1/5)^0 + (7/10)^0 = 2$ and $(1/5)^1 + (7/10)^1 = 9/10$, and thus p lies in the range $0 < p < 1$. That turns out to be sufficient for the Akra-Bazzi method to give us the solution. We'll use the fact from calculus that if $k \neq -1$, then $\int x^k dx = x^{k+1}/(k+1)$, which we'll apply with $k = -p \neq -1$. The Akra-Bazzi solution (4.23) gives us

$$\begin{aligned}
T(n) &= \Theta\left(n^p\left(1+\int_1^n \frac{f(x)}{x^{p+1}}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\int_1^n x^{-p}\,dx\right)\right) \\
&= \Theta\left(n^p\left(1+\left[\frac{x^{1-p}}{1-p}\right]_1^n\right)\right) \\
&= \Theta\left(n^p\left(1+\left(\frac{n^{1-p}}{1-p}-\frac{1}{1-p}\right)\right)\right) \\
&= \Theta\left(n^p\cdot\Theta(n^{1-p})\right) && \text{(because } 1-p \text{ is a positive constant)} \\
&= \Theta(n) && \text{(by Problem 3-5(d))}\ .
\end{aligned}$$

Although the Akra-Bazzi method is more general than the master theorem, it requires calculus and sometimes a bit more reasoning. You also must ensure that your driving function satisfies the polynomial-growth condition if you want to ignore floors and ceilings, although that's rarely a problem. When it applies, the master method is much simpler to use, but only when subproblem sizes are more or less equal. They are both good tools for your algorithmic toolkit.

Exercises

★ ***4.7-1***

Consider an Akra-Bazzi recurrence $T(n)$ on the reals as given in recurrence (4.22), and define $T'(n)$ as

$$T'(n) = cf(n) + \sum_{i=1}^{k} a_i T'(n/b_i)\ ,$$

where $c > 0$ is constant. Prove that whatever the implicit initial conditions for $T(n)$ might be, there exist initial conditions for $T'(n)$ such that $T'(n) = cT(n)$ for all $n > 0$. Conclude that we can drop the asymptotics on a driving function in any Akra-Bazzi recurrence without affecting its asymptotic solution.

4.7-2

Show that $f(n) = n^2$ satisfies the polynomial-growth condition but that $f(n) = 2^n$ does not.

4.7-3

Let $f(n)$ be a function that satisfies the polynomial-growth condition. Prove that $f(n)$ is asymptotically positive, that is, there exists a constant $n_0 \geq 0$ such that $f(n) \geq 0$ for all $n \geq n_0$.

★ ***4.7-4***
Give an example of a function $f(n)$ that does not satisfy the polynomial-growth condition but for which $f(\Theta(n)) = \Theta(f(n))$.

4.7-5
Use the Akra-Bazzi method to solve the following recurrences.

a. $T(n) = T(n/2) + T(n/3) + T(n/6) + n \lg n$.

b. $T(n) = 3T(n/3) + 8T(n/4) + n^2/\lg n$.

c. $T(n) = (2/3)T(n/3) + (1/3)T(2n/3) + \lg n$.

d. $T(n) = (1/3)T(n/3) + 1/n$.

e. $T(n) = 3T(n/3) + 3T(2n/3) + n^2$.

★ ***4.7-6***
Use the Akra-Bazzi method to prove the continuous master theorem.

Problems

4-1 Recurrence examples
Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following algorithmic recurrences. Justify your answers.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(8n/11) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 4T(n/2) + n^2 \lg n$.

e. $T(n) = 8T(n/3) + n^2$.

f. $T(n) = 7T(n/2) + n^2 \lg n$.

g. $T(n) = 2T(n/4) + \sqrt{n}$.

h. $T(n) = T(n-2) + n^2$.

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. Arrays are passed by pointer. Time $= \Theta(1)$.
2. Arrays are passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. Arrays are passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(n)$ if the subarray contains n elements.

Consider the following three algorithms:

a. The recursive binary-search algorithm for finding a number in a sorted array (see Exercise 2.3-6).

b. The MERGE-SORT procedure from Section 2.3.1.

c. The MATRIX-MULTIPLY-RECURSIVE procedure from Section 4.1.

Give nine recurrences $T_{a1}(N,n), T_{a2}(N,n), \ldots, T_{c3}(N,n)$ for the worst-case running times of each of the three algorithms above when arrays and matrices are passed using each of the three parameter-passing strategies above. Solve your recurrences, giving tight asymptotic bounds.

4-3 Solving recurrences with a change of variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. Let's solve the recurrence

$$T(n) = 2T\left(\sqrt{n}\right) + \Theta(\lg n) \tag{4.25}$$

by using the change-of-variables method.

a. Define $m = \lg n$ and $S(m) = T(2^m)$. Rewrite recurrence (4.25) in terms of m and $S(m)$.

b. Solve your recurrence for $S(m)$.

c. Use your solution for $S(m)$ to conclude that $T(n) = \Theta(\lg n \lg\lg n)$.

d. Sketch the recursion tree for recurrence (4.25), and use it to explain intuitively why the solution is $T(n) = \Theta(\lg n \lg\lg n)$.

Solve the following recurrences by changing variables:

e. $T(n) = 2T(\sqrt{n}) + \Theta(1)$.

f. $T(n) = 3T(\sqrt[3]{n}) + \Theta(n)$.

4-4 More recurrence examples

Give asymptotically tight upper and lower bounds for $T(n)$ in each of the following recurrences. Justify your answers.

a. $T(n) = 5T(n/3) + n \lg n$.

b. $T(n) = 3T(n/3) + n/\lg n$.

c. $T(n) = 8T(n/2) + n^3\sqrt{n}$.

d. $T(n) = 2T(n/2 - 2) + n/2$.

e. $T(n) = 2T(n/2) + n/\lg n$.

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n-1) + 1/n$.

h. $T(n) = T(n-1) + \lg n$.

i. $T(n) = T(n-2) + 1/\lg n$.

j. $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$.

4-5 Fibonacci numbers

This problem develops properties of the Fibonacci numbers, which are defined by recurrence (3.31) on page 69. We'll explore the technique of generating functions to solve the Fibonacci recurrence. Define the ***generating function*** (or ***formal power series***) $\mathcal{F}$ as

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots ,\end{aligned}$$

where F_i is the ith Fibonacci number.

a. Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1-z-z^2} \\ &= \frac{z}{(1-\phi z)(1-\widehat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\widehat{\phi} z}\right) ,\end{aligned}$$

where ϕ is the golden ratio, and $\widehat{\phi}$ is its conjugate (see page 69).

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \widehat{\phi}^i)z^i .$$

You may use without proof the generating-function version of equation (A.7) on page 1142, $\sum_{k=0}^{\infty} x^k = 1/(1-x)$. Because this equation involves a generating function, x is a formal variable, not a real-valued variable, so that you don't have to worry about convergence of the summation or about the requirement in equation (A.7) that $|x| < 1$, which doesn't make sense here.

d. Use part (c) to prove that $F_i = \phi^i/\sqrt{5}$ for $i > 0$, rounded to the nearest integer. (*Hint:* Observe that $\left|\widehat{\phi}\right| < 1$.)

e. Prove that $F_{i+2} \geq \phi^i$ for $i \geq 0$.

4-6 Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

a. Show that if at least $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

Now you will design an algorithm to identify which chips are good and which are bad, assuming that more than $n/2$ of the chips are good. First, you will determine how to identify one good chip.

b. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size. That is, show how to use $\lfloor n/2 \rfloor$ pairwise tests to obtain a set with at most $\lceil n/2 \rceil$ chips that still has the property that more than half of the chips are good.

c. Show how to apply the solution to part (b) recursively to identify one good chip. Give and solve the recurrence that describes the number of tests needed to identify one good chip.

You have now determined how to identify one good chip.

d. Show how to identify all the good chips with an additional $\Theta(n)$ pairwise tests.

