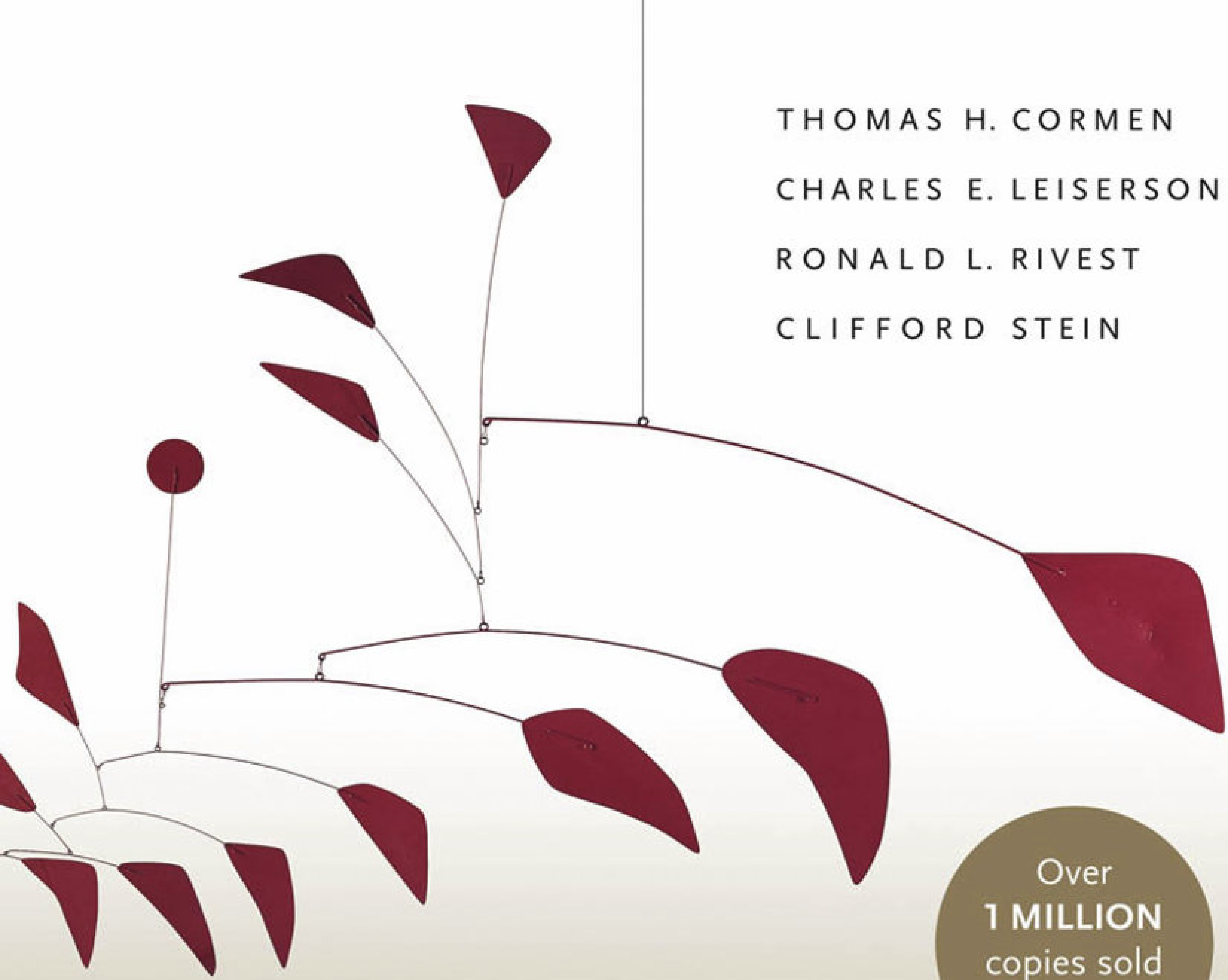
THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN



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

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$_\varepsilon$. 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$_\varepsilon$.

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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

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.

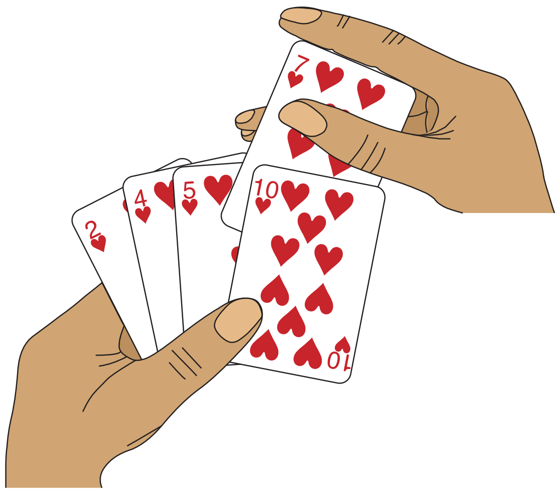


Figure 2.1 Sorting a hand of cards using insertion sort.

```
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***:

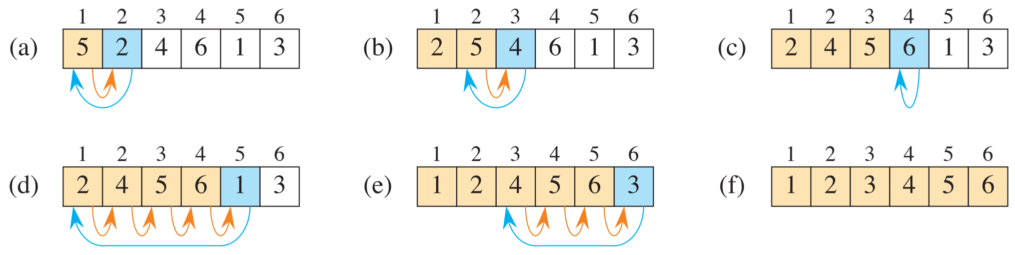


Figure 2.2 The operation of INSERTION-SORT(A, n), where A initially contains the sequence $\langle 5, 2, 4, 6, 1, 3 \rangle$ and $n = 6$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of 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      $A[j+1] = key$                                          $c_8$     $n - 1$
```

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

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

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

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.

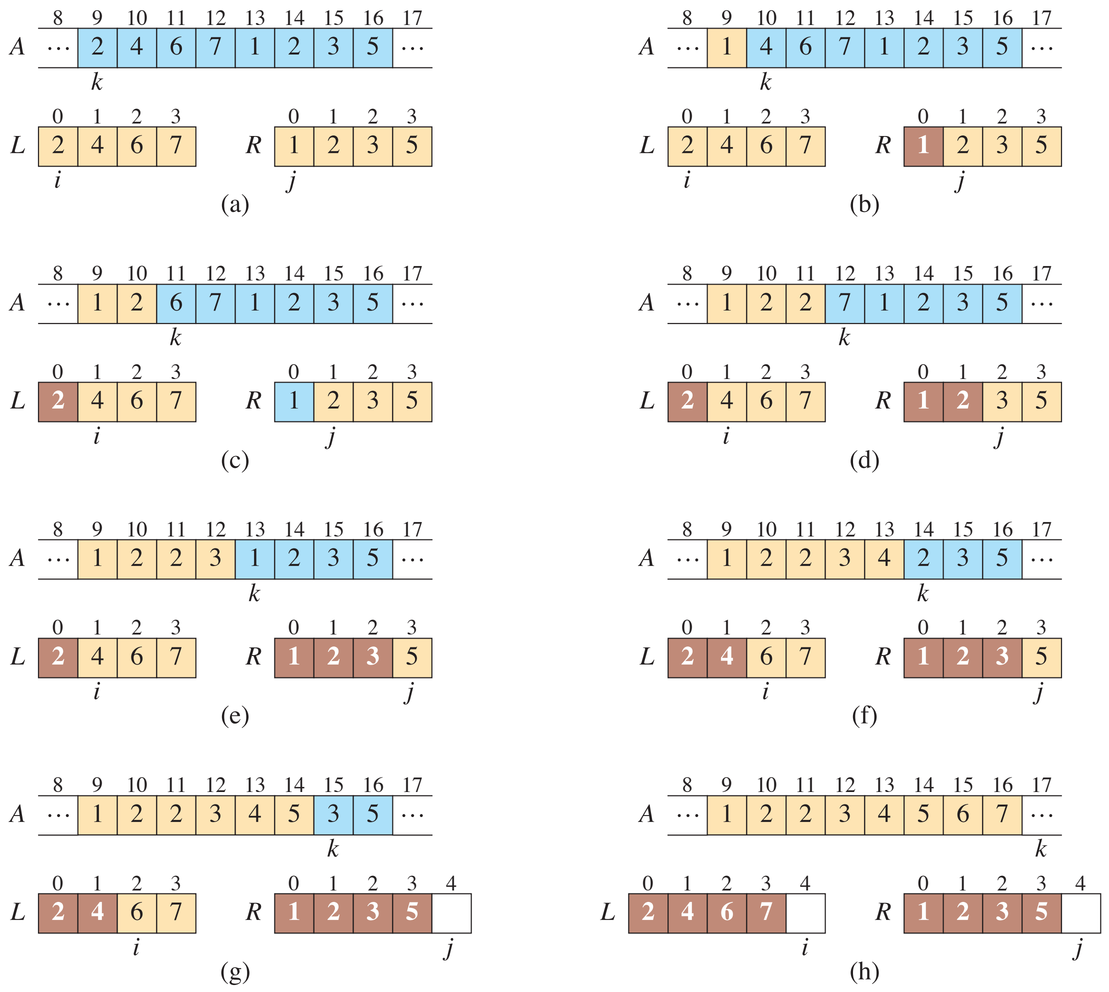


Figure 2.3 The operation of the **while** loop in lines 8–18 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9:16]$ contains the values $\langle 2, 4, 6, 7, 1, 2, 3, 5 \rangle$. After allocating and copying into the arrays L and R, the array L contains $\langle 2, 4, 6, 7 \rangle$, and the array 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)
if $p \geq r$                          // zero or one element?
    return
$q = \lfloor (p + r)/2 \rfloor$         // midpoint of $A[p:r]$
MERGE-SORT(A, p, q)                    // recursively sort $A[p:q]$
MERGE-SORT(A, q + 1, r)                // recursively sort $A[q+1:r]$
// Merge $A[p:q]$ and $A[q+1:r]$ into $A[p:r]$.
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).

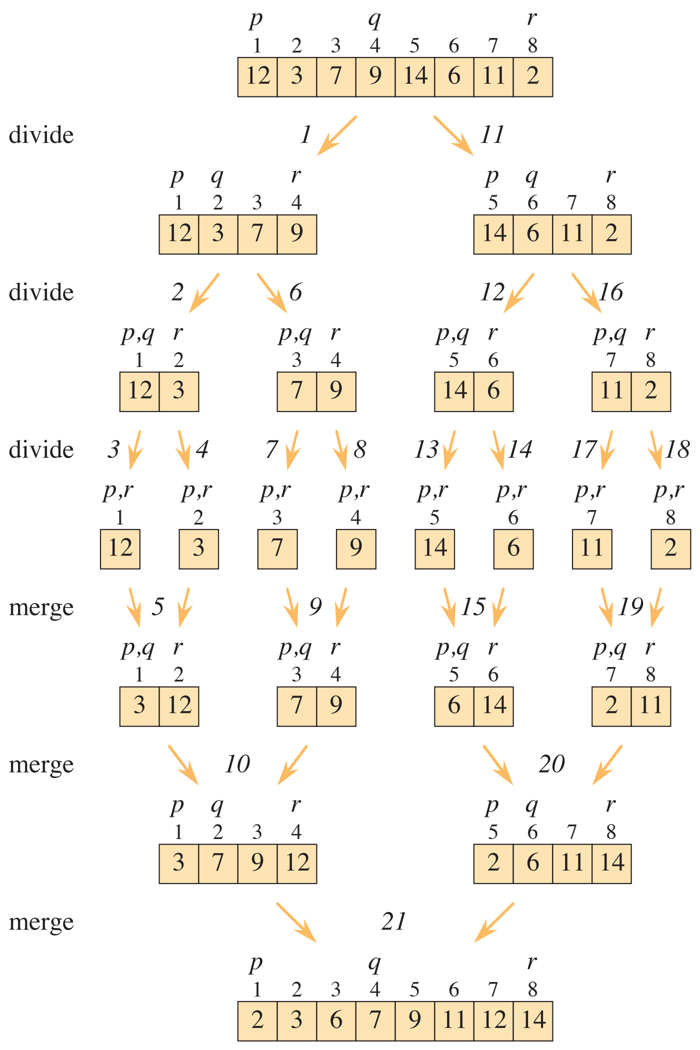


Figure 2.4 The operation of merge sort on the array A with length 8 that initially contains the sequence $\langle 12, 3, 7, 9, 14, 6, 11, 2 \rangle$. The indices p, q, and r into each subarray appear above their values. Numbers in italics indicate the order in which the MERGE-SORT and MERGE procedures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

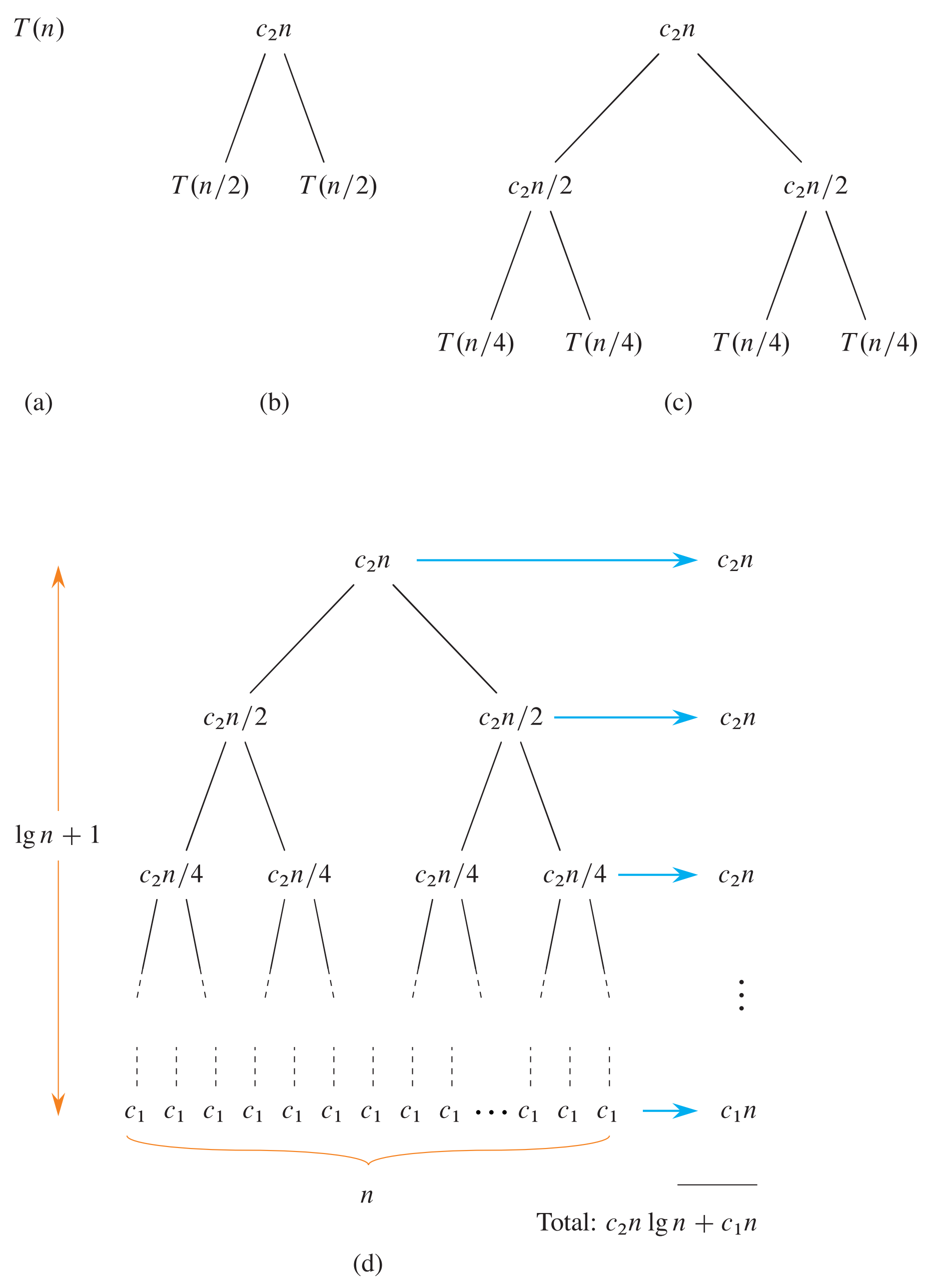


Figure 2.5 How to construct a recursion tree for the recurrence (2.4). Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels. Each level above the leaves contributes a total cost of c_2n, and the leaf 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 $c \leq 3$.

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 $i - 1$

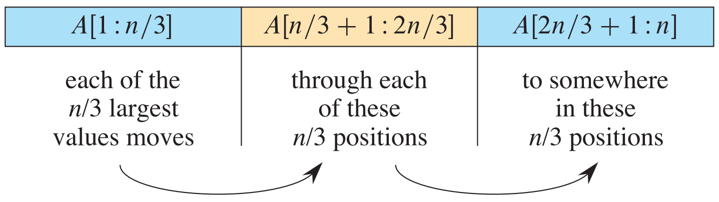

$A[1:n/3]$
$A[n/3+1:2n/3]$
$A[2n/3+1:n]$
each of the $n/3$ largest values moves
through each of these $n/3$ positions
to somewhere in these $n/3$ positions

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.

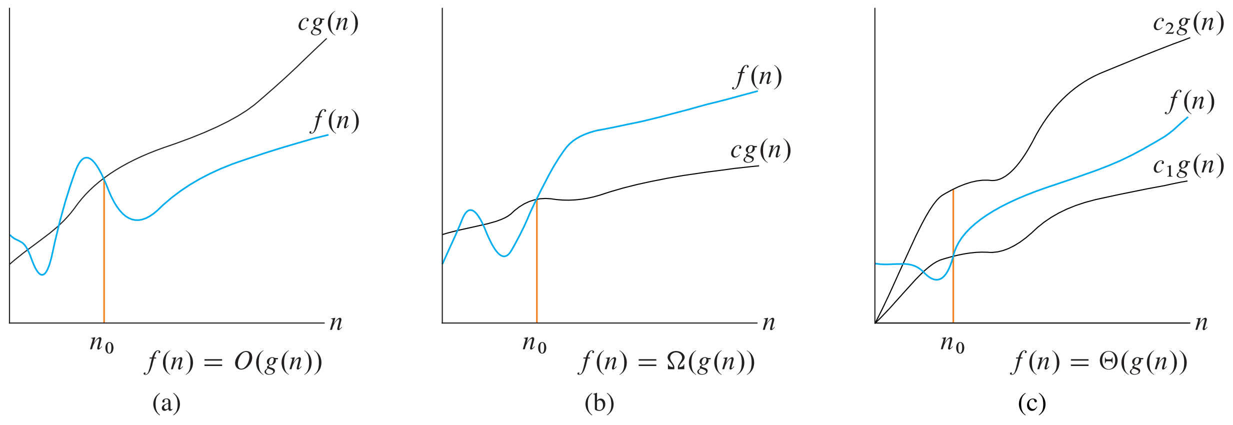


Figure 3.2 Graphic examples of the O, Ω, and Θ notations. In each part, the value of n_0 shown is the minimum possible value, but any greater value also works. **(a)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants 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

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

Figure 3.2(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

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

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

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

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

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

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

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

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

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

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

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

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

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 $n \leq 0$, 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

$$a = b^{\log_b a} , \tag{3.17}$$

$$\log_c(ab) = \log_c a + \log_c b , \tag{3.18}$$

$$\log_b a^n = n \log_b a ,$$

$$\log_b a = \frac{\log_c a}{\log_c b} , \tag{3.19}$$

$$\log_b(1/a) = -\log_b a , \tag{3.20}$$

$$\log_b a = \frac{1}{\log_a b} ,$$

$$a^{\log_b c} = c^{\log_b a} , \tag{3.21}$$

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 non-negative 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

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

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

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

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

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

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

3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, 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 $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, ..., $g_{29} = \Omega(g_{30})$. 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:

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

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

$$T(n) = 8T(n/2) + \Theta(1) . \qquad (4.9)$$

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

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

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 $T(n) \leq cn$ for all $n \geq n_0$, where $c, n_0 > 0$ are constants. Repeating the first two steps in the inequality chain yields

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

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 $cn + \Theta(n) \leq cn$, 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 $T(n) \leq cn$ and then argue

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

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 $T(n) \leq cn$. 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 $T(n) \leq cn$ 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.