4-7 Monge arrays

An $m \times n$ array A of real numbers is a ***Monge array*** if for all i, j, k, and l such that $1 \le i < k \le m$ and $1 \le j < l \le n$, we have

$$A[i,j] + A[k,l] \le A[i,l] + A[k,j] .$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots, n-1$, we have

$$A[i,j] + A[i+1, j+1] \le A[i, j+1] + A[i+1, j] .$$

(*Hint:* For the "if" part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it Monge. (*Hint:* Use part (a).)

$$\begin{array}{rrrr}
37 & 23 & 22 & 32 \\
21 & 6 & 7 & 10 \\
53 & 34 & 30 & 31 \\
32 & 13 & 9 & 6 \\
43 & 21 & 15 & 8
\end{array}$$

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \le f(2) \le \cdots \le f(m)$ for any $m \times n$ Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m + n)$ time.

e. Write the recurrence for the running time of the algorithm in part (d). Show that its solution is $O(m + n \log m)$.

Chapter notes

Divide-and-conquer as a technique for designing algorithms dates back at least to 1962 in an article by Karatsuba and Ofman [242], but it might have been used well before then. According to Heideman, Johnson, and Burrus [211], C. F. Gauss devised the first fast Fourier transform algorithm in 1805, and Gauss's formulation breaks the problem into smaller subproblems whose solutions are combined.

Strassen's algorithm [424] caused much excitement when it appeared in 1969. Before then, few imagined the possibility of an algorithm asymptotically faster than the basic MATRIX-MULTIPLY procedure. Shortly thereafter, S. Winograd reduced the number of submatrix additions from 18 to 15 while still using seven submatrix multiplications. This improvement, which Winograd apparently never published (and which is frequently miscited in the literature), may enhance the practicality of the method, but it does not affect its asymptotic performance. Probert [368] described Winograd's algorithm and showed that with seven multiplications, 15 additions is the minimum possible.

Strassen's $\Theta(n^{\lg 7}) = O(n^{2.81})$ bound for matrix multiplication held until 1987, when Coppersmith and Winograd [103] made a significant advance, improving the bound to $O(n^{2.376})$ time with a mathematically sophisticated but wildly impractical algorithm based on tensor products. It took approximately 25 years before the asymptotic upper bound was again improved. In 2012 Vassilevska Williams [445] improved it to $O(n^{2.37287})$, and two years later Le Gall [278] achieved $O(n^{2.37286})$, both of them using mathematically fascinating but impractical algorithms. The best lower bound to date is just the obvious $\Omega(n^2)$ bound (obvious because any algorithm for matrix multiplication must fill in the n^2 elements of the product matrix).

The performance of MATRIX-MULTIPLY-RECURSIVE can be improved in practice by coarsening the leaves of the recursion. It also exhibits better cache behavior than MATRIX-MULTIPLY, although MATRIX-MULTIPLY can be improved by "tiling." Leiserson et al. [293] conducted a performance-engineering study of matrix multiplication in which a parallel and vectorized divide-and-conquer algorithm achieved the highest performance. Strassen's algorithm can be practical for large dense matrices, although large matrices tend to be sparse, and sparse methods can be much faster. When using limited-precision floating-point values, Strassen's algorithm produces larger numerical errors than the $\Theta(n^3)$ algorithms do, although Higham [215] demonstrated that Strassen's algorithm is amply accurate for some applications.

Recurrences were studied as early as 1202 by Leonardo Bonacci [66], also known as Fibonacci, for whom the Fibonacci numbers are named, although Indian mathematicians had discovered Fibonacci numbers centuries before. The French mathematician De Moivre [108] introduced the method of generating functions with which he studied Fibonacci numbers (see Problem 4-5). Knuth [259] and Liu [302] are good resources for learning the method of generating functions.

Aho, Hopcroft, and Ullman [5, 6] offered one of the first general methods for solving recurrences arising from the analysis of divide-and-conquer algorithms. The master method was adapted from Bentley, Haken, and Saxe [52]. The Akra-Bazzi method is due (unsurprisingly) to Akra and Bazzi [13]. Divide-and-conquer recurrences have been studied by many researchers, including Campbell [79], Graham, Knuth, and Patashnik [199], Kuszmaul and Leiserson [274], Leighton [287], Purdom and Brown [371], Roura [389], Verma [447], and Yap [462].

The issue of floors and ceilings in divide-and-conquer recurrences, including a theorem similar to Theorem 4.5, was studied by Leighton [287]. Leighton proposed a version of the polynomial-growth condition. Campbell [79] removed several limitations in Leighton's statement of it and showed that there were polynomially bounded functions that do not satisfy Leighton's condition. Campbell also carefully studied many other technical issues, including the well-definedness of divide-and-conquer recurrences. Kuszmaul and Leiserson [274] provided a proof of Theorem 4.5 that does not involve calculus or other higher math. Both Campbell and Leighton explored the perturbations of arguments beyond simple floors and ceilings.

5 Probabilistic Analysis and Randomized Algorithms

This chapter introduces probabilistic analysis and randomized algorithms. If you are unfamiliar with the basics of probability theory, you should read Sections C.1–C.4 of Appendix C, which review this material. We'll revisit probabilistic analysis and randomized algorithms several times throughout this book.

5.1 The hiring problem

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency sends you one candidate each day. You interview that person and then decide either to hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and also pay a substantial hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

The procedure HIRE-ASSISTANT on the facing page expresses this strategy for hiring in pseudocode. The candidates for the office assistant job are numbered 1 through n and interviewed in that order. The procedure assumes that after interviewing candidate i, you can determine whether candidate i is the best candidate you have seen so far. It starts by creating a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