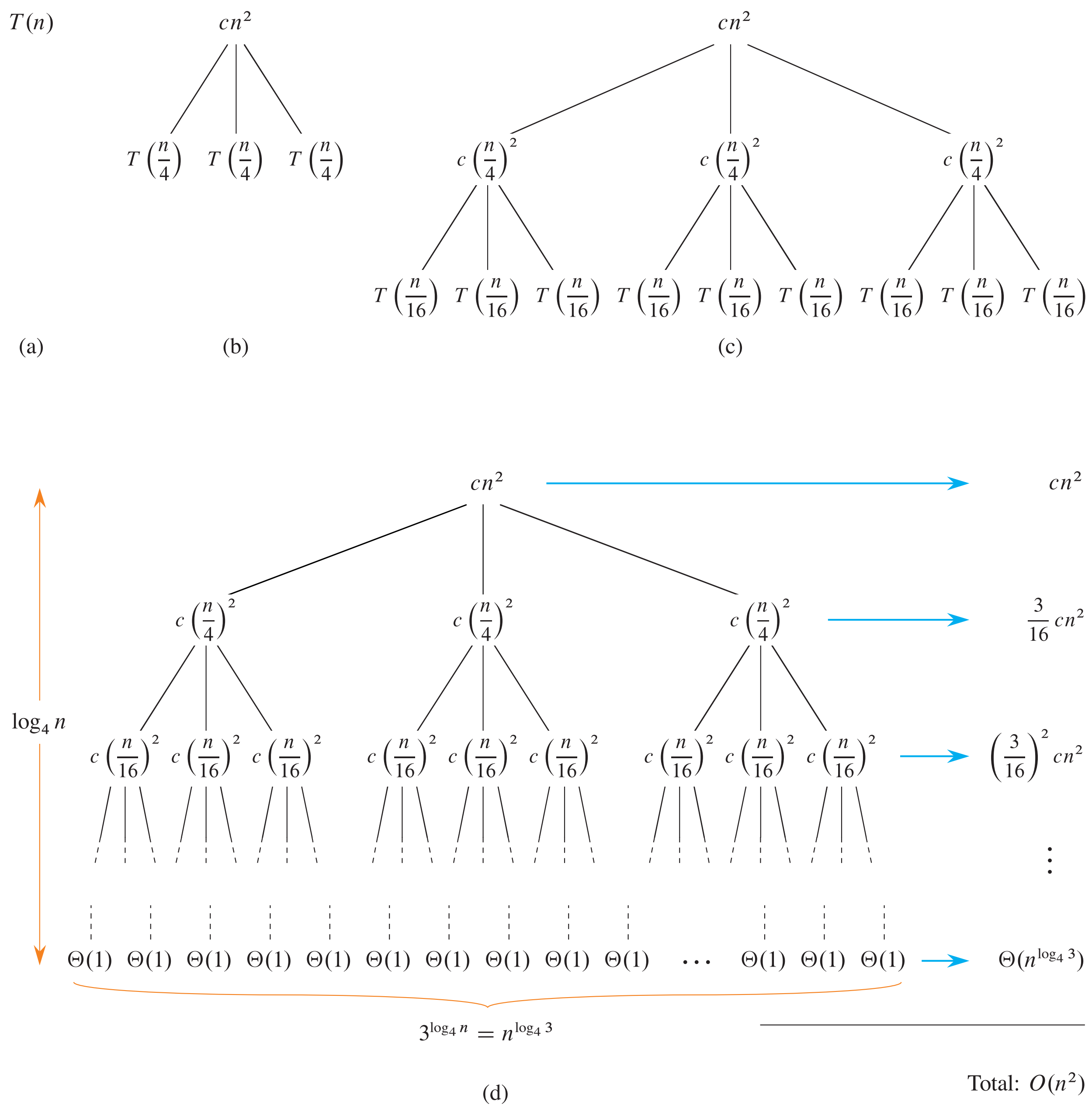
A recursion tree is best used to generate intuition for a good guess, which you can then verify by the substitution method. If you are meticulous when drawing out a recursion tree and summing the costs, however, you can use a recursion tree as a direct proof of a solution to a recurrence. But if you 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



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

$$T(n) = T(n/3) + T(2n/3) + \Theta(n) . \qquad (4.14)$$

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.

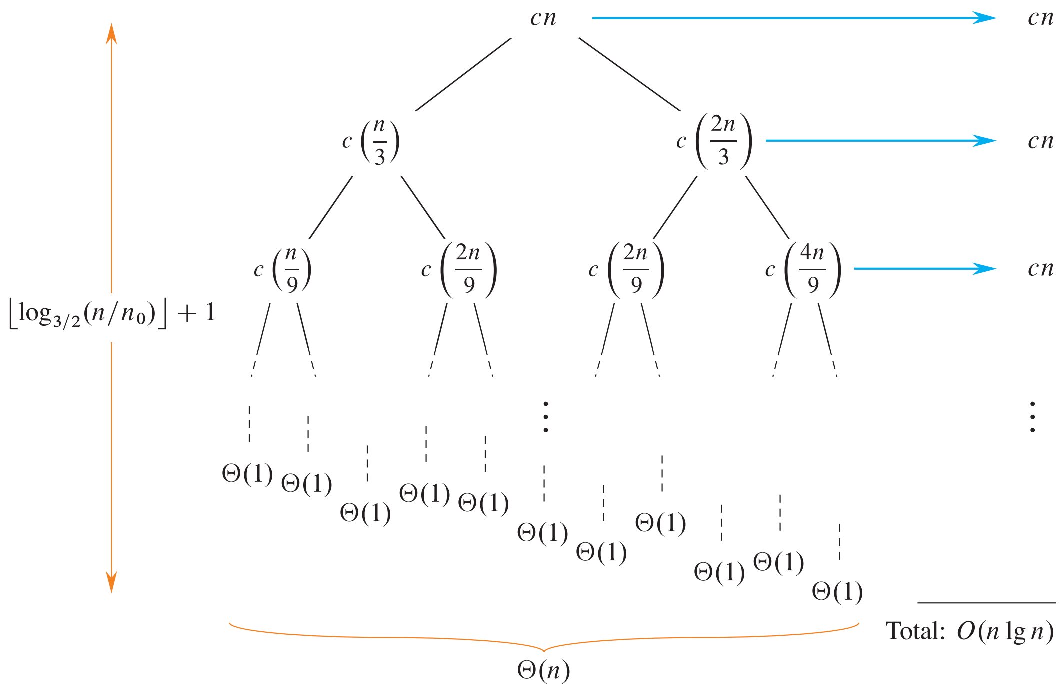

cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
cn
cn
$\lfloor \log_{3/2}(n/n_0) \rfloor + 1$
$\Theta(1)$
$\Theta(n)$
Total: $O(n \lg n)$

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 $(2/3)^h n < n_0 \leq (2/3)^{h-1} n$, 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

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

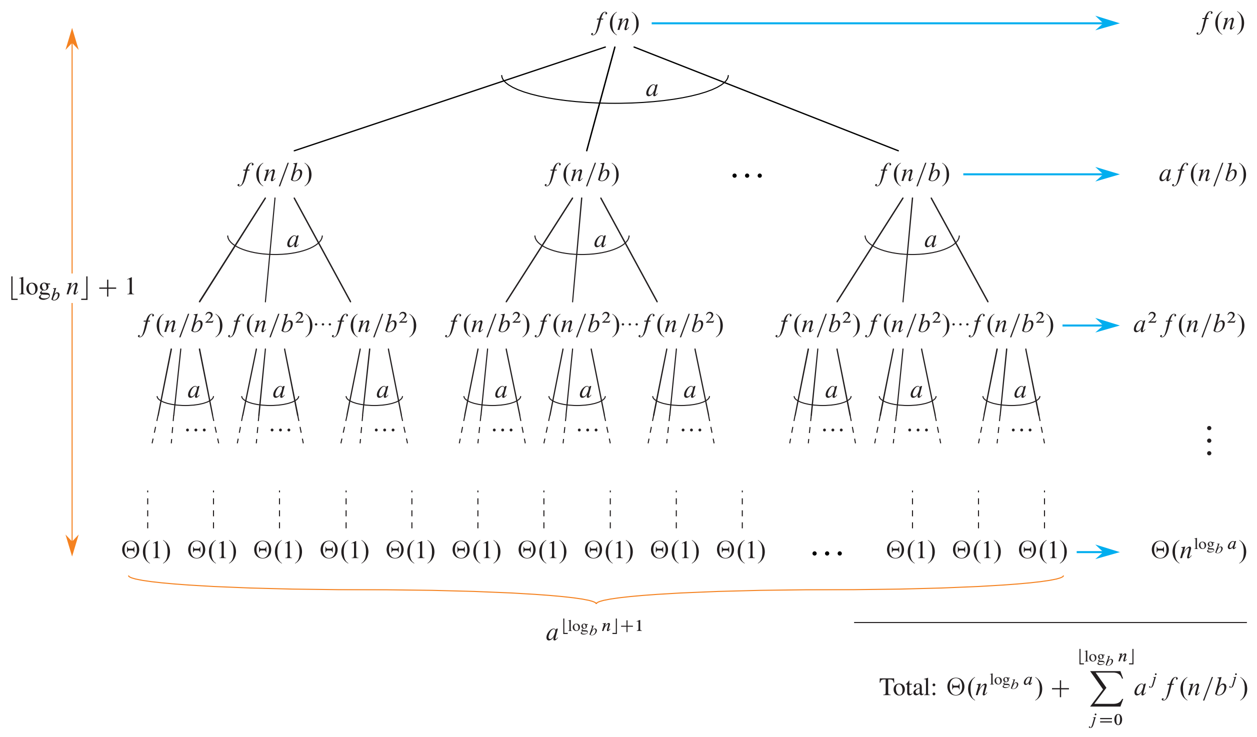
has solution

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

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
$f(n/b)$
$\cdots$
$f(n)$
$af(n/b)$
$\lfloor \log_b n \rfloor + 1$
$f(n/b^2)$
$a^2 f(n/b^2)$
$\Theta(1)$
$\Theta(n^{\log_b a})$
$a^{\lfloor \log_b n \rfloor + 1}$
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 $f(n)/d \leq f(\psi n) \leq d f(n)$ 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

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

As an example, consider the recurrence

$$T(n) = T(n/5) + T(7n/10) + n . \qquad (4.24)$$

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 $f(1) \leq f(2) \leq \cdots \leq f(m)$ 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)
1  best = 0                // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          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 $A[\text{RANDOM}(1, n)]$
```

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
$offset = \text{RANDOM}(1, n)$
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
    $i = \text{RANDOM}(1, k)$
    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

$$\begin{aligned} &\Pr\left\{ \bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil} \right\} \\ &\quad \le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \Pr\{A_{i,2\lceil \lg n \rceil}\} \quad \text{(by Boole's inequality (C.21) on page 1190)} \\ &\quad \le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} \frac{1}{n^2} \\ &\quad < \sum_{i=1}^{n} \frac{1}{n^2} \\ &\quad = \frac{1}{n} . \end{aligned} \tag{5.10}$$

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

$$\begin{aligned}
\Pr\left\{\bigcup_{i=1}^{n} A_{i,r\lceil\lg n\rceil}\right\} &\le \sum_{i=1}^{n} \Pr\{A_{i,r\lceil\lg n\rceil}\} \qquad \text{(by Boole's inequality (C.21))} \\
&\le \sum_{i=1}^{n} \frac{1}{n^r} \\
&= \frac{1}{n^{r-1}} .
\end{aligned}$$

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 $\mathrm{E}[X_{(1/2)\lg n}] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, 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.

Part II Sorting and Order Statistics

Introduction

This part presents several algorithms that solve the following ***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 $a'_1 \le a'_2 \le \cdots \le a'_n$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as a linked list.

The structure of the data

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a ***record***. Each record contains a ***key***, which is the value to be sorted. The remainder of the record consists of ***satellite data***, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, it often pays to permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

In a sense, it is these implementation details that distinguish an algorithm from a full-blown program. A sorting algorithm describes the *method* to determine the sorted order, regardless of whether what's being sorted are individual numbers or large records containing many bytes of satellite data. Thus, when focusing on the problem of sorting, we typically assume that the input consists only of numbers. Translating an algorithm for sorting numbers into a program for sorting records is conceptually straightforward, although in a given engineering situation other subtleties may make the actual programming task a challenge.

Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. There are several reasons:

- Sometimes an application inherently needs to sort information. For example, in order to prepare customer statements, banks need to sort checks by check number.
- Algorithms often use sorting as a key subroutine. For example, a program that renders graphical objects which are layered on top of each other might have to sort the objects according to an "above" relation so that it can draw these objects from bottom to top. We will see numerous algorithms in this text that use sorting as a subroutine.
- We can draw from among a wide variety of sorting algorithms, and they employ a rich set of techniques. In fact, many important techniques used throughout algorithm design appear in sorting algorithms that have been developed over the years. In this way, sorting is also a problem of historical interest.
- We can prove a nontrivial lower bound for sorting (as we'll do in Chapter 8). Since the best upper bounds match the lower bound asymptotically, we can conclude that certain of our sorting algorithms are asymptotically optimal. Moreover, we can use the lower bound for sorting to prove lower bounds for various other problems.
- Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as prior knowledge about the keys and satellite data, the memory hierarchy (caches and virtual memory) of the host computer, and the software environment. Many of these issues are best dealt with at the algorithmic level, rather than by "tweaking" the code.

Sorting algorithms

We introduced two algorithms that sort n real numbers in Chapter 2. Insertion sort takes $\Theta(n^2)$ time in the worst case. Because its inner loops are tight, however, it is a fast sorting algorithm for small input sizes. Moreover, unlike merge sort, it sorts ***in place***, meaning that at most a constant number of elements of the input array are ever stored outside the array, which can be advantageous for space efficiency. Merge sort has a better asymptotic running time, $\Theta(n \lg n)$, but the MERGE procedure it uses does not operate in place. (We'll see a parallelized version of merge sort in Section 26.3.)

This part introduces two more algorithms that sort arbitrary real numbers. Heapsort, presented in Chapter 6, sorts n numbers in place in $O(n \lg n)$ time. It uses an important data structure, called a heap, which can also implement a priority queue.

Quicksort, in Chapter 7, also sorts n numbers in place, but its worst-case running time is $\Theta(n^2)$. Its expected running time is $\Theta(n \lg n)$, however, and it generally outperforms heapsort in practice. Like insertion sort, quicksort has tight code, and so the hidden constant factor in its running time is small. It is a popular algorithm for sorting large arrays.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements. Chapter 8 begins by introducing the decision-tree model in order to study the performance limitations of comparison sorts. Using this model, we prove a lower bound of $\Omega(n \lg n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

Chapter 8 then goes on to show that we might be able to beat this lower bound of $\Omega(n \lg n)$ if an algorithm can gather information about the sorted order of the input by means other than comparing elements. The counting sort algorithm, for example, assumes that the input numbers belong to the set $\{0, 1, \ldots, k\}$. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $\Theta(k + n)$ time. Thus, when $k = O(n)$, counting sort runs in time that is linear in the size of the input array. A related algorithm, radix sort, can be used to extend the range of counting sort. If there are n integers to sort, each integer has d digits, and each digit can take on up to k possible values, then radix sort can sort the numbers in $\Theta(d(n + k))$ time. When d is a constant and k is $O(n)$, radix sort runs in linear time. A third algorithm, bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array. It can sort n real numbers uniformly distributed in the half-open interval $[0, 1)$ in average-case $O(n)$ time.

The table on the following page summarizes the running times of the sorting algorithms from Chapters 2 and 6–8. As usual, n denotes the number of items to sort. For counting sort, the items to sort are integers in the set $\{0, 1, \ldots, k\}$. For radix sort, each item is a d-digit number, where each digit takes on k possible values. For bucket sort, we assume that the keys are real numbers uniformly distributed in the half-open interval $[0, 1)$. The rightmost column gives the average-case or expected running time, indicating which one it gives when it differs from the worst-case running time. We omit the average-case running time of heapsort because we do not analyze it in this book.

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Order statistics

The ith order statistic of a set of n numbers is the ith smallest number in the set. You can, of course, select the ith order statistic by sorting the input and indexing the ith element of the output. With no assumptions about the input distribution, this method runs in $\Omega(n \lg n)$ time, as the lower bound proved in Chapter 8 shows.

Chapter 9 shows how to find the ith smallest element in $O(n)$ time, even when the elements are arbitrary real numbers. We present a randomized algorithm with tight pseudocode that runs in $\Theta(n^2)$ time in the worst case, but whose expected running time is $O(n)$. We also give a more complicated algorithm that runs in $O(n)$ worst-case time.

Background

Although most of this part does not rely on difficult mathematics, some sections do require mathematical sophistication. In particular, analyses of quicksort, bucket sort, and the order-statistic algorithm use probability, which is reviewed in Appendix C, and the material on probabilistic analysis and randomized algorithms in Chapter 5.

6 Heapsort

This chapter introduces another sorting algorithm: heapsort. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a "heap," to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

The term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming languages Java and Python provide. Please don't be confused. The heap data structure is *not* garbage-collected storage. This book is consistent in using the term "heap" to refer to the data structure, not the storage class.

6.1 Heaps

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A[1:n]$ that represents a heap is an object with an attribute $A.\mathit{heap\text{-}size}$, which represents how many elements in the heap are stored within array A. That is, although $A[1:n]$ may contain numbers, only the elements in $A[1:A.\mathit{heap\text{-}size}]$, where $0 \le A.\mathit{heap\text{-}size} \le n$, are valid elements of the heap. If $A.\mathit{heap\text{-}size} = 0$, then the heap is empty. The root of the tree is $A[1]$, and given the index i of a node,

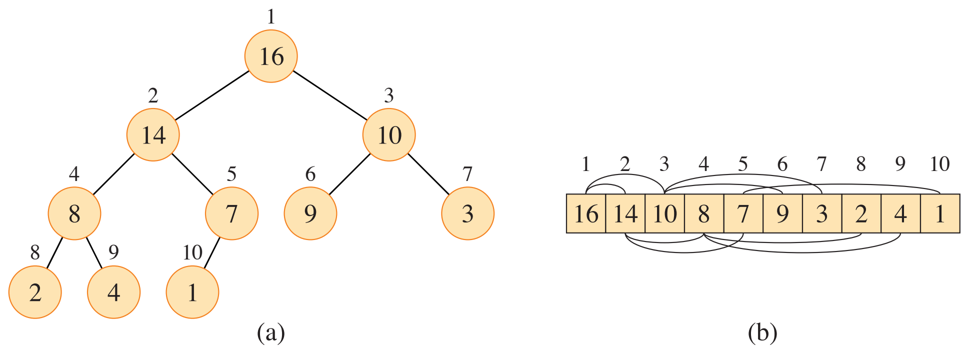


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, with parents always to the left of their children. The tree has height 3, and the node at index 4 (with value 8) has height 1.

there's a simple way to compute the indices of its parent, left child, and right child with the one-line procedures PARENT, LEFT, and RIGHT.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left by one bit position and then adding 1. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. Good implementations of heapsort often implement these procedures as macros or inline procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way: the ***min-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \le A[i] \ .$$

The smallest element in a min-heap is at the root.

The heapsort algorithm uses max-heaps. Min-heaps commonly implement priority queues, which we discuss in Section 6.5. We'll be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term "heap."

Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 6.1-2). As we'll see, the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The procedures MAX-HEAP-INSERT, MAX-HEAP-EXTRACT-MAX, MAX-HEAP-INCREASE-KEY, and MAX-HEAP-MAXIMUM allow the heap data structure to implement a priority queue. They run in $O(\lg n)$ time plus the time for mapping between objects being inserted into the priority queue and indices in the heap.

Exercises

6.1-1
What are the minimum and maximum numbers of elements in a heap of height h?

6.1-2
Show that an n-element heap has height $\lfloor \lg n \rfloor$.

6.1-3
Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

6.1-4
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5
At which levels in a max-heap might the kth largest element reside, for $2 \le k \le \lfloor n/2 \rfloor$, assuming that all elements are distinct?

6.1-6
Is an array that is in sorted order a min-heap?

6.1-7
Is the array with values $\langle 33, 19, 20, 15, 13, 10, 2, 13, 16, 12 \rangle$ a max-heap?

6.1-8
Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Maintaining the heap property

The procedure MAX-HEAPIFY on the facing page maintains the max-heap property. Its inputs are an array A with the *heap-size* attribute and an index i into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Figure 6.2 illustrates the action of MAX-HEAPIFY. Each step determines the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ and stores the index of the largest element in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and nothing else needs to be done. Otherwise, one of the two children contains the largest element. Positions i and *largest* swap their contents, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, just had its value decreased, and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, MAX-HEAPIFY calls itself recursively on that subtree.

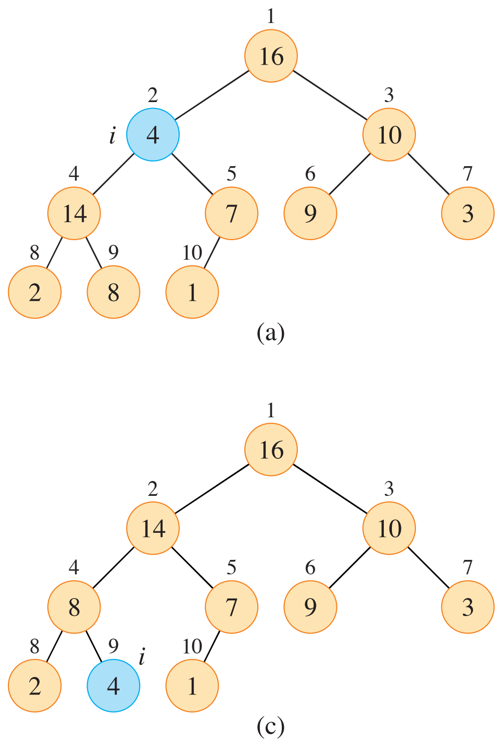


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. The node that potentially violates the max-heap property is shown in blue. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After $A[4]$ and $A[9]$ are swapped, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

To analyze MAX-HEAPIFY, let $T(n)$ be the worst-case running time that the procedure takes on a subtree of size at most n. For a tree rooted at a given node i, the running time is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$ (see Exercise 6.2-2), and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \le T(2n/3) + \Theta(1) . \tag{6.1}$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1 on page 102), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Exercises

6.2-1
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2
Show that each child of the root of an n-node heap is the root of a subtree containing at most $2n/3$ nodes. What is the smallest constant α such that each subtree has at most αn nodes? How does that affect the recurrence (6.1) and its solution?

6.2-3
Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare with that of MAX-HEAPIFY?

6.2-4
What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

6.2-5
What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.\mathit{heap\text{-}size}/2$?

6.2-6
The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, for which some compilers might produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-7
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

6.3 Building a heap

The procedure BUILD-MAX-HEAP converts an array $A[1:n]$ into a max-heap by calling MAX-HEAPIFY in a bottom-up manner. Exercise 6.1-8 says that the elements in the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one. Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