The cost model for this problem differs from the model described in Chapter 2. We focus not on the running time of HIRE-ASSISTANT, but instead on the fees paid for interviewing and hiring. On the surface, analyzing the cost of this algorithm

```
HIRE-ASSISTANT(n)
best = 0            // candidate 0 is a least-qualified dummy candidate
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

may seem very different from analyzing the running time of, say, merge sort. The analytical techniques used, however, are identical whether we are analyzing cost or running time. In either case, we are counting the number of times certain basic operations are executed.

Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h. Letting m be the number of people hired, the total cost associated with this algorithm is $O(c_i n + c_h m)$. No matter how many people you hire, you always interview n candidates and thus always incur the cost $c_i n$ associated with interviewing. We therefore concentrate on analyzing $c_h m$, the hiring cost. This quantity depends on the order in which you interview candidates.

This scenario serves as a model for a common computational paradigm. Algorithms often need to find the maximum or minimum value in a sequence by examining each element of the sequence and maintaining a current "winner." The hiring problem models how often a procedure updates its notion of which element is currently winning.

Worst-case analysis

In the worst case, you actually hire every candidate that you interview. This situation occurs if the candidates come in strictly increasing order of quality, in which case you hire n times, for a total hiring cost of $O(c_h n)$.

Of course, the candidates do not always come in increasing order of quality. In fact, you have no idea about the order in which they arrive, nor do you have any control over this order. Therefore, it is natural to ask what we expect to happen in a typical or average case.

Probabilistic analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm. Sometimes we use it to analyze other quantities, such as the hiring cost in procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze our algorithm, computing an average-case running time, where we take the average, or expected value, over the distribution of the possible inputs. When reporting such a running time, we refer to it as the ***average-case running time***.

You must be careful in deciding on the distribution of inputs. For some problems, you may reasonably assume something about the set of all possible inputs, and then you can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, you cannot characterize a reasonable input distribution, and in these cases you cannot use probabilistic analysis.

For the hiring problem, we can assume that the applicants come in a random order. What does that mean for this problem? We assume that you can compare any two candidates and decide which one is better qualified, which is to say that there is a total order on the candidates. (See Section B.2 for the definition of a total order.) Thus, you can rank each candidate with a unique number from 1 through n, using $rank(i)$ to denote the rank of applicant i, and adopt the convention that a higher rank corresponds to a better qualified applicant. The ordered list $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ is a permutation of the list $\langle 1, 2, \ldots, n \rangle$. Saying that the applicants come in a random order is equivalent to saying that this list of ranks is equally likely to be any one of the $n!$ permutations of the numbers 1 through n. Alternatively, we say that the ranks form a ***uniform random permutation***, that is, each of the possible $n!$ permutations appears with equal probability.

Section 5.2 contains a probabilistic analysis of the hiring problem.

Randomized algorithms

In order to use probabilistic analysis, you need to know something about the distribution of the inputs. In many cases, you know little about the input distribution. Even if you do know something about the distribution, you might not be able to model this knowledge computationally. Yet, probability and randomness often serve as tools for algorithm design and analysis, by making part of the algorithm behave randomly.

In the hiring problem, it may seem as if the candidates are being presented to you in a random order, but you have no way of knowing whether they really are. Thus, in order to develop a randomized algorithm for the hiring problem, you need greater control over the order in which you'll interview the candidates. We will, therefore, change the model slightly. The employment agency sends you a list of the n candidates in advance. On each day, you choose, randomly, which candidate to interview. Although you know nothing about the candidates (besides their names), we have made a significant change. Instead of accepting the order given to you by the employment agency and hoping that it's random, you have instead gained control of the process and enforced a random order.

More generally, we call an algorithm ***randomized*** if its behavior is determined not only by its input but also by values produced by a ***random-number generator***. We assume that we have at our disposal a random-number generator RANDOM. A call to RANDOM(a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM$(0, 1)$ produces 0 with probability $1/2$, and it produces 1 with probability $1/2$. A call to RANDOM$(3, 7)$ returns any one of $3, 4, 5, 6$, or 7, each with probability $1/5$. Each integer returned by RANDOM is independent of the integers returned on previous calls. You may imagine RANDOM as rolling a $(b - a + 1)$-sided die to obtain its output. (In practice, most programming environments offer a ***pseudorandom-number generator***: a deterministic algorithm returning numbers that "look" statistically random.)

When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator. We distinguish these algorithms from those in which the input is random by referring to the running time of a randomized algorithm as an ***expected running time***. In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm, and we discuss the expected running time when the algorithm itself makes random choices.

Exercises

5.1-1

Show that the assumption that you are always able to determine which candidate is best, in line 4 of procedure HIRE-ASSISTANT, implies that you know a total order on the ranks of the candidates.

★ ***5.1-2***

Describe an implementation of the procedure RANDOM(a, b) that makes calls only to RANDOM$(0, 1)$. What is the expected running time of your procedure, as a function of a and b?

★ ***5.1-3***

You wish to implement a program that outputs 0 with probability $1/2$ and 1 with probability $1/2$. At your disposal is a procedure BIASED-RANDOM that outputs either 0 or 1, but it outputs 1 with some probability p and 0 with probability $1 - p$, where $0 < p < 1$. You do not know what p is. Give an algorithm that uses BIASED-RANDOM as a subroutine, and returns an unbiased answer, returning 0 with probability $1/2$ and 1 with probability $1/2$. What is the expected running time of your algorithm as a function of p?

5.2 Indicator random variables

In order to analyze many algorithms, including the hiring problem, we use indicator random variables. Indicator random variables provide a convenient method for converting between probabilities and expectations. Given a sample space S and an event A, the ***indicator random variable*** $\mathrm{I}\{A\}$ associated with event A is defined as

$$\mathrm{I}\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} , \\ 0 & \text{if } A \text{ does not occur} . \end{cases} \tag{5.1}$$

As a simple example, let us determine the expected number of heads obtained when flipping a fair coin. The sample space for a single coin flip is $S = \{H, T\}$, with $\Pr\{H\} = \Pr\{T\} = 1/2$. We can then define an indicator random variable X_H, associated with the coin coming up heads, which is the event H. This variable counts the number of heads obtained in this flip, and it is 1 if the coin comes up heads and 0 otherwise. We write

$$\begin{aligned} X_H &= \mathrm{I}\{H\} \\ &= \begin{cases} 1 & \text{if } H \text{ occurs} , \\ 0 & \text{if } T \text{ occurs} . \end{cases} \end{aligned}$$

The expected number of heads obtained in one flip of the coin is simply the expected value of our indicator variable X_H:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{H\}] \\ &= 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 . \end{aligned}$$

Thus the expected number of heads obtained by one flip of a fair coin is $1/2$. As the following lemma shows, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs.

Lemma 5.1
Given a sample space S and an event A in the sample space S, let $X_A = \mathrm{I}\{A\}$. Then $\mathrm{E}[X_A] = \Pr\{A\}$.

Proof By the definition of an indicator random variable from equation (5.1) and the definition of expected value, we have

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\left\{\overline{A}\right\} \\ &= \Pr\{A\} , \end{aligned}$$

where $\overline{A}$ denotes $S - A$, the complement of A. ■

Although indicator random variables may seem cumbersome for an application such as counting the expected number of heads on a flip of a single coin, they are useful for analyzing situations that perform repeated random trials. In Appendix C, for example, indicator random variables provide a simple way to determine the expected number of heads in n coin flips. One option is to consider separately the probability of obtaining 0 heads, 1 head, 2 heads, etc. to arrive at the result of equation (C.41) on page 1199. Alternatively, we can employ the simpler method proposed in equation (C.42), which uses indicator random variables implicitly. Making this argument more explicit, let X_i be the indicator random variable associated with the event in which the ith flip comes up heads: $X_i = \mathrm{I}\{\text{the } i\text{th flip results in the event } H\}$. Let X be the random variable denoting the total number of heads in the n coin flips, so that

$$X = \sum_{i=1}^{n} X_i \; .$$

In order to compute the expected number of heads, take the expectation of both sides of the above equation to obtain

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \; . \tag{5.2}$$

By Lemma 5.1, the expectation of each of the random variables is $\mathrm{E}[X_i] = 1/2$ for $i = 1, 2, \ldots, n$. Then we can compute the sum of the expectations: $\sum_{i=1}^{n} \mathrm{E}[X_i] = n/2$. But equation (5.2) calls for the expectation of the sum, not the sum of the expectations. How can we resolve this conundrum? Linearity of expectation, equation (C.24) on page 1192, to the rescue: *the expectation of the sum always equals the sum of the expectations*. Linearity of expectation applies even when there is dependence among the random variables. Combining indicator random variables with linearity of expectation gives us a powerful technique to compute expected values when multiple events occur. We now can compute the expected number of heads:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 \; .
\end{aligned}$$

Thus, compared with the method used in equation (C.41), indicator random variables greatly simplify the calculation. We use indicator random variables throughout this book.

Analysis of the hiring problem using indicator random variables

Returning to the hiring problem, we now wish to compute the expected number of times that you hire a new office assistant. In order to use a probabilistic analysis, let's assume that the candidates arrive in a random order, as discussed in Section 5.1. (We'll see in Section 5.3 how to remove this assumption.) Let X be the random variable whose value equals the number of times you hire a new office assistant. We could then apply the definition of expected value from equation (C.23) on page 1192 to obtain

$$\mathrm{E}[X] = \sum_{x=1}^{n} x \Pr\{X = x\} ,$$

but this calculation would be cumbersome. Instead, let's simplify the calculation by using indicator random variables.

To use indicator random variables, instead of computing $\mathrm{E}[X]$ by defining just one variable denoting the number of times you hire a new office assistant, think of the process of hiring as repeated random trials and define n variables indicating whether each particular candidate is hired. In particular, let X_i be the indicator random variable associated with the event in which the ith candidate is hired. Thus,

$$\begin{aligned} X_i &= \mathrm{I}\{\text{candidate } i \text{ is hired}\} \\ &= \begin{cases} 1 & \text{if candidate } i \text{ is hired} , \\ 0 & \text{if candidate } i \text{ is not hired} , \end{cases} \end{aligned}$$

and

$$X = X_1 + X_2 + \cdots + X_n . \tag{5.3}$$

Lemma 5.1 gives

$$\mathrm{E}[X_i] = \Pr\{\text{candidate } i \text{ is hired}\} ,$$

and we must therefore compute the probability that lines 5–6 of HIRE-ASSISTANT are executed.

Candidate i is hired, in line 6, exactly when candidate i is better than each of candidates 1 through $i-1$. Because we have assumed that the candidates arrive in a random order, the first i candidates have appeared in a random order. Any one of these first i candidates is equally likely to be the best qualified so far. Candidate i has a probability of $1/i$ of being better qualified than candidates 1 through $i-1$ and thus a probability of $1/i$ of being hired. By Lemma 5.1, we conclude that

$$\mathrm{E}[X_i] = 1/i \ . \tag{5.4}$$

Now we can compute $\mathrm{E}[X]$:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.3))} && (5.5)\\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by equation (C.24), linearity of expectation)}\\
&= \sum_{i=1}^{n} \frac{1}{i} && \text{(by equation (5.4))}\\
&= \ln n + O(1) && \text{(by equation (A.9), the harmonic series) .} && (5.6)
\end{aligned}$$

Even though you interview n people, you actually hire only approximately $\ln n$ of them, on average. We summarize this result in the following lemma.

Lemma 5.2
Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has an average-case total hiring cost of $O(c_h \ln n)$.

Proof The bound follows immediately from our definition of the hiring cost and equation (5.6), which shows that the expected number of hires is approximately $\ln n$. ■

The average-case hiring cost is a significant improvement over the worst-case hiring cost of $O(c_h n)$.

Exercises

5.2-1
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

5.2-2
In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly twice?

5.2-3
Use indicator random variables to compute the expected value of the sum of n dice.

5.2-4

This exercise asks you to (partly) verify that linearity of expectation holds even if the random variables are not independent. Consider two 6-sided dice that are rolled independently. What is the expected value of the sum? Now consider the case where the first die is rolled normally and then the second die is set equal to the value shown on the first die. What is the expected value of the sum? Now consider the case where the first die is rolled normally and the second die is set equal to 7 minus the value of the first die. What is the expected value of the sum?

5.2-5

Use indicator random variables to solve the following problem, which is known as the ***hat-check problem***. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

5.2-6

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. (See Problem 2-4 on page 47 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

5.3 Randomized algorithms

In the previous section, we showed how knowing a distribution on the inputs can help us to analyze the average-case behavior of an algorithm. What if you do not know the distribution? Then you cannot perform an average-case analysis. As mentioned in Section 5.1, however, you might be able to use a randomized algorithm.

For a problem such as the hiring problem, in which it is helpful to assume that all permutations of the input are equally likely, a probabilistic analysis can guide us when developing a randomized algorithm. Instead of *assuming* a distribution of inputs, we *impose* a distribution. In particular, before running the algorithm, let's randomly permute the candidates in order to enforce the property that every permutation is equally likely. Although we have modified the algorithm, we still expect to hire a new office assistant approximately $\ln n$ times. But now we expect this to be the case for *any* input, rather than for inputs drawn from a particular distribution.

Let us further explore the distinction between probabilistic analysis and randomized algorithms. In Section 5.2, we claimed that, assuming that the candidates arrive in a random order, the expected number of times you hire a new office assistant is about $\ln n$. This algorithm is deterministic: for any particular input, the number of times a new office assistant is hired is always the same. Furthermore, the number of times you hire a new office assistant differs for different inputs, and it depends on the ranks of the various candidates. Since this number depends only on the ranks of the candidates, to represent a particular input, we can just list, in order, the ranks $\langle rank(1), rank(2), \ldots, rank(n) \rangle$ of the candidates. Given the rank list $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$, a new office assistant is always hired 10 times, since each successive candidate is better than the previous one, and lines 5–6 of HIRE-ASSISTANT are executed in each iteration. Given the list of ranks $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$, a new office assistant is hired only once, in the first iteration. Given a list of ranks $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$, a new office assistant is hired three times, upon interviewing the candidates with ranks 5, 8, and 10. Recalling that the cost of our algorithm depends on how many times you hire a new office assistant, we see that there are expensive inputs such as A_1, inexpensive inputs such as A_2, and moderately expensive inputs such as A_3.

Consider, on the other hand, the randomized algorithm that first permutes the list of candidates and then determines the best candidate. In this case, we randomize in the algorithm, not in the input distribution. Given a particular input, say A_3 above, we cannot say how many times the maximum is updated, because this quantity differs with each run of the algorithm. The first time you run the algorithm on A_3, it might produce the permutation A_1 and perform 10 updates. But the second time you run the algorithm, it might produce the permutation A_2 and perform only one update. The third time you run the algorithm, it might perform some other number of updates. Each time you run the algorithm, its execution depends on the random choices made and is likely to differ from the previous execution of the algorithm. For this algorithm and many other randomized algorithms, *no particular input elicits its worst-case behavior*. Even your worst enemy cannot produce a bad input array, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation.

For the hiring problem, the only change needed in the code is to randomly permute the array, as done in the RANDOMIZED-HIRE-ASSISTANT procedure. This simple change creates a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

```
RANDOMIZED-HIRE-ASSISTANT($n$)
1  randomly permute the list of candidates
2  HIRE-ASSISTANT($n$)
```

Lemma 5.3
The expected hiring cost of the procedure RANDOMIZED-HIRE-ASSISTANT is $O(c_h \ln n)$.

Proof Permuting the input array achieves a situation identical to that of the probabilistic analysis of HIRE-ASSISTANT in Section 5.2. ■

By carefully comparing Lemmas 5.2 and 5.3, you can see the difference between probabilistic analysis and randomized algorithms. Lemma 5.2 makes an assumption about the input. Lemma 5.3 makes no such assumption, although randomizing the input takes some additional time. To remain consistent with our terminology, we couched Lemma 5.2 in terms of the average-case hiring cost and Lemma 5.3 in terms of the expected hiring cost. In the remainder of this section, we discuss some issues involved in randomly permuting inputs.

Randomly permuting arrays

Many randomized algorithms randomize the input by permuting a given input array. We'll see elsewhere in this book other ways to randomize an algorithm, but now, let's see how we can randomly permute an array of n elements. The goal is to produce a ***uniform random permutation***, that is, a permutation that is as likely as any other permutation. Since there are $n!$ possible permutations, we want the probability that any particular permutation is produced to be $1/n!$.

You might think that to prove that a permutation is a uniform random permutation, it suffices to show that, for each element $A[i]$, the probability that the element winds up in position j is $1/n$. Exercise 5.3-4 shows that this weaker condition is, in fact, insufficient.

Our method to generate a random permutation permutes the array ***in place***: at most a constant number of elements of the input array are ever stored outside the array. The procedure RANDOMLY-PERMUTE permutes an array $A[1:n]$ in place in $\Theta(n)$ time. In its ith iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$. After the ith iteration, $A[i]$ is never altered.