```
BUILD-MAX-HEAP($A, n$)
1  $A.heap\text{-}size = n$
2  for $i = \lfloor n/2 \rfloor$ downto 1
3      MAX-HEAPIFY($A, i$)
```

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1$, $i + 2, \ldots, n$ is the root of a max-heap.

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: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1$, $i + 2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

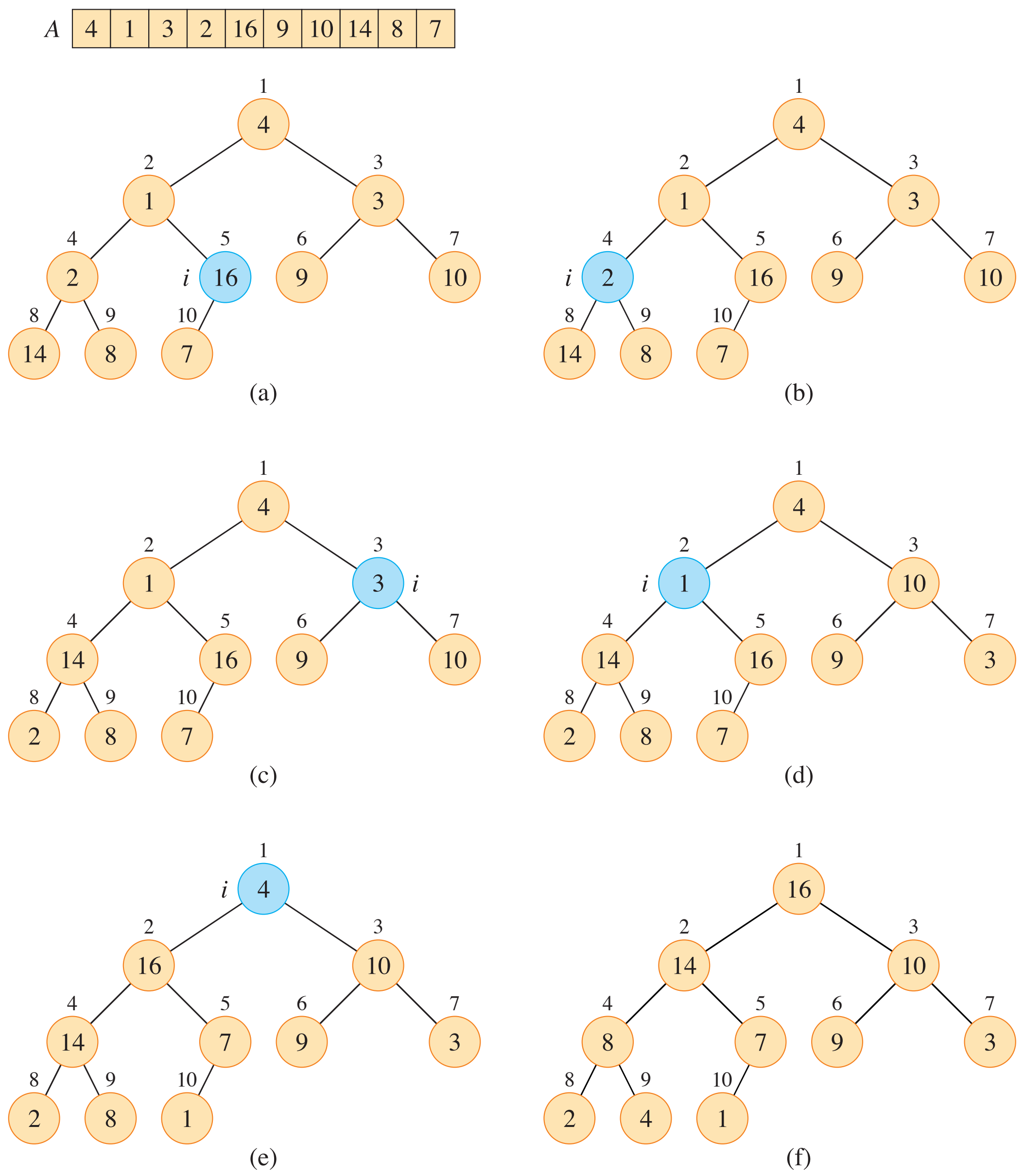


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. The node indexed by i in each iteration is shown in blue. **(a)** A 10-element input array A and the binary tree it represents. The loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

Termination: The loop makes exactly $\lfloor n/2 \rfloor$ iterations, and so it terminates. At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not as tight as it can be.

We can derive a tighter asymptotic bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and that the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-4).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$. Letting c be the constant implicit in the asymptotic notation, we can express the total cost of BUILD-MAX-HEAP as being bounded from above by $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil ch$. As Exercise 6.3-2 shows, we have $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$. Since $\lceil x \rceil \leq 2x$ for any $x \geq 1/2$, we have $\lceil n/2^{h+1} \rceil \leq n/2^h$. We thus obtain

$$
\begin{aligned}
\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&= cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\
&\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\leq cn \cdot \frac{1/2}{(1-1/2)^2} \quad \text{(by equation (A.11) on page 1142 with } x = 1/2) \\
&= O(n) .
\end{aligned}
$$

Hence, we can build a max-heap from an unordered array in linear time.

To build a min-heap, use the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-3). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

Exercises

6.3-1
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2
Show that $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$.

6.3-3
Why does the loop index i in line 2 of BUILD-MAX-HEAP decrease from $\lfloor n/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor n/2 \rfloor$?

6.3-4
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

6.4 The heapsort algorithm

The heapsort algorithm, given by the procedure HEAPSORT, starts by calling the BUILD-MAX-HEAP procedure to build a max-heap on the input array $A[1:n]$. Since the maximum element of the array is stored at the root $A[1]$, HEAPSORT can place it into its correct final position by exchanging it with $A[n]$. If the procedure then discards node n from the heap—and it can do so by simply decrementing $A.heap\text{-}size$—the children of the root remain max-heaps, but the new root element might violate the max-heap property. To restore the max-heap property, the procedure just calls MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1:n-1]$. The HEAPSORT procedure then repeats this process for the max-heap of size $n-1$ down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

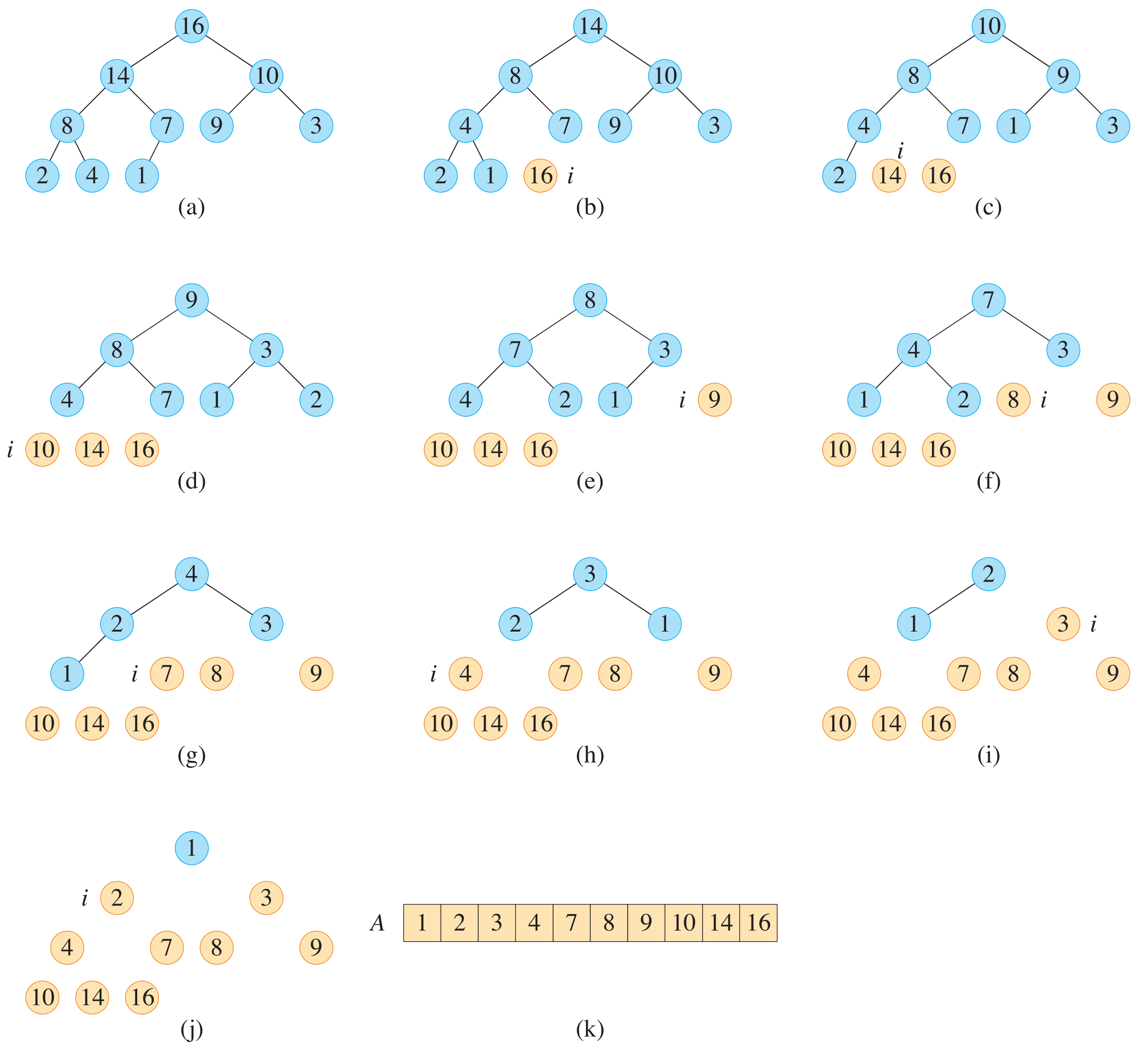


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only blue nodes remain in the heap. Tan nodes contain the largest values in the array, in sorted order. **(k)** The resulting sorted array A.

The HEAPSORT procedure takes $O(n \lg n)$ time, since the call to BUILD-MAX-HEAP takes $O(n)$ time and each of the $n-1$ calls to MAX-HEAPIFY takes $O(\lg n)$ time.

Exercises

6.4-1

Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2

Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1:i]$ is a max-heap containing the i smallest elements of $A[1:n]$, and the subarray $A[i+1:n]$ contains the $n-i$ largest elements of $A[1:n]$, sorted.

6.4-3

What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? How about if the array is already sorted in decreasing order?

6.4-4

Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

★ ***6.4-5***

Show that when all the elements of A are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

6.5 Priority queues

In Chapter 8, we will see that any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons and hence $\Omega(n \lg n)$ time. Therefore, heapsort is asymptotically optimal among comparison-based sorting algorithms. Yet, a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We'll focus here on how to implement max-priority queues, which are in turn based on max-heaps. Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT(S, x, k) inserts the element x with key k into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Among their other applications, you can use max-priority queues to schedule jobs on a computer shared among multiple users. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT. We'll see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 21 and 22.

When you use a heap to implement a priority queue within a given application, elements of the priority queue correspond to objects in the application. Each object contains a key. If the priority queue is implemented by a heap, you need to determine which application object corresponds to a given heap element, and vice versa. Because the heap elements are stored in an array, you need a way to map application objects to and from array indices.

One way to map between application objects and heap elements uses ***handles***, which are additional information stored in the objects and heap elements that give enough information to perform the mapping. Handles are often implemented to be opaque to the surrounding code, thereby maintaining an abstraction barrier between the application and the priority queue. For example, the handle within an application object might contain the corresponding index into the heap array. But since only the code for the priority queue accesses this index, the index is entirely hidden from the application code. Because heap elements change locations within the array during heap operations, an actual implementation of the priority queue, upon relocating a heap element, must also update the array indices in the corresponding handles. Conversely, each element in the heap might contain a pointer to the corresponding application object, but the heap element knows this pointer as only an opaque handle and the application maps this handle to an application object. Typically, the worst-case overhead for maintaining handles is $O(1)$ per access.

As an alternative to incorporating handles in application objects, you can store within the priority queue a mapping from application objects to array indices in the heap. The advantage of doing so is that the mapping is contained entirely within the priority queue, so that the application objects need no further embellishment. The disadvantage lies in the additional cost of establishing and maintaining the mapping. One option for the mapping is a hash table (see Chapter 11).[1] The added expected time for a hash table to map an object to an array index is just $O(1)$, though the worst-case time can be as bad as $\Theta(n)$.

Let's see how to implement the operations of a max-priority queue using a max-heap. In the previous sections, we treated the array elements as the keys to be sorted, implicitly assuming that any satellite data moved with the corresponding keys. When a heap implements a priority queue, we instead treat each array element as a pointer to an object in the priority queue, so that the object is analogous to the satellite data when sorting. We further assume that each such object has an attribute *key*, which determines where in the heap the object belongs. For a heap implemented by an array A, we refer to $A[i].key$.

The procedure MAX-HEAP-MAXIMUM on the facing page implements the MAXIMUM operation in $\Theta(1)$ time, and MAX-HEAP-EXTRACT-MAX implements the operation EXTRACT-MAX. MAX-HEAP-EXTRACT-MAX is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure. We implicitly assume that MAX-HEAPIFY compares priority-queue objects based on their *key* attributes. We also assume that when MAX-HEAPIFY exchanges elements in the array, it is exchanging pointers and also that it updates the mapping between objects and array indices. The running time of MAX-HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY, plus whatever overhead is incurred within MAX-HEAPIFY for mapping priority-queue objects to array indices.

The procedure MAX-HEAP-INCREASE-KEY on page 176 implements the INCREASE-KEY operation. It first verifies that the new key k will not cause the key in the object x to decrease, and if there is no problem, it gives x the new key value. The procedure then finds the index i in the array corresponding to object x,

[1] In Python, dictionaries are implemented with hash tables.

```
MAX-HEAP-MAXIMUM(A)
1  if $A.heap\text{-}size < 1$
2      error "heap underflow"
3  return $A[1]$
```

```
MAX-HEAP-EXTRACT-MAX(A)
1  $max$ = MAX-HEAP-MAXIMUM(A)
2  $A[1] = A[A.heap\text{-}size]$
3  $A.heap\text{-}size = A.heap\text{-}size - 1$
4  MAX-HEAPIFY(A, 1)
5  return $max$
```

so that $A[i]$ is x. Because increasing the key of $A[i]$ might violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT on page 19, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As MAX-HEAP-INCREASE-KEY traverses this path, it repeatedly compares an element's key to that of its parent, exchanging pointers and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds. (See Exercise 6.5-7 for a precise loop invariant.) Like MAX-HEAPIFY when used in a priority queue, MAX-HEAP-INCREASE-KEY updates the information that maps objects to array indices when array elements are exchanged. Figure 6.5 shows an example of a MAX-HEAP-INCREASE-KEY operation. In addition to the overhead for mapping priority queue objects to array indices, the running time of MAX-HEAP-INCREASE-KEY on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT on the next page implements the INSERT operation. It takes as inputs the array A implementing the max-heap, the new object x to be inserted into the max-heap, and the size n of array A. The procedure first verifies that the array has room for the new element. It then expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls MAX-HEAP-INCREASE-KEY to set the key of this new element to its correct value and maintain the max-heap property. The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$ plus the overhead for mapping priority queue objects to indices.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time, plus the overhead for mapping priority queue objects to array indices.