```
RANDOMLY-PERMUTE($A, n$)
1  for $i = 1$ to $n$
2      swap $A[i]$ with $A[$RANDOM$(i, n)]$
```

We use a loop invariant to show that procedure RANDOMLY-PERMUTE produces a uniform random permutation. A ***k-permutation*** on a set of n elements is a sequence containing k of the n elements, with no repetitions. (See page 1180 in Appendix C.) There are $n!/(n-k)!$ such possible k-permutations.

Lemma 5.4
Procedure RANDOMLY-PERMUTE computes a uniform random permutation.

Proof We use the following loop invariant:

> Just prior to the ith iteration of the **for** loop of lines 1–2, for each possible $(i-1)$-permutation of the n elements, the subarray $A[1:i-1]$ contains this $(i-1)$-permutation with probability $(n-i+1)!/n!$.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Consider the situation just before the first loop iteration, so that $i = 1$. The loop invariant says that for each possible 0-permutation, the subarray $A[1:0]$ contains this 0-permutation with probability $(n-i+1)!/n! = n!/n! = 1$. The subarray $A[1:0]$ is an empty subarray, and a 0-permutation has no elements. Thus, $A[1:0]$ contains any 0-permutation with probability 1, and the loop invariant holds prior to the first iteration.

Maintenance: By the loop invariant, we assume that just before the ith iteration, each possible $(i-1)$-permutation appears in the subarray $A[1:i-1]$ with probability $(n-i+1)!/n!$. We shall show that after the ith iteration, each possible i-permutation appears in the subarray $A[1:i]$ with probability $(n-i)!/n!$. Incrementing i for the next iteration then maintains the loop invariant.

Let us examine the ith iteration. Consider a particular i-permutation, and denote the elements in it by $\langle x_1, x_2, \ldots, x_i \rangle$. This permutation consists of an $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1} \rangle$ followed by the value x_i that the algorithm places in $A[i]$. Let E_1 denote the event in which the first $i-1$ iterations have created the particular $(i-1)$-permutation $\langle x_1, \ldots, x_{i-1} \rangle$ in $A[1:i-1]$. By the loop invariant, $\Pr\{E_1\} = (n-i+1)!/n!$. Let E_2 be the event that the ith iteration puts x_i in position $A[i]$. The i-permutation $\langle x_1, \ldots, x_i \rangle$ appears in $A[1:i]$ precisely when both E_1 and E_2 occur, and so we wish to compute $\Pr\{E_2 \cap E_1\}$. Using equation (C.16) on page 1187, we have

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} .$$

The probability $\Pr\{E_2 \mid E_1\}$ equals $1/(n-i+1)$ because in line 2 the algorithm chooses x_i randomly from the $n-i+1$ values in positions $A[i:n]$. Thus, we have

$$\begin{aligned}\Pr\{E_2 \cap E_1\} &= \Pr\{E_2 \mid E_1\} \Pr\{E_1\} \\ &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!} .\end{aligned}$$

Termination: The loop terminates, since it is a **for** loop iterating n times. At termination, $i = n + 1$, and we have that the subarray $A[1:n]$ is a given n-permutation with probability $(n - (n+1) + 1)!/n! = 0!/n! = 1/n!$.

Thus, RANDOMLY-PERMUTE produces a uniform random permutation. ■

A randomized algorithm is often the simplest and most efficient way to solve a problem.

Exercises

5.3-1

Professor Marceau objects to the loop invariant used in the proof of Lemma 5.4. He questions whether it holds prior to the first iteration. He reasons that we could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMLY-PERMUTE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.4 for your procedure.

5.3-2

Professor Kelp decides to write a procedure that produces at random any permutation except the ***identity permutation***, in which every element ends up where it started. He proposes the procedure PERMUTE-WITHOUT-IDENTITY. Does this procedure do what Professor Kelp intends?

```
PERMUTE-WITHOUT-IDENTITY($A, n$)
1  for $i = 1$ to $n - 1$
2      swap $A[i]$ with $A[$RANDOM$(i + 1, n)]$
```

5.3-3

Consider the PERMUTE-WITH-ALL procedure on the facing page, which instead of swapping element $A[i]$ with a random element from the subarray $A[i:n]$, swaps it with a random element from anywhere in the array. Does PERMUTE-WITH-ALL produce a uniform random permutation? Why or why not?

```
PERMUTE-WITH-ALL($A, n$)
for $i = 1$ to $n$
    swap $A[i]$ with $A[$RANDOM$(1, n)]$
```

5.3-4

Professor Knievel suggests the procedure PERMUTE-BY-CYCLE to generate a uniform random permutation. Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B. Then show that Professor Knievel is mistaken by showing that the resulting permutation is not uniformly random.

```
PERMUTE-BY-CYCLE($A, n$)
let $B[1:n]$ be a new array
$offset$ = RANDOM$(1, n)$
for $i = 1$ to $n$
    $dest = i + offset$
    if $dest > n$
        $dest = dest - n$
    $B[dest] = A[i]$
return $B$
```

5.3-5

Professor Gallup wants to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \le m \le n$, such that each m-subset is equally likely to be created. One way is to set $A[i] = i$, for $i = 1, 2, 3, \ldots, n$, call RANDOMLY-PERMUTE(A), and then take just the first m array elements. This method makes n calls to the RANDOM procedure. In Professor Gallup's application, n is much larger than m, and so the professor wants to create a random sample with fewer calls to RANDOM.