```
MAX-HEAP-INCREASE-KEY($A, x, k$)
1  if $k < x.key$
2      error "new key is smaller than current key"
3  $x.key = k$
4  find the index $i$ in array $A$ where object $x$ occurs
5  while $i > 1$ and $A[\text{PARENT}(i)].key < A[i].key$
6      exchange $A[i]$ with $A[\text{PARENT}(i)]$, updating the information that maps
           priority queue objects to array indices
7      $i = \text{PARENT}(i)$
```

```
MAX-HEAP-INSERT($A, x, n$)
1  if $A.heap\text{-}size == n$
2      error "heap overflow"
3  $A.heap\text{-}size = A.heap\text{-}size + 1$
4  $k = x.key$
5  $x.key = -\infty$
6  $A[A.heap\text{-}size] = x$
7  map $x$ to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY($A, x, k$)
```

Exercises

6.5-1

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT(A, 10) on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-3

Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

6.5-4

Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY(A, x, k) in a max-heap. What is the running time of your procedure?

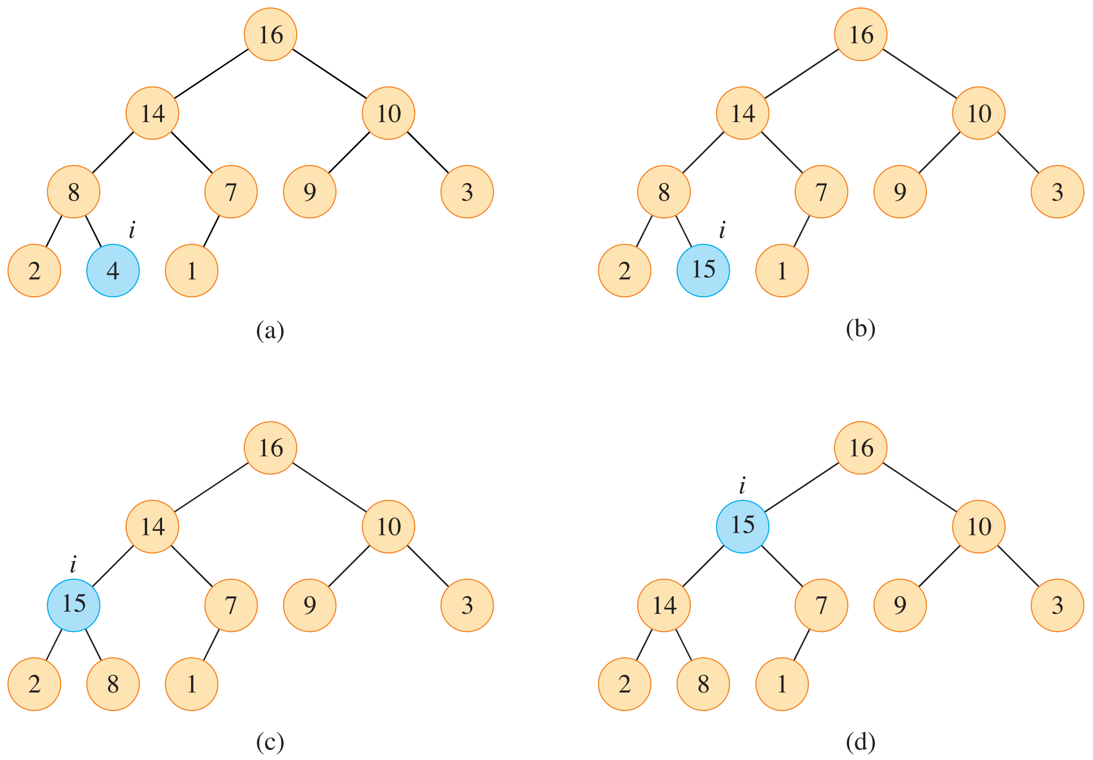


Figure 6.5 The operation of MAX-HEAP-INCREASE-KEY. Only the key of each element in the priority queue is shown. The node indexed by i in each iteration is shown in blue. **(a)** The max-heap of Figure 6.4(a) with i indexing the node whose key is about to be increased. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 5–7, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

6.5-5

Why does MAX-HEAP-INSERT bother setting the key of the inserted object to $-\infty$ in line 5 given that line 8 will set the object's key to the desired value?

6.5-6

Professor Uriah suggests replacing the **while** loop of lines 5–7 in MAX-HEAP-INCREASE-KEY by a call to MAX-HEAPIFY. Explain the flaw in the professor's idea.

6.5-7

Argue the correctness of MAX-HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the **while** loop of lines 5–7:

a. If both nodes PARENT(i) and LEFT(i) exist, then $A[\text{PARENT}(i)].\mathit{key} \geq A[\text{LEFT}(i)].\mathit{key}$.

b. If both nodes PARENT(i) and RIGHT(i) exist, then $A[\text{PARENT}(i)].\mathit{key} \geq A[\text{RIGHT}(i)].\mathit{key}$.

c. The subarray $A[1:A.\mathit{heap\text{-}size}]$ satisfies the max-heap property, except that there may be one violation, which is that $A[i].\mathit{key}$ may be greater than $A[\text{PARENT}(i)].\mathit{key}$.

You may assume that the subarray $A[1:A.\mathit{heap\text{-}size}]$ satisfies the max-heap property at the time MAX-HEAP-INCREASE-KEY is called.

6.5-8
Each exchange operation on line 6 of MAX-HEAP-INCREASE-KEY typically requires three assignments, not counting the updating of the mapping from objects to array indices. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments to just one assignment.

6.5-9
Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.3.)

6.5-10
The operation MAX-HEAP-DELETE(A, x) deletes the object x from max-heap A. Give an implementation of MAX-HEAP-DELETE for an n-element max-heap that runs in $O(\lg n)$ time plus the overhead for mapping priority queue objects to array indices.

6.5-11
Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a min-heap for k-way merging.)

Problems

6-1 Building a heap using insertion
One way to build a heap is by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the procedure BUILD-MAX-HEAP$'$ on the facing page. It assumes that the objects being inserted are just the heap elements.

```
BUILD-MAX-HEAP′$(A, n)$
1  $A.heap\text{-}size = 1$
2  for $i = 2$ to $n$
3      MAX-HEAP-INSERT$(A, A[i], n)$
```

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

6-2 Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is $O(1)$ per operation.

a. Describe how to represent a d-ary heap in an array.

b. Using Θ-notation, express the height of a d-ary heap of n elements in terms of n and d.

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INCREASE-KEY in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

6-3 Young tableaus

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \le mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m,n] < \infty$.

c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m - 1) \times n$ or an $m \times (n - 1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Explain why your implementation of EXTRACT-MIN runs in $O(m + n)$ time.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Chapter notes

The heapsort algorithm was invented by Williams [456], who also described how to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure was suggested by Floyd [145]. Schaffer and Sedgewick [395] showed that in the best case, the number of times elements move in the heap during heapsort is approximately $(n/2) \lg n$ and that the average number of moves is approximately $n \lg n$.

We use min-heaps to implement min-priority queues in Chapters 15, 21, and 22. Other, more complicated, data structures give better time bounds for certain min-priority queue operations. Fredman and Tarjan [156] developed Fibonacci heaps, which support INSERT and DECREASE-KEY in $O(1)$ amortized time (see Chapter 16). That is, the average worst-case running time for these operations is $O(1)$. Brodal, Lagogiannis, and Tarjan [73] subsequently devised strict Fibonacci heaps, which make these time bounds the actual running times. If the keys are unique and drawn from the set $\{0, 1, \ldots, n - 1\}$ of nonnegative integers, van Emde Boas trees [440, 441] support the operations INSERT, DELETE, SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR in $O(\lg \lg n)$ time.

If the data are b-bit integers, and the computer memory consists of addressable b-bit words, Fredman and Willard [157] showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ time. Thorup [436] has improved the $O(\sqrt{\lg n})$ bound to $O(\lg \lg n)$ time by using randomized hashing, requiring only linear space.

An important special case of priority queues occurs when the sequence of EXTRACT-MIN operations is ***monotone***, that is, the values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in several important applications, such as Dijkstra's single-source shortest-paths algorithm, which we discuss in Chapter 22, and in discrete-event simulation. For Dijkstra's algorithm it is particularly important that the DECREASE-KEY operation be implemented efficiently. For the monotone case, if the data are integers in the range $1, 2, \ldots, C$, Ahuja, Mehlhorn, Orlin, and Tarjan [8] describe how to implement EXTRACT-MIN and INSERT in $O(\lg C)$ amortized time (Chapter 16 presents amortized analysis) and DECREASE-KEY in $O(1)$ time, using a data structure called a radix heap. The $O(\lg C)$ bound can be improved to $O(\sqrt{\lg C})$ using Fibonacci heaps in conjunction with radix heaps. Cherkassky, Goldberg, and Silverstein [90] further improved the bound to $O(\lg^{1/3+\epsilon} C)$ expected time by combining the multilevel bucketing structure of Denardo and Fox [112] with the heap of Thorup mentioned earlier. Raman [375] further improved these results to obtain a bound of $O\left(\min\left\{\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n\right\}\right)$, for any fixed $\epsilon > 0$.

Many other variants of heaps have been proposed. Brodal [72] surveys some of these developments.

7 Quicksort

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on average: its expected running time is $\Theta(n \lg n)$ when all numbers are distinct, and the constant factors hidden in the $\Theta(n \lg n)$ notation are small. Unlike merge sort, it also has the advantage of sorting in place (see page 158), and it works well even in virtual-memory environments.

Our study of quicksort is broken into four sections. Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we'll start with an intuitive discussion of its performance in Section 7.2 and analyze it precisely at the end of the chapter. Section 7.3 presents a randomized version of quicksort. When all elements are distinct,[1] this randomized algorithm has a good expected running time and no particular input elicits its worst-case behavior. (See Problem 7-2 for the case in which elements may be equal.) Section 7.4 analyzes the randomized algorithm, showing that it runs in $\Theta(n^2)$ time in the worst case and, assuming distinct elements, in expected $O(n \lg n)$ time.

[1] You can enforce the assumption that the values in an array A are distinct at the cost of $\Theta(n)$ additional space and only constant overhead in running time by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if $A[i] < A[j]$ or if $A[i] = A[j]$ and $i < j$. There are also more practical variants of quicksort that work well when elements are not distinct.

7.1 Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer method introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a subarray $A[p:r]$:

Divide by partitioning (rearranging) the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ (the ***low side***) and $A[q+1:r]$ (the ***high side***) such that each element in the low side of the partition is less than or equal to the ***pivot*** $A[q]$, which is, in turn, less than or equal to each element in the high side. Compute the index q of the pivot as part of this partitioning procedure.

Conquer by calling quicksort recursively to sort each of the subarrays $A[p:q-1]$ and $A[q+1:r]$.

Combine by doing nothing: because the two subarrays are already sorted, no work is needed to combine them. All elements in $A[p:q-1]$ are sorted and less than or equal to $A[q]$, and all elements in $A[q+1:r]$ are sorted and greater than or equal to the pivot $A[q]$. The entire subarray $A[p:r]$ cannot help but be sorted!

The QUICKSORT procedure implements quicksort. To sort an entire n-element array $A[1:n]$, the initial call is QUICKSORT$(A, 1, n)$.

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q - 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Partitioning the array

The key to the algorithm is the PARTITION procedure on the next page, which rearranges the subarray $A[p:r]$ in place, returning the index of the dividing point between the two sides of the partition.

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects the element $x = A[r]$ as the pivot. As the procedure runs, each element falls into exactly one of four regions, some of which may be empty. At the start of each iteration of the **for** loop in lines 3–6, the regions satisfy certain properties, shown in Figure 7.2. We state these properties as a loop invariant:

```
PARTITION(A, p, r)
1  $x = A[r]$                          // the pivot
2  $i = p - 1$                         // highest index into the low side
3  for $j = p$ to $r - 1$              // process each element other than the pivot
4      if $A[j] \le x$                 // does this element belong on the low side?
5          $i = i + 1$                     // index of a new slot in the low side
6          exchange $A[i]$ with $A[j]$     // put this element there
7  exchange $A[i + 1]$ with $A[r]$     // pivot goes just to the right of the low side
8  return $i + 1$                      // new index of the pivot
```

At the beginning of each iteration of the loop of lines 3–6, for any array index k, the following conditions hold:

1. if $p \le k \le i$, then $A[k] \le x$ (the tan region of Figure 7.2);
2. if $i + 1 \le k \le j - 1$, then $A[k] > x$ (the blue region);
3. if $k = r$, then $A[k] = x$ (the yellow region).

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that correctness follows from the invariant when the loop terminates.

Initialization: Prior to the first iteration of the loop, we have $i = p - 1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$: the only action in the loop is to increment j. After j has been incremented, the second condition holds for $A[j - 1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$: the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j - 1] > x$, since the item that was swapped into $A[j - 1]$ is, by the loop invariant, greater than x.

Termination: Since the loop makes exactly $r - p$ iterations, it terminates, whereupon $j = r$. At that point, the unexamined subarray $A[j : r - 1]$ is empty, and every entry in the array belongs to one of the other three sets described by the invariant. Thus, the values in the array have been partitioned into three sets: those less than or equal to x (the low side), those greater than x (the high side), and a singleton set containing x (the pivot).

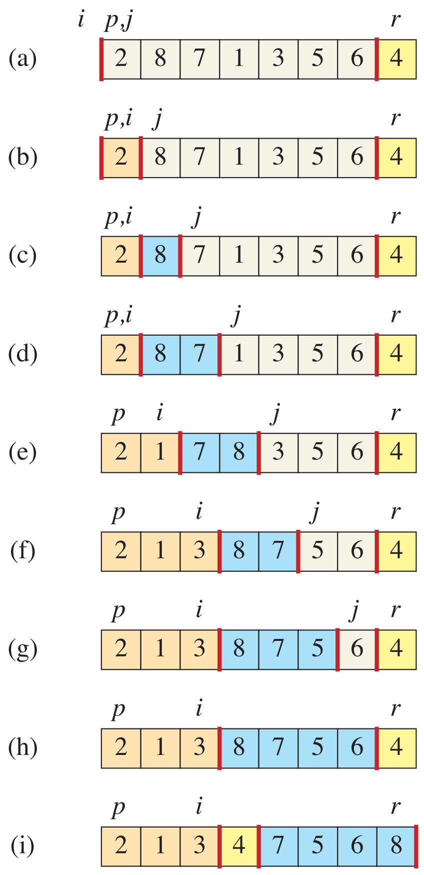


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Tan array elements all belong to the low side of the partition, with values at most x. Blue elements belong to the high side, with values greater than x. White elements have not yet been put into either side of the partition, and the yellow element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed into either side of the partition. **(b)** The value 2 is "swapped with itself" and put into the low side. **(c)–(d)** The values 8 and 7 are placed into to high side. **(e)** The values 1 and 8 are swapped, and the low side grows. **(f)** The values 3 and 7 are swapped, and the low side grows. **(g)–(h)** The high side of the partition grows to include 5 and 6, and the loop terminates. **(i)** Line 7 swaps the pivot element so that it lies between the two sides of the partition, and line 8 returns the pivot's new index.

The final two lines of PARTITION finish up by swapping the pivot with the left-most element greater than x, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index. The output of PARTITION now satisfies the specifications given for the divide step. In fact, it satisfies a slightly stronger condition: after line 3 of QUICKSORT, $A[q]$ is strictly less than every element of $A[q+1:r]$.

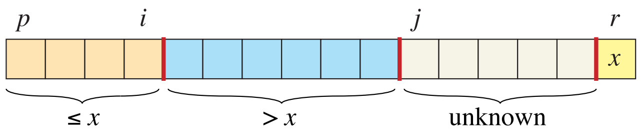


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p:r]$. The tan values in $A[p:i]$ are all less than or equal to x, the blue values in $A[i+1:j-1]$ are all greater than x, the white values in $A[j:r-1]$ have unknown relationships to x, and $A[r] = x$.

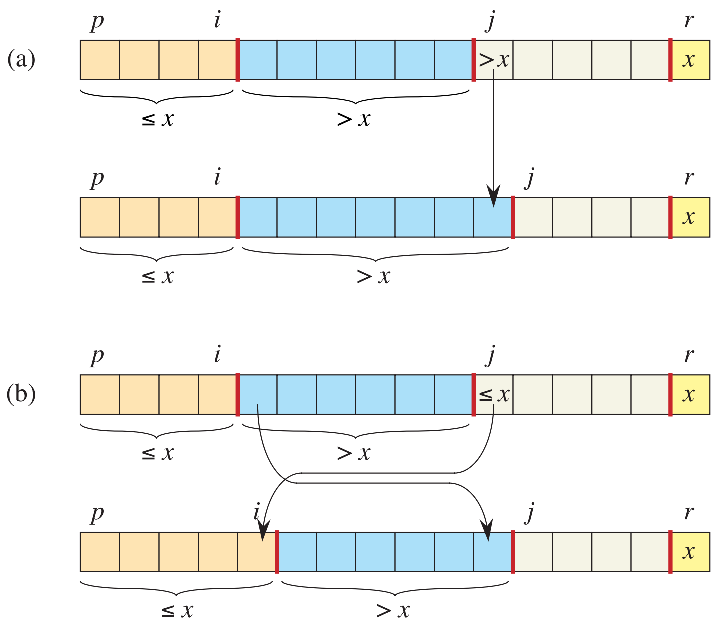


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j]>x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Exercise 7.1-3 asks you to show that the running time of PARTITION on a subarray $A[p:r]$ of $n = r - p + 1$ elements is $\Theta(n)$.

Exercises

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

7.1-2
What value of q does PARTITION return when all elements in the subarray $A[p:r]$ have the same value? Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the subarray $A[p:r]$ have the same value.

7.1-3
Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

7.1-4
Modify QUICKSORT to sort into monotonically decreasing order.

7.2 Performance of quicksort

The running time of quicksort depends on how balanced each partitioning is, which in turn depends on which elements are used as pivots. If the two sides of a partition are about the same size—the partitioning is balanced—then the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. To allow you to gain some intuition before diving into a formal analysis, this section informally investigates how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

But first, let's briefly look at the maximum amount of memory that quicksort requires. Although quicksort sorts in place according to the definition on page 158, the amount of memory it uses—aside from the array being sorted—is not constant. Since each recursive call requires a constant amount of space on the runtime stack, outside of the array being sorted, quicksort requires space proportional to the maximum depth of the recursion. As we'll see now, that could be as bad as $\Theta(n)$ in the worst case.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 0 elements. (See Section 7.4.1.) Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns without doing anything, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

By summing the costs incurred at each level of the recursion, we obtain an arithmetic series (equation (A.3) on page 1141), which evaluates to $\Theta(n^2)$. Indeed, the substitution method can be used to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. The worst-case running time of quicksort is therefore no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor (n-1)/2 \rfloor \le n/2$ and one of size $\lceil (n-1)/2 \rceil - 1 \le n/2$. In this case, quicksort runs much faster. An upper bound on the running time can then be described by the recurrence

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

By case 2 of the master theorem (Theorem 4.1 on page 102), this recurrence has the solution $T(n) = \Theta(n \lg n)$. Thus, if the partitioning is equally balanced at every level of the recursion, an asymptotically faster algorithm results.

Balanced partitioning

As the analyses in Section 7.4 will show, the average-case running time of quicksort is much closer to the best case than to the worst case. By appreciating how the balance of the partitioning affects the recurrence describing the running time, we can gain an understanding of why.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) ,$$

on the running time of quicksort. Figure 7.4 shows the recursion tree for this recurrence, where for simplicity the $\Theta(n)$ driving function has been replaced by n, which won't affect the asymptotic solution of the recurrence (as Exercise 4.7-1 on page 118 justifies). Every level of the tree has cost n, until the recursion bottoms out in a base case at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost

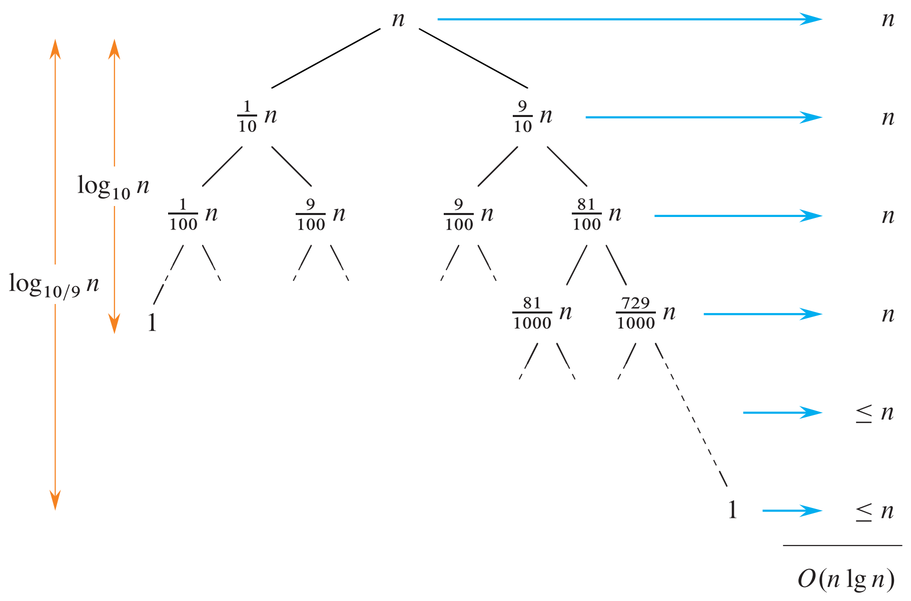


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right.

at most n. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems highly unbalanced, quicksort runs in $O(n \lg n)$ time—asymptotically the same as if the split were right down the middle. Indeed, even a 99-to-1 split yields an $O(n \lg n)$ running time. In fact, any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality. The ratio of the split affects only the constant hidden in the O-notation.

Intuition for the average case

To develop a clear notion of the expected behavior of quicksort, we must assume something about how its inputs are distributed. Because quicksort determines the sorted order using only comparisons between input elements, its behavior depends on the relative ordering of the values in the array elements given as the input, not on the particular values in the array. As in the probabilistic analysis of the hiring problem in Section 5.2, assume that all permutations of the input numbers are equally likely and that the elements are distinct.

When quicksort runs on a random input array, the partitioning is highly unlikely to happen in the same way at every level, as our informal analysis has assumed.

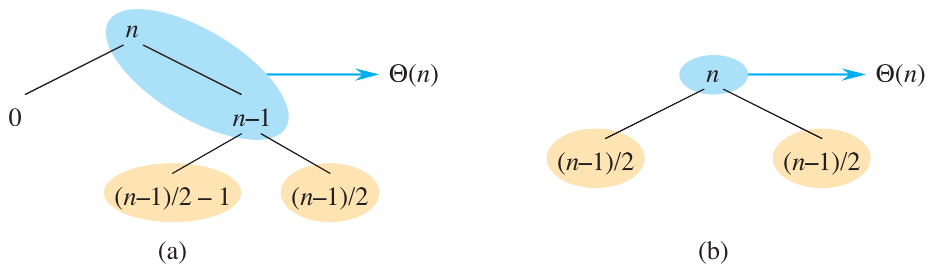


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. **(b)** A single level of a recursion tree that is well balanced. In both parts, the partitioning cost for the subproblems shown with blue shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with tan shading, are no larger than the corresponding subproblems remaining to be solved in (b).

We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show that about 80% of the time PARTITION produces a split that is at least as balanced as 9 to 1, and about 20% of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose for the sake of intuition that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is n for partitioning, and the subarrays produced have sizes $n - 1$ and 0: the worst case. At the next level, the subarray of size $n - 1$ undergoes best-case partitioning into subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. Let's assume that the base-case cost is 1 for the subarray of size 0.

The combination of the bad split followed by the good split produces three subarrays of sizes 0, $(n - 1)/2 - 1$, and $(n - 1)/2$ at a combined partitioning cost of $\Theta(n) + \Theta(n - 1) = \Theta(n)$. This situation is at most a constant factor worse than that in Figure 7.5(b), namely, where a single level of partitioning produces two subarrays of size $(n - 1)/2$, at a cost of $\Theta(n)$. Yet this latter situation is balanced! Intuitively, the $\Theta(n - 1)$ cost of the bad split in Figure 7.5(a) can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the O-notation. We'll analyze the expected running time of a randomized version of quicksort rigorously in Section 7.4.2.

Exercises

7.2-1
Use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

7.2-2
What is the running time of QUICKSORT when all elements of array A have the same value?

7.2-3
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

7.2-4
Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Explain persuasively why the procedure INSERTION-SORT might tend to beat the procedure QUICKSORT on this problem.

7.2-5
Suppose that the splits at every level of quicksort are in the constant proportion α to β, where $\alpha + \beta = 1$ and $0 < \alpha \leq \beta < 1$. Show that the minimum depth of a leaf in the recursion tree is approximately $\log_{1/\alpha} n$ and that the maximum depth is approximately $\log_{1/\beta} n$. (Don't worry about integer round-off.)

7.2-6
Consider an array with distinct elements and for which all permutations of the elements are equally likely. Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that PARTITION produces a split at least as balanced as $1 - \alpha$ to α.

7.3 A randomized version of quicksort

In exploring the average-case behavior of quicksort, we have assumed that all permutations of the input numbers are equally likely. This assumption does not always hold, however, as, for example, in the situation laid out in the premise for

Exercise 7.2-4. Section 5.3 showed that judicious randomization can sometimes be added to an algorithm to obtain good expected performance over all inputs. For quicksort, randomization yields a fast and practical algorithm. Many software libraries provide a randomized version of quicksort as their algorithm of choice for sorting large data sets.

In Section 5.3, the RANDOMIZED-HIRE-ASSISTANT procedure explicitly permutes its input and then runs the deterministic HIRE-ASSISTANT procedure. We could do the same for quicksort as well, but a different randomization technique yields a simpler analysis. Instead of always using $A[r]$ as the pivot, a randomized version randomly chooses the pivot from the subarray $A[p:r]$, where each element in $A[p:r]$ has an equal probability of being chosen. It then exchanges that element with $A[r]$ before partitioning. Because the pivot is chosen randomly, we expect the split of the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. The new partitioning procedure, RANDOMIZED-PARTITION, simply swaps before performing the partitioning. The new quicksort procedure, RANDOMIZED-QUICKSORT, calls RANDOMIZED-PARTITION instead of PARTITION. We'll analyze this algorithm in the next section.