```
RANDOM-SAMPLE($m, n$)
$S = \emptyset$
for $k = n - m + 1$ to $n$          // iterates $m$ times
    $i$ = RANDOM$(1, k)$
    if $i \in S$
        $S = S \cup \{k\}$
    else $S = S \cup \{i\}$
return $S$
```

Show that the procedure RANDOM-SAMPLE on the previous page returns a random m-subset S of $\{1, 2, 3, \ldots, n\}$, in which each m-subset is equally likely, while making only m calls to RANDOM.

★ 5.4 Probabilistic analysis and further uses of indicator random variables

This advanced section further illustrates probabilistic analysis by way of four examples. The first determines the probability that in a room of k people, two of them share the same birthday. The second example examines what happens when randomly tossing balls into bins. The third investigates "streaks" of consecutive heads when flipping coins. The final example analyzes a variant of the hiring problem in which you have to make decisions without actually interviewing all the candidates.

5.4.1 The birthday paradox

Our first example is the ***birthday paradox***. How many people must there be in a room before there is a 50% chance that two of them were born on the same day of the year? The answer is surprisingly few. The paradox is that it is in fact far fewer than the number of days in a year, or even half the number of days in a year, as we shall see.

To answer this question, we index the people in the room with the integers $1, 2, \ldots, k$, where k is the number of people in the room. We ignore the issue of leap years and assume that all years have $n = 365$ days. For $i = 1, 2, \ldots, k$, let b_i be the day of the year on which person i's birthday falls, where $1 \le b_i \le n$. We also assume that birthdays are uniformly distributed across the n days of the year, so that $\Pr\{b_i = r\} = 1/n$ for $i = 1, 2, \ldots, k$ and $r = 1, 2, \ldots, n$.

The probability that two given people, say i and j, have matching birthdays depends on whether the random selection of birthdays is independent. We assume from now on that birthdays are independent, so that the probability that i's birthday and j's birthday both fall on day r is

$$\begin{aligned}\Pr\{b_i = r \text{ and } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= \frac{1}{n^2} .\end{aligned}$$

Thus, the probability that they both fall on the same day is

$$\Pr\{b_i = b_j\} = \sum_{r=1}^{n} \Pr\{b_i = r \text{ and } b_j = r\}$$

$$\begin{aligned} &= \sum_{r=1}^{n} \frac{1}{n^2} \\ &= \frac{1}{n} \ . \end{aligned} \tag{5.7}$$

More intuitively, once b_i is chosen, the probability that b_j is chosen to be the same day is $1/n$. As long as the birthdays are independent, the probability that i and j have the same birthday is the same as the probability that the birthday of one of them falls on a given day.

We can analyze the probability of at least 2 out of k people having matching birthdays by looking at the complementary event. The probability that at least two of the birthdays match is 1 minus the probability that all the birthdays are different. The event B_k that k people have distinct birthdays is

$$B_k = \bigcap_{i=1}^{k} A_i \ ,$$

where A_i is the event that person i's birthday is different from person j's for all $j < i$. Since we can write $B_k = A_k \cap B_{k-1}$, we obtain from equation (C.18) on page 1189 the recurrence

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \ , \tag{5.8}$$

where we take $\Pr\{B_1\} = \Pr\{A_1\} = 1$ as an initial condition. In other words, the probability that $b_1, b_2, \ldots, b_k$ are distinct birthdays equals the probability that $b_1, b_2, \ldots, b_{k-1}$ are distinct birthdays multiplied by the probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k-1$, given that $b_1, b_2, \ldots, b_{k-1}$ are distinct.

If $b_1, b_2, \ldots, b_{k-1}$ are distinct, the conditional probability that $b_k \neq b_i$ for $i = 1, 2, \ldots, k-1$ is $\Pr\{A_k \mid B_{k-1}\} = (n - k + 1)/n$, since out of the n days, $n - (k-1)$ days are not taken. We iteratively apply the recurrence (5.8) to obtain

$$\begin{aligned} \Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \\ &= \Pr\{B_{k-2}\} \Pr\{A_{k-1} \mid B_{k-2}\} \Pr\{A_k \mid B_{k-1}\} \\ &\ \ \vdots \\ &= \Pr\{B_1\} \Pr\{A_2 \mid B_1\} \Pr\{A_3 \mid B_2\} \cdots \Pr\{A_k \mid B_{k-1}\} \\ &= 1 \cdot \left(\frac{n-1}{n}\right) \left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\ &= 1 \cdot \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right) \ . \end{aligned}$$

Inequality (3.14) on page 66, $1 + x \leq e^x$, gives us

$$\begin{aligned}
\Pr\{B_k\} &\le e^{-1/n}e^{-2/n}\cdots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\le \frac{1}{2}
\end{aligned}$$

when $-k(k-1)/2n \le \ln(1/2)$. The probability that all k birthdays are distinct is at most $1/2$ when $k(k-1) \ge 2n\ln 2$ or, solving the quadratic equation, when $k \ge (1+\sqrt{1+(8\ln 2)n})/2$. For $n = 365$, we must have $k \ge 23$. Thus, if at least 23 people are in a room, the probability is at least $1/2$ that at least two people have the same birthday. Since a year on Mars is 669 Martian days long, it takes 31 Martians to get the same effect.

An analysis using indicator random variables

Indicator random variables afford a simpler but approximate analysis of the birthday paradox. For each pair (i, j) of the k people in the room, define the indicator random variable X_{ij}, for $1 \le i < j \le k$, by

$$\begin{aligned}
X_{ij} &= \mathrm{I}\{\text{person } i \text{ and person } j \text{ have the same birthday}\} \\
&= \begin{cases} 1 & \text{if person } i \text{ and person } j \text{ have the same birthday}, \\ 0 & \text{otherwise}. \end{cases}
\end{aligned}$$

By equation (5.7), the probability that two people have matching birthdays is $1/n$, and thus by Lemma 5.1 on page 130, we have

$$\begin{aligned}
\mathrm{E}[X_{ij}] &= \Pr\{\text{person } i \text{ and person } j \text{ have the same birthday}\} \\
&= 1/n .
\end{aligned}$$

Letting X be the random variable that counts the number of pairs of individuals having the same birthday, we have

$$X = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} X_{ij} .$$

Taking expectations of both sides and applying linearity of expectation, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} X_{ij}\right] \\
&= \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \mathrm{E}[X_{ij}]
\end{aligned}$$

$$= \binom{k}{2}\frac{1}{n}$$
$$= \frac{k(k-1)}{2n} .$$

When $k(k-1) \geq 2n$, therefore, the expected number of pairs of people with the same birthday is at least 1. Thus, if we have at least $\sqrt{2n}+1$ individuals in a room, we can expect at least two to have the same birthday. For $n = 365$, if $k = 28$, the expected number of pairs with the same birthday is $(28 \cdot 27)/(2 \cdot 365) \approx 1.0356$. Thus, with at least 28 people, we expect to find at least one matching pair of birthdays. On Mars, with 669 days per year, we need at least 38 Martians.

The first analysis, which used only probabilities, determined the number of people required for the probability to exceed $1/2$ that a matching pair of birthdays exists, and the second analysis, which used indicator random variables, determined the number such that the expected number of matching birthdays is 1. Although the exact numbers of people differ for the two situations, they are the same asymptotically: $\Theta(\sqrt{n})$.

5.4.2 Balls and bins

Consider a process in which you randomly toss identical balls into b bins, numbered $1, 2, \ldots, b$. The tosses are independent, and on each toss the ball is equally likely to end up in any bin. The probability that a tossed ball lands in any given bin is $1/b$. If we view the ball-tossing process as a sequence of Bernoulli trials (see Appendix C.4), where success means that the ball falls in the given bin, then each trial has a probability $1/b$ of success. This model is particularly useful for analyzing hashing (see Chapter 11), and we can answer a variety of interesting questions about the ball-tossing process. (Problem C-2 asks additional questions about balls and bins.)

- *How many balls fall in a given bin?* The number of balls that fall in a given bin follows the binomial distribution $b(k; n, 1/b)$. If you toss n balls, equation (C.41) on page 1199 tells us that the expected number of balls that fall in the given bin is n/b.

- *How many balls must you toss, on the average, until a given bin contains a ball?* The number of tosses until the given bin receives a ball follows the geometric distribution with probability $1/b$ and, by equation (C.36) on page 1197, the expected number of tosses until success is $1/(1/b) = b$.

- *How many balls must you toss until every bin contains at least one ball?* Let us call a toss in which a ball falls into an empty bin a "hit." We want to know the expected number n of tosses required to get b hits.

Using the hits, we can partition the n tosses into stages. The ith stage consists of the tosses after the $(i-1)$st hit up to and including the ith hit. The first stage consists of the first toss, since you are guaranteed to have a hit when all bins are empty. For each toss during the ith stage, $i-1$ bins contain balls and $b-i+1$ bins are empty. Thus, for each toss in the ith stage, the probability of obtaining a hit is $(b-i+1)/b$.

Let n_i denote the number of tosses in the ith stage. The number of tosses required to get b hits is $n = \sum_{i=1}^{b} n_i$. Each random variable n_i has a geometric distribution with probability of success $(b-i+1)/b$ and thus, by equation (C.36), we have

$$\mathrm{E}[n_i] = \frac{b}{b-i+1} .$$

By linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} \mathrm{E}[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b \sum_{i=1}^{b} \frac{1}{i} && \text{(by equation (A.14) on page 1144)} \\
&= b(\ln b + O(1)) && \text{(by equation (A.9) on page 1142)} .
\end{aligned}$$

It therefore takes approximately $b \ln b$ tosses before we can expect that every bin has a ball. This problem is also known as the ***coupon collector's problem***, which says that if you are trying to collect each of b different coupons, then you should expect to acquire approximately $b \ln b$ randomly obtained coupons in order to succeed.

5.4.3 Streaks

Suppose that you flip a fair coin n times. What is the longest streak of consecutive heads that you expect to see? We'll prove upper and lower bounds separately to show that the answer is $\Theta(\lg n)$.

We first prove that the expected length of the longest streak of heads is $O(\lg n)$. The probability that each coin flip is a head is $1/2$. Let A_{ik} be the event that a streak of heads of length at least k begins with the ith coin flip or, more precisely, the event that the k consecutive coin flips $i, i+1, \ldots, i+k-1$ yield only heads, where $1 \le k \le n$ and $1 \le i \le n-k+1$. Since coin flips are mutually independent, for any given event A_{ik}, the probability that all k flips are heads is

$$\Pr\{A_{ik}\} = \frac{1}{2^k} . \tag{5.9}$$

For $k = 2\lceil \lg n \rceil$,

$$\begin{aligned} \Pr\{A_{i,2\lceil \lg n \rceil}\} &= \frac{1}{2^{2\lceil \lg n \rceil}} \\ &\le \frac{1}{2^{2 \lg n}} \\ &= \frac{1}{n^2} , \end{aligned}$$