```
RANDOMIZED-PARTITION($A, p, r$)
1  $i$ = RANDOM($p, r$)
2  exchange $A[r]$ with $A[i]$
3  return PARTITION($A, p, r$)

RANDOMIZED-QUICKSORT($A, p, r$)
1  if $p < r$
2      $q$ = RANDOMIZED-PARTITION($A, p, r$)
3      RANDOMIZED-QUICKSORT($A, p, q - 1$)
4      RANDOMIZED-QUICKSORT($A, q + 1, r$)
```

Exercises

7.3-1

Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

7.3-2

When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

7.4 Analysis of quicksort

Section 7.2 gave some intuition for the worst-case behavior of quicksort and for why we expect the algorithm to run quickly. This section analyzes the behavior of quicksort more rigorously. We begin with a worst-case analysis, which applies to either QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an analysis of the expected running time of RANDOMIZED-QUICKSORT.

7.4.1 Worst-case analysis

We saw in Section 7.2 that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

We'll use the substitution method (see Section 4.3) to show that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. Because the procedure PARTITION produces two subproblems with total size $n-1$, we obtain the recurrence

$$T(n) = \max\{T(q) + T(n-1-q) : 0 \le q \le n-1\} + \Theta(n) , \tag{7.1}$$

We guess that $T(n) \le cn^2$ for some constant $c > 0$. Substituting this guess into recurrence (7.1) yields

$$\begin{aligned} T(n) &\le \max\left\{cq^2 + c(n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n) \\ &= c \cdot \max\left\{q^2 + (n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n) . \end{aligned}$$

Let's focus our attention on the maximization. For $q = 0, 1, \ldots, n-1$, we have

$$\begin{aligned} q^2 + (n-1-q)^2 &= q^2 + (n-1)^2 - 2q(n-1) + q^2 \\ &= (n-1)^2 + 2q(q-(n-1)) \\ &\le (n-1)^2 \end{aligned}$$

because $q \le n-1$ implies that $2q(q-(n-1)) \le 0$. Thus every term in the maximization is bounded by $(n-1)^2$.

Continuing with our analysis of $T(n)$, we obtain

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

by picking the constant c large enough that the $c(2n-1)$ term dominates the $\Theta(n)$ term. Thus $T(n) = O(n^2)$. Section 7.2 showed a specific case where quicksort takes $\Omega(n^2)$ time: when partitioning is maximally unbalanced. Thus, the worst-case running time of quicksort is $\Theta(n^2)$.

7.4.2 Expected running time

We have already seen the intuition behind why the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$: if, in each level of recursion, the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$ and $O(n)$ work is performed at each level. Even if we add a few new levels with the most unbalanced split possible between these levels, the total time remains $O(n \lg n)$. We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates and then using this understanding to derive an $O(n \lg n)$ bound on the expected running time. This upper bound on the expected running time, combined with the $\Theta(n \lg n)$ best-case bound we saw in Section 7.2, yields a $\Theta(n \lg n)$ expected running time. We assume throughout that the values of the elements being sorted are distinct.

Running time and comparisons

The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how they select pivot elements. They are the same in all other respects. We can therefore analyze RANDOMIZED-QUICKSORT by considering the QUICKSORT and PARTITION procedures, but with the assumption that pivot elements are selected randomly from the subarray passed to RANDOMIZED-PARTITION. Let's start by relating the asymptotic running time of QUICKSORT to the number of times elements are compared (all in line 4 of PARTITION), understanding that this analysis also applies to RANDOMIZED-QUICKSORT. Note that we are counting the number of times that *array elements* are compared, not comparisons of indices.

Lemma 7.1
The running time of QUICKSORT on an n-element array is $O(n + X)$, where X is the number of element comparisons performed.

Proof The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure. Each time PARTITION is called, it selects a pivot element, which is never included in any future recursive calls to QUICKSORT and PARTITION. Thus, there can be at most n calls to PARTITION over the entire execution of the quicksort algorithm. Each time QUICKSORT calls PARTITION, it also recursively calls itself twice, so there are at most $2n$ calls to the QUICKSORT procedure itself.

One call to PARTITION takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6. Each iteration of this **for** loop performs one comparison in line 4, comparing the pivot element to another element of the array A. Therefore, the total time spent in the **for** loop across all executions is proportional to X. Since there are at most n calls to PARTITION and the time spent outside the **for** loop is $O(1)$ for each call, the total time spent in PARTITION outside of the **for** loop is $O(n)$. Thus the total time for quicksort is $O(n + X)$. ■

Our goal for analyzing RANDOMIZED-QUICKSORT, therefore, is to compute the expected value $\mathrm{E}[X]$ of the random variable X denoting the total number of comparisons performed in all calls to PARTITION. To do so, we must understand when the quicksort algorithm compares two elements of the array and when it does not. For ease of analysis, let's index the elements of the array A by their position in the sorted output, rather than their position in the input. That is, although the elements in A may start out in any order, we'll refer to them by $z_1, z_2, \ldots, z_n$, where $z_1 < z_2 < \cdots < z_n$, with strict inequality because we assume that all elements are distinct. We denote the set $\{z_i, z_{i+1}, \ldots, z_j\}$ by Z_{ij}.

The next lemma characterizes when two elements are compared.

Lemma 7.2
During the execution of RANDOMIZED-QUICKSORT on an array of n distinct elements $z_1 < z_2 < \cdots < z_n$, an element z_i is compared with an element z_j, where $i < j$, if and only if one of them is chosen as a pivot before any other element in the set Z_{ij}. Moreover, no two elements are ever compared twice.

Proof Let's look at the first time that an element $x \in Z_{ij}$ is chosen as a pivot during the execution of the algorithm. There are three cases to consider. If x is neither z_i nor z_j—that is, $z_i < x < z_j$—then z_i and z_j are not compared at any subsequent time, because they fall into different sides of the partition around x. If $x = z_i$, then PARTITION compares z_i with every other item in Z_{ij}. Similarly, if $x = z_j$, then PARTITION compares z_j with every other item in Z_{ij}. Thus, z_i and z_j are compared if and only if the first element to be chosen as a pivot from Z_{ij} is either z_i or z_j. In the latter two cases, where one of z_i and z_j is chosen as a pivot, since the pivot is removed from future comparisons, it is never compared again with the other element. ■

As an example of this lemma, consider an input to quicksort of the numbers 1 through 10 in some arbitrary order. Suppose that the first pivot element is 7. Then the first call to PARTITION separates the numbers into two sets: $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. In the process, the pivot element 7 is compared with all other elements, but no number from the first set (e.g., 2) is or ever will be compared with any number from the second set (e.g., 9). The values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 are never compared because the first pivot element chosen from $Z_{2,9}$ is 7.

The next lemma gives the probability that two elements are compared.

Lemma 7.3
Consider an execution of the procedure RANDOMIZED-QUICKSORT on an array of n distinct elements $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements z_i and z_j where $i < j$, the probability that they are compared is $2/(j - i + 1)$.

Proof Let's look at the tree of recursive calls that RANDOMIZED-QUICKSORT makes, and consider the sets of elements provided as input to each call. Initially, the root set contains all the elements of Z_{ij}, since the root set contains every element in A. The elements belonging to Z_{ij} all stay together for each recursive call of RANDOMIZED-QUICKSORT until PARTITION chooses some element $x \in Z_{ij}$ as a pivot. From that point on, the pivot x appears in no subsequent input set. The first time that RANDOMIZED-SELECT chooses a pivot $x \in Z_{ij}$ from a set containing all the elements of Z_{ij}, each element in Z_{ij} is equally likely to be x because the pivot is chosen uniformly at random. Since $|Z_{ij}| = j - i + 1$, the probability is $1/(j - i + 1)$ that any given element in Z_{ij} is the first pivot chosen from Z_{ij}. Thus, by Lemma 7.2, we have

$$\begin{aligned}
\Pr\{z_i \text{ is compared with } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \frac{2}{j - i + 1},
\end{aligned}$$

where the second line follows from the first because the two events are mutually exclusive. ■

We can now complete the analysis of randomized quicksort.

Theorem 7.4
The expected running time of RANDOMIZED-QUICKSORT on an input of n distinct elements is $O(n \lg n)$.

Proof The analysis uses indicator random variables (see Section 5.2). Let the n distinct elements be $z_1 < z_2 < \cdots < z_n$, and for $1 \le i < j \le n$, define the indicator random variable $X_{ij} = \mathrm{I}\{z_i \text{ is compared with } z_j\}$. From Lemma 7.2, each pair is compared at most once, and so we can express X as follows:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

By taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192) and Lemma 5.1 on page 130, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} && \text{(by Lemma 5.1)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && \text{(by Lemma 7.3) .}
\end{aligned}$$

We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series in equation (A.9) on page 1142:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}$$

This bound and Lemma 7.1 allow us to conclude that the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$ (assuming that the element values are distinct). ■

Exercises

7.4-1
Show that the recurrence

$$T(n) = \max\{T(q) + T(n - q - 1) : 0 \le q \le n - 1\} + \Theta(n)$$

has a lower bound of $T(n) = \Omega(n^2)$.

7.4-2
Show that quicksort's best-case running time is $\Omega(n \lg n)$.

7.4-3
Show that the expression $q^2 + (n - q - 1)^2$ achieves its maximum value over $q = 0, 1, \ldots, n - 1$ when $q = 0$ or $q = n - 1$.

7.4-4
Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

7.4-5
Coarsening the recursion, as we did in Problem 2-1 for merge sort, is a common way to improve the running time of quicksort in practice. We modify the base case of the recursion so that if the array has fewer than k elements, the subarray is sorted by insertion sort, rather than by continued recursive calls to quicksort. Argue that the randomized version of this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should you pick k, both in theory and in practice?

★ ***7.4-6***
Consider modifying the PARTITION procedure by randomly picking three elements from subarray $A[p:r]$ and partitioning about their median (the middle value of the three elements). Approximate the probability of getting worse than an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1/2$.

Problems

7-1 Hoare partition correctness

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partitioning algorithm, which is due to C. A. R. Hoare.

```
HOARE-PARTITION($A, p, r$)
 1  $x = A[p]$
 2  $i = p - 1$
 3  $j = r + 1$
 4  while TRUE
 5      repeat
 6          $j = j - 1$
 7      until $A[j] \le x$
 8      repeat
 9          $i = i + 1$
10      until $A[i] \ge x$
11      if $i < j$
12          exchange $A[i]$ with $A[j]$
13      else return $j$
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and the indices i and j after each iteration of the **while** loop in lines 4–13.

b. Describe how the PARTITION procedure in Section 7.1 differs from HOARE-PARTITION when all elements in $A[p:r]$ are equal. Describe a practical advantage of HOARE-PARTITION over PARTITION for use in quicksort.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p:r]$ contains at least two elements, prove the following:

c. The indices i and j are such that the procedure never accesses an element of A outside the subarray $A[p:r]$.

d. When HOARE-PARTITION terminates, it returns a value j such that $p \le j < r$.

e. Every element of $A[p:j]$ is less than or equal to every element of $A[j+1:r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p:j]$ and $A[j+1:r]$. Since $p \le j < r$, neither partition is empty.

f. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

7-2 Quicksort with equal element values

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. This problem examines what happens when they are not.

a. Suppose that all element values are equal. What is randomized quicksort's running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p:q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1:r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p:r]$ and returns two indices q and t, where $p \le q \le t \le r$, such that

- all elements of $A[q:t]$ are equal,
- each element of $A[p:q-1]$ is less than $A[q]$, and
- each element of $A[t+1:r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

c. Modify the RANDOMIZED-PARTITION procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-PARTITION$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(A, p, r)$ that calls RANDOMIZED-PARTITION$'$ and recurses only on partitions where elements are not known to be equal to each other.

d. Using QUICKSORT$'$, adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct.

7-3 Alternative quicksort analysis

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed. As in the analysis of Section 7.4.2, assume that the values of the elements are distinct.

a. Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this probability to define indicator random variables $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathrm{E}[X_i]$?

b. Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \tag{7.2}$$

c. Show how to rewrite equation (7.2) as

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=1}^{n-1} \mathrm{E}[T(q)] + \Theta(n) . \tag{7.3}$$

d. Show that

$$\sum_{q=1}^{n-1} q \lg q \le \frac{n^2}{2}\lg n - \frac{n^2}{8} \tag{7.4}$$

for $n \ge 2$. (*Hint:* Split the summation into two parts, one summation for $q = 1, 2, \ldots, \lceil n/2 \rceil - 1$ and one summation for $q = \lceil n/2 \rceil, \ldots, n-1$.)

e. Using the bound from equation (7.4), show that the recurrence in equation (7.3) has the solution $\mathrm{E}[T(n)] = O(n \lg n)$. (*Hint:* Show, by substitution, that $\mathrm{E}[T(n)] \le an \lg n$ for sufficiently large n and for some positive constant a.)

7-4 Stooge sort

Professors Howard, Fine, and Howard have proposed a deceptively simple sorting algorithm, named stooge sort in their honor, appearing on the following page.

a. Argue that the call STOOGE-SORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

b. Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (Θ-notation) bound on the worst-case running time.

c. Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

```
STOOGE-SORT($A, p, r$)
1  if $A[p] > A[r]$
2      exchange $A[p]$ with $A[r]$
3  if $p + 1 < r$
4      $k = \lfloor (r - p + 1)/3 \rfloor$      // round down
5      STOOGE-SORT($A, p, r - k$)   // first two-thirds
6      STOOGE-SORT($A, p + k, r$)   // last two-thirds
7      STOOGE-SORT($A, p, r - k$)   // first two-thirds again
```

7-5 Stack depth for quicksort

The QUICKSORT procedure of Section 7.1 makes two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the low side of the partition and then it recursively sorts the high side of the partition. The second recursive call in QUICKSORT is not really necessary, because the procedure can instead use an iterative control structure. This transformation technique, called ***tail-recursion elimination***, is provided automatically by good compilers. Applying tail-recursion elimination transforms QUICKSORT into the TRE-QUICKSORT procedure.

```
TRE-QUICKSORT($A, p, r$)
1  while $p < r$
2      // Partition and then sort the low side.
3      $q$ = PARTITION($A, p, r$)
4      TRE-QUICKSORT($A, p, q - 1$)
5      $p = q + 1$
```

a. Argue that TRE-QUICKSORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is called, its information is ***pushed*** onto the stack, and when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which TRE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c. Modify TRE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

7-6 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. A common approach is the ***median-of-3*** method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, assume that the n elements in the input subarray $A[p:r]$ are distinct and that $n \geq 3$. Denote the sorted version of $A[p:r]$ by $z_1, z_2, \ldots, z_n$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = z_i\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n-1$. (Observe that $p_1 = p_n = 0$.)

b. By what amount does the median-of-3 method increase the likelihood of choosing the pivot to be $x = z_{\lfloor (n+1)/2 \rfloor}$, the median of $A[p:r]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. Suppose that we define a "good" split to mean choosing the pivot as $x = z_i$, where $n/3 \leq i \leq 2n/3$. By what amount does the median-of-3 method increase the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

7-7 Fuzzy sorting of intervals

Consider a sorting problem in which you do not know the numbers exactly. Instead, for each number, you know an interval on the real line to which it belongs. That is, you are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. The goal is to ***fuzzy-sort*** these intervals: to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

b. Argue that your algorithm runs in $\Theta(n \lg n)$ expected time in general, but runs in $\Theta(n)$ expected time when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly, but rather, its performance should naturally improve as the amount of overlap increases.

Chapter notes

Quicksort was invented by Hoare [219], and his version of PARTITION appears in Problem 7-1. Bentley [51, p. 117] attributes the PARTITION procedure given in Section 7.1 to N. Lomuto. The analysis in Section 7.4 based on an analysis due to Motwani and Raghavan [336]. Sedgewick [401] and Bentley [51] provide good references on the details of implementation and how they matter.

McIlroy [323] shows how to engineer a “killer adversary” that produces an array on which virtually any implementation of quicksort takes $\Theta(n^2)$ time.

8 Sorting in Linear Time

We have now seen a handful of algorithms that can sort n numbers in $O(n \lg n)$ time. Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we'll prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time on certain types of inputs. Of course, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

8.1 Lower bounds for sorting

A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, it performs one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order. It may not inspect the values of the elements or gain order information about them in any other way.

Since we are proving a lower bound, we assume without loss of generality in this section that all the input elements are distinct. After all, a lower bound for distinct elements applies when elements may or may not be distinct. Consequently,

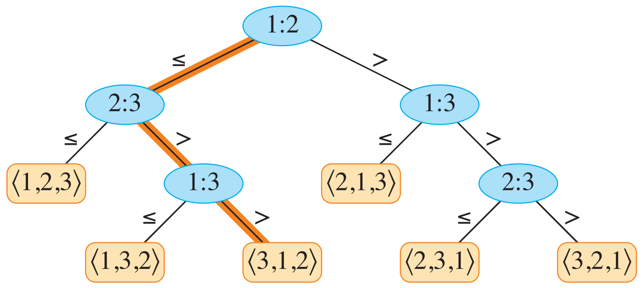


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node (shown in blue) annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The highlighted path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$. Going left from the root node, labeled 1:2, indicates that $a_1 \le a_2$. Going right from the node labeled 2:3 indicates that $a_2 > a_3$. Going right from the node labeled 1:3 indicates that $a_1 > a_3$. Therefore, we have the ordering $a_3 \le a_1 \le a_2$, as indicated in the leaf labeled $\langle 3, 1, 2 \rangle$. Because the three input elements have $3! = 6$ possible permutations, the decision tree must have at least 6 leaves.

comparisons of the form $a_i = a_j$ are useless, which means that we can assume that no comparisons for exact equality occur. Moreover, the comparisons $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \le a_j$.

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A ***decision tree*** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

A decision tree has each internal node annotated by $i:j$ for some i and j in the range $1 \le i, j \le n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$. (See Section C.1 for background on permutations.) Indices in the internal nodes and the leaves always refer to the original positions of the array elements at the start of the sorting algorithm. The execution of the comparison sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \le a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \le a_j$, and the right subtree dictates subsequent comparisons when $a_i > a_j$. Arriving at a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort. (We call such leaves "reachable.") Thus, we consider only decision trees in which each permutation appears as a reachable leaf.

A lower bound for the worst case

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \le l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.28) on page 67)} . \end{aligned}$$ ■

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

Exercises

8.1-1
What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

8.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

8.1-3
Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

8.1-4
You are given an n-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position i such that $i \bmod 4 = 0$ is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions i where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison-based sorting still holds in this case.

8.2 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. It runs in $\Theta(n + k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

The COUNTING-SORT procedure on the facing page takes as input an array $A[1:n]$, the size n of this array, and the limit k on the nonnegative integer values in A. It returns its sorted output in the array $B[1:n]$ and uses an array $C[0:k]$ for temporary working storage.

```
COUNTING-SORT$(A, n, k)$
1   let $B[1:n]$ and $C[0:k]$ be new arrays
2   for $i = 0$ to $k$
3       $C[i] = 0$
4   for $j = 1$ to $n$
5       $C[A[j]] = C[A[j]] + 1$
6   // $C[i]$ now contains the number of elements equal to $i$.
7   for $i = 1$ to $k$
8       $C[i] = C[i] + C[i-1]$
9   // $C[i]$ now contains the number of elements less than or equal to $i$.
10  // Copy $A$ to $B$, starting from the end of $A$.
11  for $j = n$ downto 1
12      $B[C[A[j]]] = A[j]$
13      $C[A[j]] = C[A[j]] - 1$    // to handle duplicate values
14  return $B$
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array C to all zeros, the **for** loop of lines 4–5 makes a pass over the array A to inspect each input element. Each time it finds an input element whose value is i, it increments $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.

Finally, the **for** loop of lines 11–13 makes another pass over A, but in reverse, to place each element $A[j]$ into its correct sorted position in the output array B. If all n elements are distinct, then when line 11 is first entered, for each $A[j]$, the value $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, the loop decrements $C[A[j]]$ each time it places a value $A[j]$ into B. Decrementing $C[A[j]]$ causes the previous element in A with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array B.

How much time does counting sort require? The **for** loop of lines 2–3 takes $\Theta(k)$ time, the **for** loop of lines 4–5 takes $\Theta(n)$ time, the **for** loop of lines 7–8 takes $\Theta(k)$ time, and the **for** loop of lines 11–13 takes $\Theta(n)$ time. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Counting sort can beat the lower bound of $\Omega(n \lg n)$ proved in Section 8.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the

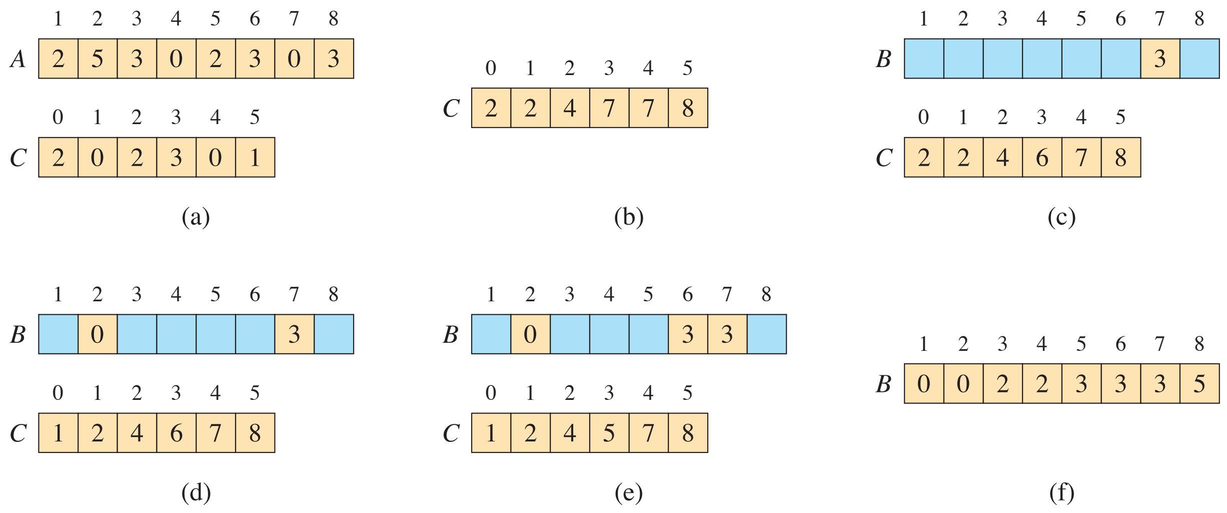


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1:8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 11–13, respectively. Only the tan elements of array B have been filled in. **(f)** The final sorted output array B.

elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

An important property of counting sort is that it is ***stable***: elements with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Exercises

8.2-1

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Prove that COUNTING-SORT is stable.

8.2-3

Suppose that we were to rewrite the **for** loop header in line 11 of the COUNTING-SORT as

```
11  for $j = 1$ to $n$
```

Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

8.2-4

Prove the following loop invariant for COUNTING-SORT:

> At the start of each iteration of the **for** loop of lines 11–13, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

8.2-5

Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of into a new array B.

8.2-6

Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a:b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

8.2-7

Counting sort can also work efficiently if the input values have fractional parts, but the number of digits in the fractional part is small. Suppose that you are given n numbers in the range 0 to k, each with at most d decimal (base 10) digits to the right of the decimal point. Modify counting sort to run in $\Theta(n + 10^d k)$ time.

8.3 Radix sort

Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one

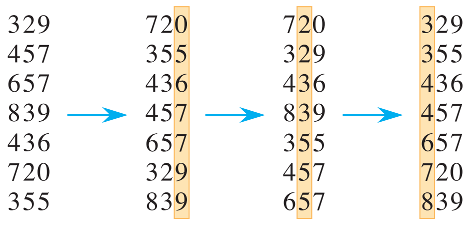


Figure 8.3 The operation of radix sort on seven 3-digit numbers. The leftmost column is the input. The remaining columns show the numbers after successive sorts on increasingly significant digit positions. Tan shading indicates the digit position sorted on to produce each list from the previous one.

of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A d-digit number occupies a field of d columns. Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm.

Intuitively, you might sort numbers on their *most significant* (leftmost) digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-6.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all d digits. Remarkably, at that point the cards are fully sorted on the d-digit number. Thus, only d passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator must be careful not to change the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day (the "least significant" part), next on month, and finally on year.

The code for radix sort is straightforward. The RADIX-SORT procedure assumes that each element in array $A[1:n]$ has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
RADIX-SORT($A, n, d$)
1  for $i = 1$ to $d$
2      use a stable sort to sort array $A[1:n]$ on digit $i$
```

Although the pseudocode for RADIX-SORT does not specify which stable sort to use, COUNTING-SORT is commonly used. If you use COUNTING-SORT as the stable sort, you can make RADIX-SORT a little more efficient by revising COUNTING-SORT to take a pointer to the output array as a parameter, having RADIX-SORT preallocate this array, and alternating input and output between the two arrays in successive iterations of the **for** loop in RADIX-SORT.

Lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

Proof The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to $k - 1$ (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n + k)$ time. There are d passes, and so the total time for radix sort is $\Theta(d(n + k))$. ■

When d is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

Lemma 8.4
Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to k.

Proof For a value $r \leq b$, view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes $\Theta(n + k) = \Theta(n + 2^r)$ time and there are d passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Given n and b, what value of $r \leq b$ minimizes the expression $(b/r)(n + 2^r)$? As r decreases, the factor b/r increases, but as r increases so does 2^r. The answer depends on whether $b < \lfloor \lg n \rfloor$. If $b < \lfloor \lg n \rfloor$, then $r \leq b$ implies $(n + 2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal. If $b \geq \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best running time to within a constant factor, which we can see as follows.[1] Choosing $r = \lfloor \lg n \rfloor$ yields a running time of $\Theta(bn/\lg n)$. As r increases above $\lfloor \lg n \rfloor$, the 2^r term in the numerator increases faster than the r term in the denominator, and so increasing r above $\lfloor \lg n \rfloor$ yields a running time of $\Omega(bn/\lg n)$. If instead r were to decrease below $\lfloor \lg n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If $b = O(\lg n)$, as is often the case, and $r \approx \lg n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's expected running time of $\Theta(n \lg n)$. The constant factors hidden in the Θ-notation differ, however. Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer. Which sorting algorithm to prefer depends on the characteristics of the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort could be the better choice.

Exercises

8.3-1
Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

[1] The choice of $r = \lfloor \lg n \rfloor$ assumes that $n > 1$. If $n \leq 1$, there is nothing to sort.

8.3-2

Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

8.3-3

Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

8.3-4

Suppose that COUNTING-SORT is used as the stable sort within RADIX-SORT. If RADIX-SORT calls COUNTING-SORT d times, then since each call of COUNTING-SORT makes two passes over the data (lines 4–5 and 11–13), altogether $2d$ passes over the data occur. Describe how to reduce the total number of passes to $d + 1$.

8.3-5

Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

★ ***8.3-6***

In the first card-sorting algorithm in this section, which sorts on the most significant digit first, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards does an operator need to keep track of in the worst case?

8.4 Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$. (See Section C.2 for a definition of a uniform distribution.)

Bucket sort divides the interval $[0, 1)$ into n equal-sized subintervals, or ***buckets***, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

The BUCKET-SORT procedure on the next page assumes that the input is an array $A[1:n]$ and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0:n-1]$ of linked lists (buckets) and assumes

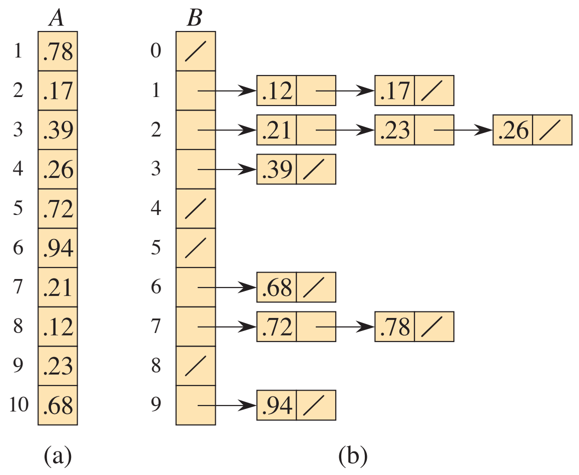


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1:10]$. **(b)** The array $B[0:9]$ of sorted lists (buckets) after line 7 of the algorithm, with slashes indicating the end of each bucket. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation of the lists $B[0], B[1], \ldots, B[9]$ in order.

that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.) Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

```
BUCKET-SORT($A, n$)
1  let $B[0:n-1]$ be a new array
2  **for** $i = 0$ **to** $n - 1$
3      make $B[i]$ an empty list
4  **for** $i = 1$ **to** $n$
5      insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
6  **for** $i = 0$ **to** $n - 1$
7      sort list $B[i]$ with insertion sort
8  concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
9  **return** the concatenated lists
```

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Since $\lfloor n \cdot A[i] \rfloor \le \lfloor n \cdot A[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index. If $A[i]$ and $A[j]$ go into the same bucket, then the **for** loop of lines 6–7 puts them into the proper order. If $A[i]$ and $A[j]$ go into different buckets, then line 8 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that, together, all lines except line 7 take $O(n)$ time in the worst case. We need to analyze the total time taken by the n calls to insertion sort in line 7.

To analyze the cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \; . \tag{8.1}$$

We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.25) on page 1193)} \; . \end{aligned} \tag{8.2}$$

We claim that

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \tag{8.3}$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket i has the same value of $\mathrm{E}[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket.

To prove equation (8.3), view each random variable n_i as the number of successes in n Bernoulli trials (see Section C.4). Success in a trial occurs when an element goes into bucket $B[i]$, with a probability $p = 1/n$ of success and $q = 1 - 1/n$ of failure. A binomial distribution counts n_i, the number of successes, in the n trials. By equations (C.41) and (C.44) on pages 1199–1200, we have $\mathrm{E}[n_i] = np = n(1/n) = 1$ and $\mathrm{Var}[n_i] = npq = 1 - 1/n$. Equation (C.31) on page 1194 gives

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \; , \end{aligned}$$

which proves equation (8.3). Using this expected value in equation (8.2), we get that the average-case running time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort runs in linear time.

Exercises

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4-2

Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

8.4-3

Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $\mathrm{E}[X^2]$? What is $\mathrm{E}^2[X]$?

8.4-4

An array A of size $n > 10$ is filled in the following way. For each element $A[i]$, choose two random variables x_i and y_i uniformly and independently from $[0, 1)$. Then set

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n} .$$

Modify bucket sort so that it sorts the array A in $O(n)$ expected time.

★ ***8.4-5***

You are given n points in the unit disk, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \le 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed, that is, the probability of finding a point in any region of the disk is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit disk.)

★ ***8.4-6***

A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \le x\}$. Suppose that you draw a list of n random variables

$X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time (given y you can find x such that $P(x) = y$ in $O(1)$ time). Give an algorithm that sorts these numbers in linear average-case time.

Problems

8-1 Probabilistic lower bounds on comparison sorting

In this problem, you will prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. You'll begin by examining a deterministic comparison sort A with decision tree T_A. Assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T—the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let *LT* and *RT* be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$. (*Hint:* Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in *LT* and $k - i_0$ the number of leaves in *RT*.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

Now consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form RANDOM$(1, r)$ made by algorithm B. The node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

8-2 Sorting in place in linear time

You have an array of n data records to sort, each with a key of 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n+k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable?

8-3 Sorting variable-length items

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (The desired order is the standard alphabetical order: for example, `a` $<$ `ab` $<$ `b`.)

8-4 Water jugs

You are given n red and n blue water jugs, all of different shapes and sizes. All the red jugs hold different amounts of water, as do all the blue jugs, and you cannot tell from the size of a jug how much water it holds. Moreover, for every jug of one color, there is a jug of the other color that holds the same amount of water.

Your task is to group the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation tells you whether the red jug or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \le A[i+k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show how to sort a k-sorted array of length n in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-11.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to part (e) along with the lower bound on comparison sorts.)

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, you will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items. First, you will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now you will show a slightly tighter $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to part (c) to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

8-7 The 0-1 sorting lemma and columnsort

A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

```
COMPARE-EXCHANGE(A, i, j)
1  if A[i] > A[j]
2      exchange A[i] with A[j]
```

After the compare-exchange operation, we know that $A[i] \leq A[j]$.

An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, the COMPARE-EXCHANGE-INSERTION-SORT procedure on the facing page shows a variation of insertion sort as an oblivious compare-exchange algorithm. (Unlike the INSERTION-SORT procedure on page 19, the oblivious version runs in $\Theta(n^2)$ time in all cases.)

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

```
COMPARE-EXCHANGE-INSERTION-SORT$(A, n)$
1  for $i = 2$ to $n$
2      for $j = i - 1$ downto 1
3          COMPARE-EXCHANGE$(A, j, j + 1)$
```

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1:n]$. Let $A[p]$ be the smallest value in A that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1:n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p] \,, \\ 1 & \text{if } A[i] > A[p] \,. \end{cases}$$

a. Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down each column in turn, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.