and thus the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins in position i is quite small. There are at most $n - 2\lceil \lg n \rceil + 1$ positions where such a streak can begin. The probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is therefore

$$\begin{aligned} \Pr\left\{ \bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil} \right\} & \\ &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \Pr\{A_{i,2\lceil \lg n \rceil}\} \quad \text{(by Boole's inequality (C.21) on page 1190)} \\ &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \frac{1}{n^2} \\ &< \sum_{i=1}^{n} \frac{1}{n^2} \\ &= \frac{1}{n} . \end{aligned} \tag{5.10}$$

We can use inequality (5.10) to bound the length of the longest streak. For $j = 0, 1, 2, \ldots, n$, let L_j be the event that the longest streak of heads has length exactly j, and let L be the length of the longest streak. By the definition of expected value, we have

$$\mathrm{E}[L] = \sum_{j=0}^{n} j \Pr\{L_j\} . \tag{5.11}$$

We could try to evaluate this sum using upper bounds on each $\Pr\{L_j\}$ similar to those computed in inequality (5.10). Unfortunately, this method yields weak bounds. We can use some intuition gained by the above analysis to obtain a good bound, however. For no individual term in the summation in equation (5.11) are both the factors j and $\Pr\{L_j\}$ large. Why? When $j \geq 2\lceil \lg n \rceil$, then $\Pr\{L_j\}$ is very small, and when $j < 2\lceil \lg n \rceil$, then j is fairly small. More precisely, since the events L_j for $j = 0, 1, \ldots, n$ are disjoint, the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\}$. Inequality (5.10) tells us that the probability that a streak of heads of length at least $2\lceil \lg n \rceil$ begins anywhere is less than $1/n$, which means that $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < 1/n$. Also, noting that $\sum_{j=0}^{n} \Pr\{L_j\} = 1$, we have that $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} \leq 1$. Thus, we obtain

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&< \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot \frac{1}{n} \\
&= O(\lg n) .
\end{aligned}$$

The probability that a streak of heads exceeds $r\lceil \lg n \rceil$ flips diminishes quickly with r. Let's get a rough bound on the probability that a streak of at least $r\lceil \lg n \rceil$ heads occurs, for $r \geq 1$. The probability that a streak of at least $r\lceil \lg n \rceil$ heads starts in position i is

$$\begin{aligned}
\Pr\{A_{i,r\lceil \lg n \rceil}\} &= \frac{1}{2^{r\lceil \lg n \rceil}} \\
&\leq \frac{1}{n^r} .
\end{aligned}$$

A streak of at least $r\lceil \lg n \rceil$ heads cannot start in the last $n - r\lceil \lg n \rceil + 1$ flips, but let's overestimate the probability of such a streak by allowing it to start anywhere within the n coin flips. Then the probability that a streak of at least $r\lceil \lg n \rceil$ heads occurs is at most

$$\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n} A_{i,r\lceil \lg n\rceil}\right\} &\le \sum_{i=1}^{n} \Pr\{A_{i,r\lceil \lg n\rceil}\} && \text{(by Boole's inequality (C.21))} \\
&\le \sum_{i=1}^{n} \frac{1}{n^r} \\
&= \frac{1}{n^{r-1}} \,.
\end{aligned}$$

Equivalently, the probability is at least $1-1/n^{r-1}$ that the longest streak has length less than $r\lceil \lg n\rceil$.

As an example, during $n = 1000$ coin flips, the probability of encountering a streak of at least $2\lceil \lg n\rceil = 20$ heads is at most $1/n = 1/1000$. The chance of a streak of at least $3\lceil \lg n\rceil = 30$ heads is at most $1/n^2 = 1/1{,}000{,}000$.

Let's now prove a complementary lower bound: the expected length of the longest streak of heads in n coin flips is $\Omega(\lg n)$. To prove this bound, we look for streaks of length s by partitioning the n flips into approximately n/s groups of s flips each. If we choose $s = \lfloor (\lg n)/2\rfloor$, we'll see that it is likely that at least one of these groups comes up all heads, which means that it's likely that the longest streak has length at least $s = \Omega(\lg n)$. We'll then show that the longest streak has expected length $\Omega(\lg n)$.

Let's partition the n coin flips into at least $\lfloor n/\lfloor (\lg n)/2\rfloor\rfloor$ groups of $\lfloor (\lg n)/2\rfloor$ consecutive flips and bound the probability that no group comes up all heads. By equation (5.9), the probability that the group starting in position i comes up all heads is

$$\begin{aligned}
\Pr\{A_{i,\lfloor (\lg n)/2\rfloor}\} &= \frac{1}{2^{\lfloor (\lg n)/2\rfloor}} \\
&\ge \frac{1}{\sqrt{n}} \,.
\end{aligned}$$

The probability that a streak of heads of length at least $\lfloor (\lg n)/2\rfloor$ does not begin in position i is therefore at most $1 - 1/\sqrt{n}$. Since the $\lfloor n/\lfloor (\lg n)/2\rfloor\rfloor$ groups are formed from mutually exclusive, independent coin flips, the probability that every one of these groups *fails* to be a streak of length $\lfloor (\lg n)/2\rfloor$ is at most

$$\begin{aligned}
\left(1 - 1/\sqrt{n}\right)^{\lfloor n/\lfloor (\lg n)/2\rfloor\rfloor} &\le \left(1 - 1/\sqrt{n}\right)^{n/\lfloor (\lg n)/2\rfloor - 1} \\
&\le \left(1 - 1/\sqrt{n}\right)^{2n/\lg n - 1} \\
&\le e^{-(2n/\lg n - 1)/\sqrt{n}} \\
&= O(e^{-\ln n}) \\
&= O(1/n) \,. \qquad (5.12)
\end{aligned}$$

For this argument, we used inequality (3.14), $1 + x \le e^x$, on page 66 and the fact, which you may verify, that $(2n/\lg n - 1)/\sqrt{n} \ge \ln n$ for sufficiently large n.

We want to bound the probability that the longest streak equals or exceeds $\lfloor (\lg n)/2 \rfloor$. To do so, let L be the event that the longest streak of heads equals or exceeds $s = \lfloor (\lg n)/2 \rfloor$. Let $\overline{L}$ be the complementary event, that the longest streak of heads is strictly less than s, so that $\Pr\{L\} + \Pr\{\overline{L}\} = 1$. Let F be the event that every group of s flips fails to be a streak of s heads. By inequality (5.12), we have $\Pr\{F\} = O(1/n)$. If the longest streak of heads is less than s, then certainly every group of s flips fails to be a streak of s heads, which means that event $\overline{L}$ implies event F. Of course, event F could occur even if event $\overline{L}$ does not (for example, if a streak of s or more heads crosses over the boundary between two groups), and so we have $\Pr\{\overline{L}\} \le \Pr\{F\} = O(1/n)$. Since $\Pr\{L\} + \Pr\{\overline{L}\} = 1$, we have that

$$\begin{aligned}
\Pr\{L\} &= 1 - \Pr\{\overline{L}\} \\
&\ge 1 - \Pr\{F\} \\
&= 1 - O(1/n) .
\end{aligned}$$

That is, the probability that the longest streak equals or exceeds $\lfloor (\lg n)/2 \rfloor$ is

$$\sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} \Pr\{L_j\} \ge 1 - O(1/n) . \tag{5.13}$$

We can now calculate a lower bound on the expected length of the longest streak, beginning with equation (5.11) and proceeding in a manner similar to our analysis of the upper bound:

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor - 1} j \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} j \Pr\{L_j\} \\
&\ge \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor - 1} 0 \cdot \Pr\{L_j\} + \sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} \lfloor (\lg n)/2 \rfloor \Pr\{L_j\} \\
&= 0 \cdot \sum_{j=0}^{\lfloor (\lg n)/2 \rfloor - 1} \Pr\{L_j\} + \lfloor (\lg n)/2 \rfloor \sum_{j=\lfloor (\lg n)/2 \rfloor}^{n} \Pr\{L_j\} \\
&\ge 0 + \lfloor (\lg n)/2 \rfloor (1 - O(1/n)) \qquad \text{(by inequality (5.13))} \\
&= \Omega(\lg n) .
\end{aligned}$$

As with the birthday paradox, we can obtain a simpler, but approximate, analysis using indicator random variables. Instead of determining the expected length of the longest streak, we'll find the expected number of streaks with at least a given length. Let $X_{ik} = \mathrm{I}\{A_{ik}\}$ be the indicator random variable associated with a streak of heads of length at least k beginning with the ith coin flip. To count the total number of such streaks, define

$$X_k = \sum_{i=1}^{n-k+1} X_{ik} .$$

Taking expectations and using linearity of expectation, we have

$$\begin{aligned} \mathrm{E}[X_k] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\ &= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\ &= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\ &= \sum_{i=1}^{n-k+1} \frac{1}{2^k} \\ &= \frac{n-k+1}{2^k} . \end{aligned}$$

By plugging in various values for k, we can calculate the expected number of streaks of length at least k. If this expected number is large (much greater than 1), then we expect many streaks of length k to occur, and the probability that one occurs is high. If this expected number is small (much less than 1), then we expect to see few streaks of length k, and the probability that one occurs is low. If $k = c \lg n$, for some positive constant c, we obtain

$$\begin{aligned} \mathrm{E}[X_{c \lg n}] &= \frac{n - c \lg n + 1}{2^{c \lg n}} \\ &= \frac{n - c \lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}) . \end{aligned}$$

If c is large, the expected number of streaks of length $c \lg n$ is small, and we conclude that they are unlikely to occur. On the other hand, if $c = 1/2$, then we obtain $\text{E}[X_{(1/2)\lg n}] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, and we expect there to be numerous streaks of length $(1/2)\lg n$. Therefore, one streak of such a length is likely to occur. We can conclude that the expected length of the longest streak is $\Theta(\lg n)$.

5.4.4 The online hiring problem

As a final example, let's consider a variant of the hiring problem. Suppose now that you do not wish to interview all the candidates in order to find the best one. You also want to avoid hiring and firing as you find better and better applicants. Instead, you are willing to settle for a candidate who is close to the best, in exchange for hiring exactly once. You must obey one company requirement: after each interview you must either immediately offer the position to the applicant or immediately reject the applicant. What is the trade-off between minimizing the amount of interviewing and maximizing the quality of the candidate hired?

We can model this problem in the following way. After meeting an applicant, you are able to give each one a score. Let $score(i)$ denote the score you give to the ith applicant, and assume that no two applicants receive the same score. After you have seen j applicants, you know which of the j has the highest score, but you do not know whether any of the remaining $n - j$ applicants will receive a higher score. You decide to adopt the strategy of selecting a positive integer $k < n$, interviewing and then rejecting the first k applicants, and hiring the first applicant thereafter who has a higher score than all preceding applicants. If it turns out that the best-qualified applicant was among the first k interviewed, then you hire the nth applicant—the last one interviewed. We formalize this strategy in the procedure ONLINE-MAXIMUM(k, n), which returns the index of the candidate you wish to hire.