10	14	5
8	7	17
12	1	6
16	9	11
4	15	2
18	3	13

(a)

4	1	2
8	3	5
10	7	6
12	9	11
16	14	13
18	15	17

(b)

4	8	10
12	16	18
1	3	7
9	14	15
2	5	6
11	13	17

(c)

1	3	6
2	5	7
4	8	10
9	13	15
11	14	17
12	16	18

(d)

1	4	11
3	8	14
6	10	17
2	9	12
5	13	16
7	15	18

(e)

1	4	11
2	8	12
3	9	14
5	10	16
6	13	17
7	15	18

(f)

	5	10	16
	6	13	17
	7	15	18
1	4	11	
2	8	12	
3	9	14	

(g)

	4	10	16
	5	11	17
	6	12	18
1	7	13	
2	8	14	
3	9	15	

(h)

1	7	13
2	8	14
3	9	15
4	10	16
5	11	17
6	12	18

(i)

Figure 8.5 The steps of columnsort. **(a)** The input array with 6 rows and 3 columns. (This example does not obey the $r \geq 2s^2$ requirement, but it works.) **(b)** After sorting each column in step 1. **(c)** After transposing and reshaping in step 2. **(d)** After sorting each column in step 3. **(e)** After performing step 4, which inverts the permutation from step 2. **(f)** After sorting each column in step 5. **(g)** After shifting by half a column in step 6. **(h)** After sorting each column in step 7. **(i)** After performing step 8, which inverts the permutation from step 6. Steps 6–8 sort the bottom half of each column with the top half of the next column. After step 8, the array is sorted in column-major order.

3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

You can think of steps 6–8 as a single step that sorts the bottom half of each column and the top half of the next column. Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s or if it is empty. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.

d. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most s dirty rows between them. (One of the clean rows could be empty.)

e. Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle. (Again, one of the clean areas could be empty.)

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. (Once again, one of the clean areas could be empty.) How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [150]. Knuth's comprehensive treatise on sorting [261] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Ben-Or [46] studied lower bounds for sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by Isaac and Singleton [235].

Munro and Raman [338] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting n b-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable b-bit words. Fredman and Willard [157] introduced the fusion tree data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n \sqrt{\lg n})$ time by Andersson [17]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [18] have shown how to sort n integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of n. Using multiplicative hashing, we can reduce the storage needed to $O(n)$, but then the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [17], Thorup [434] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and it uses linear space. Combining these techniques with some new ideas, Han [207] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [286].

9 Medians and Order Statistics

The ith ***order statistic*** of a set of n elements is the ith smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the nth order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = (n+1)/2$. When n is even, there are two medians, the ***lower median*** occurring at $i = n/2$ and the ***upper median*** occurring at $i = n/2 + 1$. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n+1)/2 \rfloor$ and $i = \lceil (n+1)/2 \rceil$. For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

This chapter addresses the problem of selecting the ith order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the ***selection problem*** as follows:

Input: A set A of n distinct numbers[1] and an integer i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

We can solve the selection problem in $O(n \lg n)$ time simply by sorting the numbers using heapsort or merge sort and then outputting the ith element in the sorted array. This chapter presents asymptotically faster algorithms.

Section 9.1 examines the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which we investigate in the subsequent two sections. Section 9.2 analyzes a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming dis-

[1] As in the footnote on page 182, you can enforce the assumption that the numbers are distinct by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if either $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$.

tinct elements. Section 9.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? To obtain an upper bound of $n - 1$ comparisons, just examine each element of the set in turn and keep track of the smallest element seen so far. The MINIMUM procedure assumes that the set resides in array $A[1:n]$.

```
MINIMUM(A, n)
min = A[1]
for i = 2 to n
    if min > A[i]
        min = A[i]
return min
```

It's no more difficult to find the maximum with $n - 1$ comparisons.

Is this algorithm for minimum the best we can do? Yes, because it turns out that there's a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. Since every element except the winner must lose at least one match, we can conclude that $n - 1$ comparisons are necessary to determine the minimum. Hence the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

Simultaneous minimum and maximum

Some applications need to find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

Of course, we can determine both the minimum and the maximum of n elements using $\Theta(n)$ comparisons. We simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2 = \Theta(n)$ comparisons.

Although $2n-2$ comparisons is asymptotically optimal, it is possible to improve the leading constant. We can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons. The trick is to maintain both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element, process elements in pairs. Compare pairs of elements from the input first *with each other*, and then compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How you set up initial values for the current minimum and maximum depends on whether n is odd or even. If n is odd, set both the minimum and maximum to the value of the first element, and then process the rest of the elements in pairs. If n is even, perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd n.

Let's count the total number of comparisons. If n is odd, then $3\lfloor n/2 \rfloor$ comparisons occur. If n is even, 1 initial comparison occurs, followed by another $3(n-2)/2$ comparisons, for a total of $3n/2 - 2$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

Exercises

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

9.1-2

Given $n > 2$ distinct numbers, you want to find a number that is neither the minimum nor the maximum. What is the smallest number of comparisons that you need to perform?

9.1-3

A racetrack can run races with five horses at a time to determine their relative speeds. For 25 horses, it takes six races to determine the fastest horse, assuming transitivity (see page 1159). What's the minimum number of races it takes to determine the fastest three horses out of 25?

★ ***9.1-4***

Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

9.2 Selection in expected linear time

The general selection problem—finding the ith order statistic for any value of i—appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. This section presents a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. Like quicksort it partitions the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The RANDOMIZED-SELECT procedure returns the ith smallest element of the array $A[p:r]$, where $1 \le i \le r - p + 1$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6      return A[q]        // the pivot value is the answer
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Figure 9.1 illustrates how the RANDOMIZED-SELECT procedure works. Line 1 checks for the base case of the recursion, in which the subarray $A[p:r]$ consists of just one element. In this case, i must equal 1, and line 2 simply returns $A[p]$ as the ith smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ and $A[q+1:r]$ such that each element of $A[p:q-1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1:r]$. (Although our analysis assumes that the elements are distinct, the procedure still yields the correct result even if equal elements are present.) As in quicksort, we'll refer to $A[q]$ as the ***pivot*** element. Line 4 computes the number k of elements in the subarray $A[p:q]$, that is,

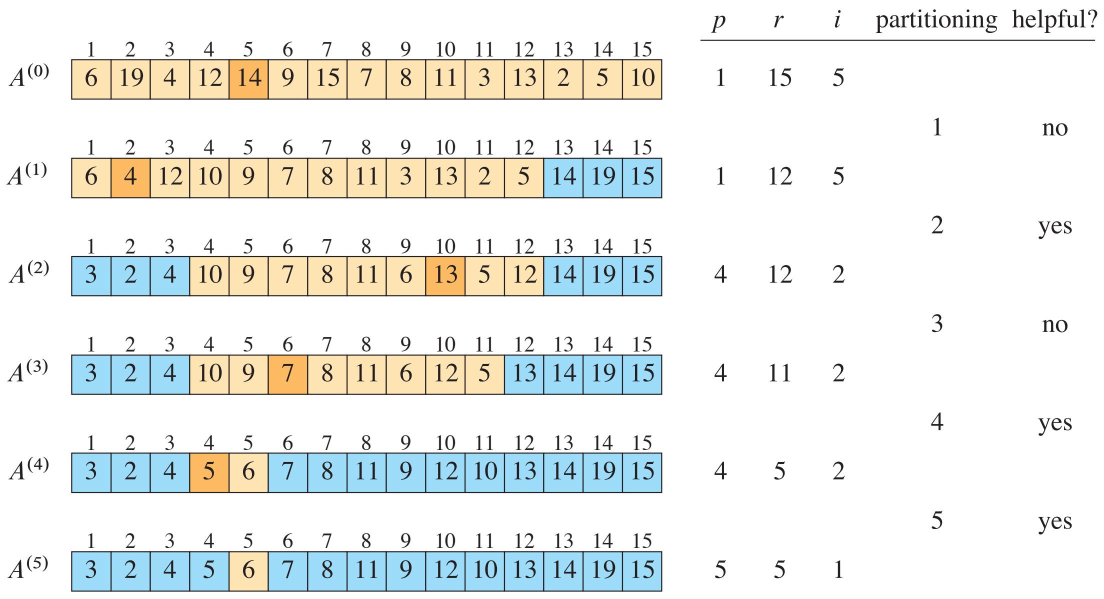

p r i partitioning helpful?
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A^(0) 6 19 4 12 14 9 15 7 8 11 3 13 2 5 10 1 15 5
1 no
A^(1) 6 4 12 10 9 7 8 11 3 13 2 5 14 19 15 1 12 5
2 yes
A^(2) 3 2 4 10 9 7 8 11 6 13 5 12 14 19 15 4 12 2
3 no
A^(3) 3 2 4 10 9 7 8 11 6 12 5 13 14 19 15 4 11 2
4 yes
A^(4) 3 2 4 5 6 7 8 11 9 12 10 13 14 19 15 4 5 2
5 yes
A^(5) 3 2 4 5 6 7 8 11 9 12 10 13 14 19 15 5 5 1

Figure 9.1 The action of RANDOMIZED-SELECT as successive partitionings narrow the subarray $A[p:r]$, showing the values of the parameters p, r, and i at each recursive call. The subarray $A[p:r]$ in each recursive step is shown in tan, with the dark tan element selected as the pivot for the next partitioning. Blue elements are outside $A[p:r]$. The answer is the tan element in the bottom array, where $p = r = 5$ and $i = 1$. The array designations $A^{(0)}, A^{(1)}, \ldots, A^{(5)}$, the partitioning numbers, and whether the partitioning is helpful are explained on the following page.

the number of elements in the low side of the partition, plus 1 for the pivot element. Line 5 then checks whether $A[q]$ is the ith smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p:q-1]$ and $A[q+1:r]$ the ith smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p:r]$—namely, the elements of $A[p:q]$—the desired element is the $(i-k)$th smallest element of $A[q+1:r]$, which line 9 finds recursively. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because it could be extremely unlucky and always partition around the largest remaining element before identifying the ith smallest when only one element remains. In this worst case, each recursive step removes only the pivot from consideration. Because partitioning n elements takes $\Theta(n)$ time, the recurrence for the worst-case running time is the same as for QUICKSORT:

$T(n) = T(n-1) + \Theta(n)$, with the solution $T(n) = \Theta(n^2)$. We'll see that the algorithm has a linear expected running time, however, and because it is randomized, no particular input elicits the worst-case behavior.

To see the intuition behind the linear expected running time, suppose that each time the algorithm randomly selects a pivot element, the pivot lies somewhere within the second and third quartiles—the "middle half"—of the remaining elements in sorted order. If the ith smallest element is less than the pivot, then all the elements greater than the pivot are ignored in all future recursive calls. These ignored elements include at least the uppermost quartile, and possibly more. Likewise, if the ith smallest element is greater than the pivot, then all the elements less than the pivot—at least the first quartile—are ignored in all future recursive calls. Either way, therefore, at least $1/4$ of the remaining elements are ignored in all future recursive calls, leaving at most $3/4$ of the remaining elements ***in play***: residing in the subarray $A[p:r]$. Since RANDOMIZED-PARTITION takes $\Theta(n)$ time on a subarray of n elements, the recurrence for the worst-case running time is $T(n) = T(3n/4) + \Theta(n)$. By case 3 of the master method (Theorem 4.1 on page 102), this recurrence has solution $T(n) = \Theta(n)$.

Of course, the pivot does not necessarily fall into the middle half every time. Since the pivot is selected at random, the probability that it falls into the middle half is about $1/2$ each time. We can view the process of selecting the pivot as a Bernoulli trial (see Section C.4) with success equating to the pivot residing in the middle half. Thus the expected number of trials needed for success is given by a geometric distribution: just two trials on average (equation (C.36) on page 1197). In other words, we expect that half of the partitionings reduce the number of elements still in play by at least $3/4$ and that half of the partitionings do not help as much. Consequently, the expected number of partitionings at most doubles from the case when the pivot always falls into the middle half. The cost of each extra partitioning is less than the one that preceded it, so that the expected running time is still $\Theta(n)$.

To make the above argument rigorous, we start by defining the random variable $A^{(j)}$ as the set of elements of A that are still in play after j partitionings (that is, within the subarray $A[p:r]$ after j calls of RANDOMIZED-SELECT), so that $A^{(0)}$ consists of all the elements in A. Since each partitioning removes at least one element—the pivot—from being in play, the sequence $|A^{(0)}|, |A^{(1)}|, |A^{(2)}|, \ldots$ strictly decreases. Set $A^{(j-1)}$ is in play before the jth partitioning, and set $A^{(j)}$ remains in play afterward. For convenience, assume that the initial set $A^{(0)}$ is the result of a 0th "dummy" partitioning.

Let's call the jth partitioning ***helpful*** if $|A^{(j)}| \le (3/4)|A^{(j-1)}|$. Figure 9.1 shows the sets $A^{(j)}$ and whether partitionings are helpful for an example array. A helpful partitioning corresponds to a successful Bernoulli trial. The following lemma shows that a partitioning is at least as likely to be helpful as not.

Lemma 9.1
A partitioning is helpful with probability at least $1/2$.

Proof Whether a partitioning is helpful depends on the randomly chosen pivot. We discussed the "middle half" in the informal argument above. Let's more precisely define the middle half of an n-element subarray as all but the smallest $\lceil n/4 \rceil - 1$ and greatest $\lceil n/4 \rceil - 1$ elements (that is, all but the first $\lceil n/4 \rceil - 1$ and last $\lceil n/4 \rceil - 1$ elements if the subarray were sorted). We'll prove that if the pivot falls into the middle half, then the pivot leads to a helpful partitioning, and we'll also prove that the probability of the pivot falling into the middle half is at least $1/2$.

Regardless of where the pivot falls, either all the elements greater than it or all the elements less than it, along with the pivot itself, will no longer be in play after partitioning. If the pivot falls into the middle half, therefore, at least $\lceil n/4 \rceil - 1$ elements less than the pivot or $\lceil n/4 \rceil - 1$ elements greater than the pivot, plus the pivot, will no longer be in play after partitioning. That is, at least $\lceil n/4 \rceil$ elements will no longer be in play. The number of elements remaining in play will be at most $n - \lceil n/4 \rceil$, which equals $\lfloor 3n/4 \rfloor$ by Exercise 3.3-2 on page 70. Since $\lfloor 3n/4 \rfloor \le 3n/4$, the partitioning is helpful.

To determine a lower bound on the probability that a randomly chosen pivot falls into the middle half, we determine an upper bound on the probability that it does not. That probability is

$$\begin{aligned} \frac{2(\lceil n/4 \rceil - 1)}{n} &\le \frac{2((n/4 + 1) - 1)}{n} && \text{(by inequality (3.2) on page 64)} \\ &= \frac{n/2}{n} \\ &= 1/2 . \end{aligned}$$

Thus, the pivot has a probability of at least $1/2$ of falling into the middle half, and so the probability is at least $1/2$ that a partitioning is helpful. ■

We can now bound the expected running time of RANDOMIZED-SELECT.

Theorem 9.2
The procedure RANDOMIZED-SELECT on an input array of n distinct elements has an expected running time of $\Theta(n)$.

Proof Since not every partitioning is necessarily helpful, let's give each partitioning an index starting at 0 and denote by $\langle h_0, h_1, h_2, \ldots, h_m \rangle$ the sequence of partitionings that are helpful, so that the h_kth partitioning is helpful for $k = 0, 1, 2, \ldots, m$. Although the number m of helpful partitionings is a random vari-

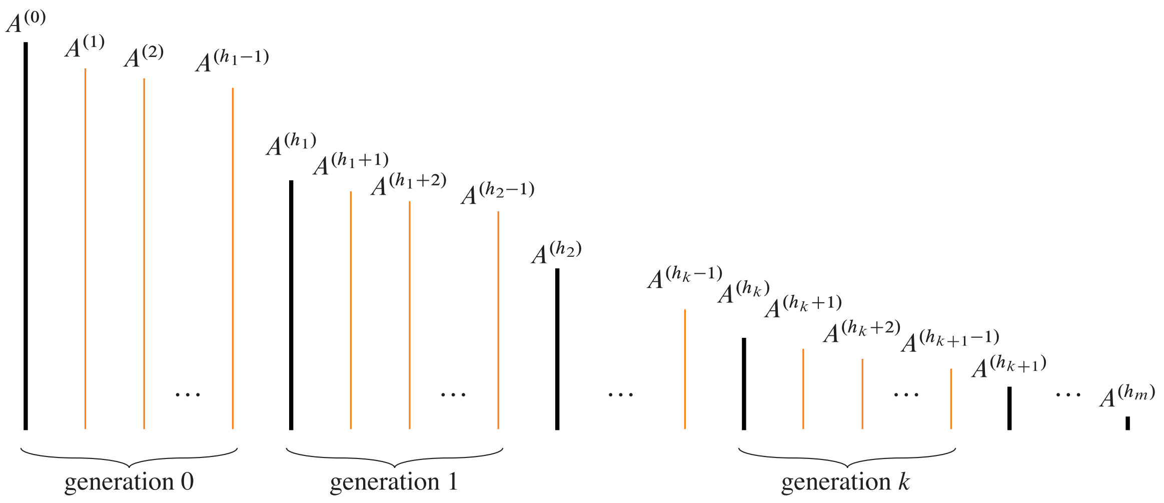


Figure 9.2 The sets within each generation in the proof of Theorem 9.2. Vertical lines represent the sets, with the height of each line indicating the size of the set, which equals the number of elements in play. Each generation starts with a set $A^{(h_k)}$, which is the result of a helpful partitioning. These sets are drawn in black and are at most $3/4$ the size of the sets to their immediate left. Sets drawn in orange are not the first within a generation. A generation may contain just one set. The sets in generation k are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_{k+1}-1)}$. The sets $A^{(h_k)}$ are defined so that $|A^{(h_k)}| \le (3/4)|A^{(h_k-1)}|$. If the partitioning gets all the way to generation h_m, set $A^{(h_m)}$ has at most one element in play.

able, we can bound it, since after at most $\lceil \log_{4/3} n \rceil$ helpful partitionings, only one element remains in play. Consider the dummy 0th partitioning as helpful, so that $h_0 = 0$. Denote $|A^{(h_k)}|$ by n_k, where $n_0 = |A^{(0)}|$ is the original problem size. Since the h_kth partitioning is helpful and the sizes of the sets $A^{(j)}$ strictly decrease, we have $n_k = |A^{(h_k)}| \le (3/4)|A^{(h_k-1)}| = (3/4)\, n_{k-1}$ for $k = 1, 2, \ldots, m$. By iterating $n_k \le (3/4)\, n_{k-1}$, we have that $n_k \le (3/4)^k n_0$ for $k = 0, 1, 2, \ldots, m$.

As Figure 9.2 depicts, we break up the sequence of sets $A^{(j)}$ into m ***generations*** consisting of consecutively partitioned sets, starting with the result $A^{(h_k)}$ of a helpful partitioning and ending with the last set $A^{(h_{k+1}-1)}$ before the next helpful partitioning, so that the sets in generation k are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_{k+1}-1)}$. Then for each set of elements $A^{(j)}$in the kth generation, we have that $|A^{(j)}| \le |A^{(h_k)}| = n_k \le (3/4)^k n_0$.

Next, we define the random variable

$$X_k = h_{k+1} - h_k$$

for $k = 0, 1, 2, \ldots, m-1$. That is, X_k is the number of sets in the kth generation, so that the sets in the kth generation are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_k+X_k-1)}$.

By Lemma 9.1, the probability that a partitioning is helpful is at least $1/2$. The probability is actually even higher, since a partitioning is helpful even if the pivot does not fall into the middle half but the ith smallest element happens to lie in the smaller side of the partitioning. We'll just use the lower bound of $1/2$, however, and then equation (C.36) gives that $\text{E}[X_k] \le 2$ for $k = 0, 1, 2, \ldots, m-1$.