```
ONLINE-MAXIMUM(k, n)
1  best-score = −∞
2  for i = 1 to k
3      if score(i) > best-score
4          best-score = score(i)
5  for i = k + 1 to n
6      if score(i) > best-score
7          return i
8  return n
```

If we determine, for each possible value of k, the probability that you hire the most qualified applicant, then you can choose the best possible k and implement the strategy with that value. For the moment, assume that k is fixed. Let

$M(j) = \max\{score(i) : 1 \le i \le j\}$ denote the maximum score among applicants 1 through j. Let S be the event that you succeed in choosing the best-qualified applicant, and let S_i be the event that you succeed when the best-qualified applicant is the ith one interviewed. Since the various S_i are disjoint, we have that $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Noting that you never succeed when the best-qualified applicant is one of the first k, we have that $\Pr\{S_i\} = 0$ for $i = 1, 2, \ldots, k$. Thus, we obtain

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} . \tag{5.14}$$

We now compute $\Pr\{S_i\}$. In order to succeed when the best-qualified applicant is the ith one, two things must happen. First, the best-qualified applicant must be in position i, an event which we denote by B_i. Second, the algorithm must not select any of the applicants in positions $k+1$ through $i-1$, which happens only if, for each j such that $k+1 \le j \le i-1$, line 6 finds that $score(j) < best\text{-}score$. (Because scores are unique, we can ignore the possibility of $score(j) = best\text{-}score$.) In other words, all of the values $score(k+1)$ through $score(i-1)$ must be less than $M(k)$. If any are greater than $M(k)$, the algorithm instead returns the index of the first one that is greater. We use O_i to denote the event that none of the applicants in position $k+1$ through $i-1$ are chosen. Fortunately, the two events B_i and O_i are independent. The event O_i depends only on the relative ordering of the values in positions 1 through $i-1$, whereas B_i depends only on whether the value in position i is greater than the values in all other positions. The ordering of the values in positions 1 through $i-1$ does not affect whether the value in position i is greater than all of them, and the value in position i does not affect the ordering of the values in positions 1 through $i-1$. Thus, we can apply equation (C.17) on page 1188 to obtain

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\} .$$

We have $\Pr\{B_i\} = 1/n$ since the maximum is equally likely to be in any one of the n positions. For event O_i to occur, the maximum value in positions 1 through $i-1$, which is equally likely to be in any of these $i-1$ positions, must be in one of the first k positions. Consequently, $\Pr\{O_i\} = k/(i-1)$ and $\Pr\{S_i\} = k/(n(i-1))$. Using equation (5.14), we have

$$\begin{aligned}\Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\ &= \sum_{i=k+1}^{n} \frac{k}{n(i-1)}\end{aligned}$$

$$
\begin{aligned}
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} .
\end{aligned}
$$

We approximate by integrals to bound this summation from above and below. By the inequalities (A.19) on page 1150, we have

$$\int_{k}^{n} \frac{1}{x}\,dx \le \sum_{i=k}^{n-1} \frac{1}{i} \le \int_{k-1}^{n-1} \frac{1}{x}\,dx .$$

Evaluating these definite integrals gives us the bounds

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)) ,$$

which provide a rather tight bound for $\Pr\{S\}$. Because you wish to maximize your probability of success, let us focus on choosing the value of k that maximizes the lower bound on $\Pr\{S\}$. (Besides, the lower-bound expression is easier to maximize than the upper-bound expression.) Differentiating the expression $(k/n)(\ln n - \ln k)$ with respect to k, we obtain

$$\frac{1}{n}(\ln n - \ln k - 1) .$$

Setting this derivative equal to 0, we see that you maximize the lower bound on the probability when $\ln k = \ln n - 1 = \ln(n/e)$ or, equivalently, when $k = n/e$. Thus, if you implement our strategy with $k = n/e$, you succeed in hiring the best-qualified applicant with probability at least $1/e$.

Exercises

5.4-1

How many people must there be in a room before the probability that someone has the same birthday as you do is at least $1/2$? How many people must there be before the probability that at least two people have a birthday on July 4 is greater than $1/2$?

5.4-2

How many people must there be in a room before the probability that two people have the same birthday is at least 0.99? For that many people, what is the expected number of pairs of people who have the same birthday?

5.4-3
You toss balls into b bins until some bin contains two balls. Each toss is independent, and each ball is equally likely to end up in any bin. What is the expected number of ball tosses?

★ ***5.4-4***
For the analysis of the birthday paradox, is it important that the birthdays be mutually independent, or is pairwise independence sufficient? Justify your answer.

★ ***5.4-5***
How many people should be invited to a party in order to make it likely that there are *three* people with the same birthday?

★ ***5.4-6***
What is the probability that a k-string (defined on page 1179) over a set of size n forms a k-permutation? How does this question relate to the birthday paradox?

★ ***5.4-7***
You toss n balls into n bins, where each toss is independent and the ball is equally likely to end up in any bin. What is the expected number of empty bins? What is the expected number of bins with exactly one ball?

★ ***5.4-8***
Sharpen the lower bound on streak length by showing that in n flips of a fair coin, the probability is at least $1 - 1/n$ that a streak of length $\lg n - 2 \lg \lg n$ consecutive heads occurs.

Problems

5-1 Probabilistic counting
With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \ldots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number—see equation (3.31) on page 69).

For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

5-2 Searching an unsorted array

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then terminate; otherwise, continue the search by picking a new random index into A. Continue picking random indices into A until you find an index j such that $A[j] = x$ or until every element of A has been checked. This strategy may examine a given element more than once, because it picks from the whole set of indices each time.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that must be picked before x is found and RANDOM-SEARCH terminates?

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that must be picked before x is found and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that must be picked before all elements of A have been checked and RANDOM-SEARCH terminates?

Now consider a deterministic linear search algorithm. The algorithm, which we call DETERMINISTIC-SEARCH, searches A for x in order, considering $A[1]$, $A[2]$,

$A[3], \ldots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

f. Generalizing your solution to part (e), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH? Your answer should be a function of n and k.

g. Suppose that there are no indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that first randomly permutes the input array and then runs the deterministic linear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running times of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalize your solution to handle the case in which $k \geq 1$.

i. Which of the three searching algorithms would you use? Explain your answer.

Chapter notes

Bollobás [65], Hofri [223], and Spencer [420] contain a wealth of advanced probabilistic techniques. The advantages of randomized algorithms are discussed and surveyed by Karp [249] and Rabin [372]. The textbook by Motwani and Raghavan [336] gives an extensive treatment of randomized algorithms.

The RANDOMLY-PERMUTE procedure is by Durstenfeld [128], based on an earlier procedure by Fisher and Yates [143, p. 34].

Several variants of the hiring problem have been widely studied. These problems are more commonly referred to as "secretary problems." Examples of work in this area are the paper by Ajtai, Meggido, and Waarts [11] and another by Kleinberg [258], which ties the secretary problem to online ad auctions.