Let's derive an upper bound on how many comparisons are made altogether during partitioning, since the running time is dominated by the comparisons. Since we are calculating an upper bound, assume that the recursion goes all the way until only one element remains in play. The jth partitioning takes the set $A^{(j-1)}$ of elements in play, and it compares the randomly chosen pivot with all the other $|A^{(j-1)}| - 1$ elements, so that the jth partitioning makes fewer than $|A^{(j-1)}|$ comparisons. The sets in the kth generation have sizes $|A^{(h_k)}|, |A^{(h_k+1)}|, \ldots, |A^{(h_k+X_k-1)}|$. Thus, the total number of comparisons during partitioning is less than

$$\begin{aligned}
\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\le \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\
&= \sum_{k=0}^{m-1} X_k \, |A^{(h_k)}| \\
&\le \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \,.
\end{aligned}$$

Since $\text{E}[X_k] \le 2$, we have that the expected total number of comparisons during partitioning is less than

$$\begin{aligned}
\text{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1} \text{E}\left[X_k \left(\frac{3}{4}\right)^k n_0\right] && \text{(by linearity of expectation)} \\
&= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \text{E}[X_k] \\
&\le 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0 && \text{(by equation (A.7) on page 1142)} \,.
\end{aligned}$$

Since n_0 is the size of the original array A, we conclude that the expected number of comparisons, and thus the expected running time, for RANDOMIZED-SELECT is $O(n)$. All n elements are examined in the first call of RANDOMIZED-

PARTITION, giving a lower bound of $\Omega(n)$. Hence the expected running time is $\Theta(n)$. ■

Exercises

9.2-1

Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

9.2-2

Write an iterative version of RANDOMIZED-SELECT.

9.2-3

Suppose that RANDOMIZED-SELECT is used to select the minimum element of the array $A = \langle 2, 3, 0, 5, 7, 9, 1, 8, 6, 4 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

9.2-4

Argue that the expected running time of RANDOMIZED-SELECT does not depend on the order of the elements in its input array $A[p:r]$. That is, the expected running time is the same for any permutation of the input array $A[p:r]$. (*Hint:* Argue by induction on the length n of the input array.)

9.3 Selection in worst-case linear time

We'll now examine a remarkable and theoretically interesting selection algorithm whose running time is $\Theta(n)$ in the worst case. Although the RANDOMIZED-SELECT algorithm from Section 9.2 achieves linear expected time, we saw that its running time in the worst case was quadratic. The selection algorithm presented in this section achieves linear time in the worst case, but it is not nearly as practical as RANDOMIZED-SELECT. It is mostly of theoretical interest.

Like the expected linear-time RANDOMIZED-SELECT, the worst-case linear-time algorithm SELECT finds the desired element by recursively partitioning the input array. Unlike RANDOMIZED-SELECT, however, SELECT *guarantees* a good split by choosing a provably good pivot when partitioning the array. The cleverness in the algorithm is that it finds the pivot recursively. Thus, there are two invocations of SELECT: one to find a good pivot, and a second to recursively find the desired order statistic.

The partitioning algorithm used by SELECT is like the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), but modified to take the element to partition around as an additional input parameter. Like PARTITION, the

PARTITION-AROUND algorithm returns the index of the pivot. Since it's so similar to PARTITION, the pseudocode for PARTITION-AROUND is omitted.

The SELECT procedure takes as input a subarray $A[p:r]$ of $n = r - p + 1$ elements and an integer i in the range $1 \le i \le n$. It returns the ith smallest element of A. The pseudocode is actually more understandable than it might appear at first.

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                  // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                       // number of 5-element groups
12  for j = p to p + g − 1                  // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)        // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                         // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

The pseudocode starts by executing the **while** loop in lines 1–10 to reduce the number $r - p + 1$ of elements in the subarray until it is divisible by 5. The **while** loop executes 0 to 4 times, each time rearranging the elements of $A[p:r]$ so that $A[p]$ contains the minimum element. If $i = 1$, which means that we actually want the minimum element, then the procedure simply returns it in line 7. Otherwise, SELECT eliminates the minimum from the subarray $A[p:r]$ and iterates to find the $(i-1)$st element in $A[p+1:r]$. Lines 9–10 do so by incrementing p and decrementing i. If the **while** loop completes all of its iterations without returning a

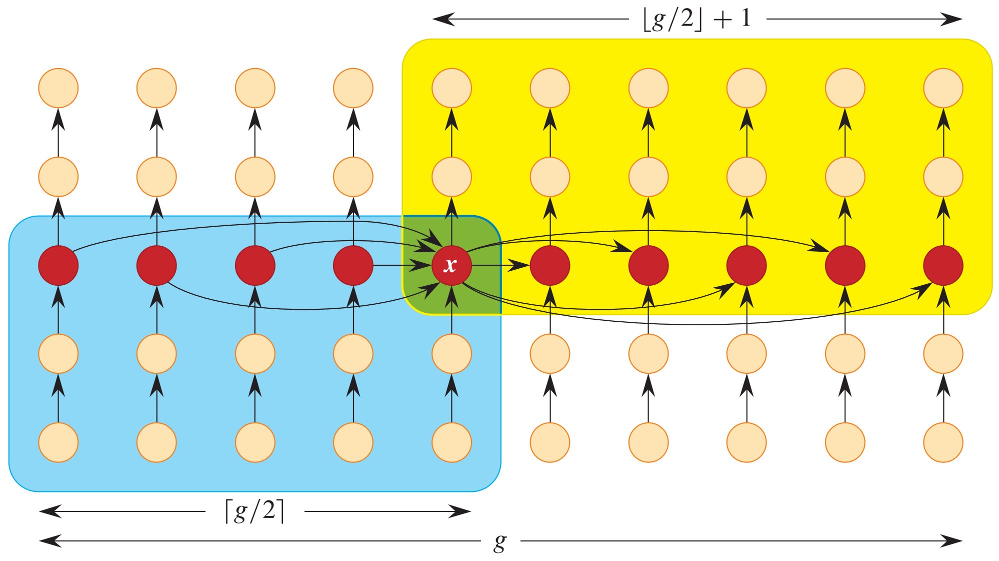


Figure 9.3 The relationships between elements (shown as circles) immediately after line 17 of the selection algorithm SELECT. There are $g = (r - p + 1)/5$ groups of 5 elements, each of which occupies a column. For example, the leftmost column contains elements $A[p]$, $A[p + g]$, $A[p + 2g]$, $A[p + 3g]$, $A[p + 4g]$, and the next column contains $A[p + 1]$, $A[p + g + 1]$, $A[p + 2g + 1]$, $A[p + 3g + 1]$, $A[p + 4g + 1]$. The medians of the groups are red, and the pivot x is labeled. Arrows go from smaller elements to larger. The elements on the blue background are all known to be less than or equal to x and cannot fall into the high side of the partition around x. The elements on the yellow background are known to be greater than or equal to x and cannot fall into the low side of the partition around x. The pivot x belongs to both the blue and yellow regions and is shown on a green background. The elements on the white background could lie on either side of the partition.

result, the procedure executes the core of the algorithm in lines 11–24, assured that the number $r - p + 1$ of elements in $A[p:r]$ is evenly divisible by 5.

The next part of the algorithm implements the following idea, illustrated in Figure 9.3. Divide the elements in $A[p:r]$ into $g = (r-p+1)/5$ groups of 5 elements each. The first 5-element group is

$$\langle A[p], A[p + g], A[p + 2g], A[p + 3g], A[p + 4g] \rangle ,$$

the second is

$$\langle A[p + 1], A[p + g + 1], A[p + 2g + 1], A[p + 3g + 1], A[p + 4g + 1] \rangle ,$$

and so forth until the last, which is

$$\langle A[p + g - 1], A[p + 2g - 1], A[p + 3g - 1], A[p + 4g - 1], A[r] \rangle .$$

(Note that $r = p + 5g - 1$.) Line 13 puts each group in order using, for example, insertion sort (Section 2.1), so that for $j = p, p + 1, \ldots, p + g - 1$, we have

$$A[j] \le A[j+g] \le A[j+2g] \le A[j+3g] \le A[j+4g] .$$

Each vertical column in Figure 9.3 depicts a sorted group of 5 elements. The median of each 5-element group is $A[j+2g]$, and thus all the 5-element medians, shown in red, lie in the range $A[p+2g : p+3g-1]$.

Next, line 16 determines the pivot x by recursively calling SELECT to find the median (specifically, the $\lceil g/2 \rceil$th smallest) of the g group medians. Line 17 uses the modified PARTITION-AROUND algorithm to partition the elements of $A[p:r]$ around x, returning the index q of x, so that $A[q] = x$, elements in $A[p:q]$ are all at most x, and elements in $A[q:r]$ are greater than or equal to x.

The remainder of the code mirrors that of RANDOMIZED-SELECT. If the pivot x is the ith largest, the procedure returns it. Otherwise, the procedure recursively calls itself on either $A[p:q-1]$ or $A[q+1:r]$, depending on the value of i.

Let's analyze the running time of SELECT and see how the judicious choice of the pivot x plays into a guarantee on its worst-case running time.

Theorem 9.3
The running time of SELECT on an input of n elements is $\Theta(n)$.

Proof Define $T(n)$ as the worst-case time to run SELECT on any input subarray $A[p:r]$ of size at most n, that is, for which $r-p+1 \le n$. By this definition, $T(n)$ is monotonically increasing.

We first determine an upper bound on the time spent outside the recursive calls in lines 16, 23, and 24. The **while** loop in lines 1–10 executes 0 to 4 times, which is $O(1)$ times. Since the dominant time within the loop is the computation of the minimum in lines 2–4, which takes $\Theta(n)$ time, lines 1–10 execute in $O(1) \cdot \Theta(n) = O(n)$ time. The sorting of the 5-element groups in lines 12–13 takes $\Theta(n)$ time because each 5-element group takes $\Theta(1)$ time to sort (even using an asymptotically inefficient sorting algorithm such as insertion sort), and there are g elements to sort, where $n/5 - 1 < g \le n/5$. Finally, the time to partition in line 17 is $\Theta(n)$, as Exercise 7.1-3 on page 187 asks you to show. Because the remaining bookkeeping only costs $\Theta(1)$ time, the total amount of time spent outside of the recursive calls is $O(n) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$.

Now let's determine the running time for the recursive calls. The recursive call to find the pivot in line 16 takes $T(g) \le T(n/5)$ time, since $g \le n/5$ and $T(n)$ monotonically increases. Of the two recursive calls in lines 23 and 24, at most one is executed. But we'll see that no matter which of these two recursive calls to SELECT actually executes, the number of elements in the recursive call turns out to be at most $7n/10$, and hence the worst-case cost for lines 23 and 24 is at most $T(7n/10)$. Let's now show that the machinations with group medians and the choice of the pivot x as the median of the group medians guarantees this property.

Figure 9.3 helps to visualize what's going on. There are $g \le n/5$ groups of 5 elements, with each group shown as a column sorted from bottom to top. The arrows show the ordering of elements within the columns. The columns are ordered from left to right with groups to the left of x's group having a group median less than x and those to the right of x's group having a group median greater than x. Although the relative order within each group matters, the relative order among groups to the left of x's column doesn't really matter, and neither does the relative order among groups to the right of x's column. The important thing is that the groups to the left have group medians less than x (shown by the horizontal arrows entering x), and that the groups to the right have group medians greater than x (shown by the horizontal arrows leaving x). Thus, the yellow region contains elements that we know are greater than or equal to x, and the blue region contains elements that we know are less than or equal to x.

These two regions each contain at least $3g/2$ elements. The number of group medians in the yellow region is $\lfloor g/2 \rfloor + 1$, and for each group median, two additional elements are greater than it, making a total of $3(\lfloor g/2 \rfloor + 1) \ge 3g/2$ elements. Similarly, the number of group medians in the blue region is $\lceil g/2 \rceil$, and for each group median, two additional elements are less than it, making a total of $3 \lceil g/2 \rceil \ge 3g/2$.

The elements in the yellow region cannot fall into the low side of the partition around x, and those in the blue region cannot fall into the high side. The elements in neither region—those lying on a white background—could fall into either side of the partition. But since the low side of the partition excludes the elements in the yellow region, and there are a total of $5g$ elements, we know that the low side of the partition can contain at most $5g - 3g/2 = 7g/2 \le 7n/10$ elements. Likewise, the high side of the partition excludes the elements in the blue region, and a similar calculation shows that it also contains at most $7n/10$ elements.

All of which leads to the following recurrence for the worst-case running time of SELECT:

$$T(n) \le T(n/5) + T(7n/10) + \Theta(n) . \tag{9.1}$$

We can show that $T(n) = O(n)$ by substitution.[2] More specifically, we'll prove that $T(n) \le cn$ for some suitably large constant $c > 0$ and all $n > 0$. Substituting this inductive hypothesis into the right-hand side of recurrence (9.1) and assuming that $n \ge 5$ yields

[2] We could also use the Akra-Bazzi method from Section 4.7, which involves calculus, to solve this recurrence. Indeed, a similar recurrence (4.24) on page 117 was used to illustrate that method.

$$\begin{aligned} T(n) &\le c(n/5) + c(7n/10) + \Theta(n) \\ &\le 9cn/10 + \Theta(n) \\ &= cn - cn/10 + \Theta(n) \\ &\le cn \end{aligned}$$

if c is chosen large enough that $c/10$ dominates the upper-bound constant hidden by the $\Theta(n)$. In addition to this constraint, we can pick c large enough that $T(n) \le cn$ for all $n \le 4$, which is the base case of the recursion within SELECT. The running time of SELECT is therefore $O(n)$ in the worst case, and because line 13 alone takes $\Theta(n)$ time, the total time is $\Theta(n)$. ■

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the type of the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input's type, only that the elements are distinct and can be pairwise compared according to a linear order. The algorithms in this chapter are not subject to the $\Omega(n \lg n)$ lower bound, because they manage to solve the selection problem without sorting all the elements. Thus, solving the selection problem by sorting and indexing, as presented in the introduction to this chapter, is asymptotically inefficient in the comparison model.

Exercises

9.3-1
In the algorithm SELECT, the input elements are divided into groups of 5. Show that the algorithm works in linear time if the input elements are divided into groups of 7 instead of 5.

9.3-2
Suppose that the preprocessing in lines 1–10 of SELECT is replaced by a base case for $n \ge n_0$, where n_0 is a suitable constant; that g is chosen as $\lfloor r - p + 1)/5 \rfloor$; and that the elements in $A[5g : n]$ belong to no group. Show that although the recurrence for the running time becomes messier, it still solves to $\Theta(n)$.

9.3-3
Show how to use SELECT as a subroutine to make quicksort run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

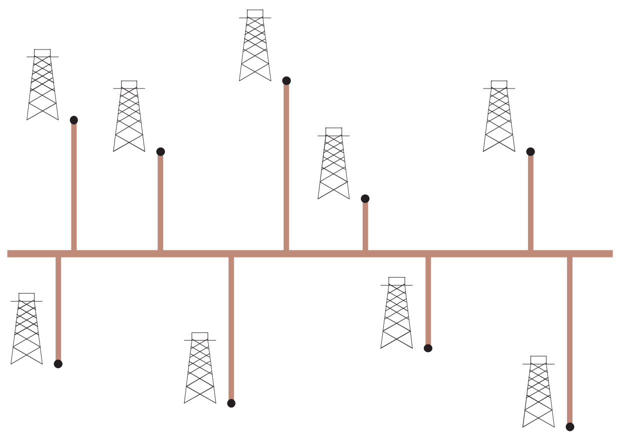


Figure 9.4 Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

★ ***9.3-4***
Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i - 1$ smaller elements and the $n - i$ larger elements without performing any additional comparisons.

9.3-5
Show how to determine the median of a 5-element set using only 6 comparisons.

9.3-6
You have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

9.3-7
Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.4. Given the x- and y-coordinates of the wells, how should the professor pick an optimal location of the main pipeline to minimize the total length of the spurs? Show how to determine an optimal location in linear time.

9.3-8
The kth ***quantiles*** of an n-element set are the $k - 1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

9.3-9
Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \le n$, determines the k numbers in S that are closest to the median of S.

9.3-10
Let $X[1:n]$ and $Y[1:n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y. Assume that all $2n$ numbers are distinct.

Problems

9-1 Largest i numbers in sorted order
You are given a set of n numbers, and you wish to find the i largest in sorted order using a comparison-based algorithm. Describe the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

9-2 Variant of randomized selection
Professor Mendel has proposed simplifying RANDOMIZED-SELECT by eliminating the check for whether i and k are equal. The simplified procedure is SIMPLER-RANDOMIZED-SELECT.

```
SIMPLER-RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]        // 1 ≤ i ≤ r − p + 1 means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i ≤ k
6      return SIMPLER-RANDOMIZED-SELECT(A, p, q, i)
7  else return SIMPLER-RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

a. Argue that in the worst case, SIMPLER-RANDOMIZED-SELECT never terminates.

b. Prove that the expected running time of SIMPLER-RANDOMIZED-SELECT is still $O(n)$.

9-3 Weighted median

Consider n elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$. The ***weighted (lower) median*** is an element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} .$$

For example, consider the following elements x_i and weights w_i:

i	1	2	3	4	5	6	7
x_i	3	8	2	5	4	1	6
w_i	0.12	0.35	0.025	0.08	0.15	0.075	0.2

For these elements, the median is $x_5 = 4$, but the weighted median is $x_7 = 6$. To see why the weighted median is x_7, observe that the elements less than x_7 are x_1, x_3, x_4, x_5, and x_6, and the sum $w_1 + w_3 + w_4 + w_5 + w_6 = 0.45$, which is less than $1/2$. Furthermore, only element x_2 is greater than x_7, and $w_2 = 0.35$, which is no greater than $1/2$.

a. Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

b. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

c. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The ***post-office location problem*** is defined as follows. The input is n points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. A solution is a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i \, d(p, p_i)$, where $d(a, b)$ is the distance between points a and b.

d. Argue that the weighted median is a best solution for the one-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a,b) = |a - b|$.

e. Find the best solution for the two-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a,b) = |x_1 - x_2| + |y_1 - y_2|$.

9-4 Small order statistics

Let's denote by $S(n)$ the worst-case number of comparisons used by SELECT to select the ith order statistic from n numbers. Although $S(n) = \Theta(n)$, the constant hidden by the Θ-notation is rather large. When i is small relative to n, there is an algorithm that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where

$$U_i(n) = \begin{cases} S(n) & \text{if } i \geq n/2 \,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + S(2i) & \text{otherwise} \,. \end{cases}$$

(*Hint:* Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

b. Show that, if $i < n/2$, then $U_i(n) = n + O(S(2i)\lg(n/i))$.

c. Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(S(2n/k)\lg k)$.

9-5 Alternative analysis of randomized selection

In this problem, you will use indicator random variables to analyze the procedure RANDOMIZED-SELECT in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus the call RANDOMIZED-SELECT$(A, 1, n, i)$ returns z_i.

For $1 \leq j < k \leq n$, let

$$X_{ijk} = \text{I}\{z_j \text{ is compared with } z_k \text{ sometime during the execution of the algorithm to find } z_i\} \,.$$

a. Give an exact expression for $\mathrm{E}[X_{ijk}]$. (*Hint:* Your expression may have different values, depending on the values of i, j, and k.)

b. Let X_i denote the total number of comparisons between elements of array A when finding z_i. Show that

$$\mathrm{E}[X_i] \le 2\left(\sum_{j=1}^{i}\sum_{k=i}^{n}\frac{1}{k-j+1} + \sum_{k=i+1}^{n}\frac{k-i-1}{k-i+1} + \sum_{j=1}^{i-2}\frac{i-j-1}{i-j+1}\right).$$

c. Show that $\mathrm{E}[X_i] \le 4n$.

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in $O(n)$ expected time.

9-6 Select with groups of 3

Exercise 9.3-1 asks you to show that the SELECT algorithm still runs in linear time if the elements are divided into groups of 7. This problem asks about dividing into groups of 3.

a. Show that SELECT runs in linear time if you divide the elements into groups whose size is any odd constant greater than 3.

b. Show that SELECT runs in $O(n \lg n)$ time if you divide the elements into groups of size 3.

Because the bound in part (b) is just an upper bound, we do not know whether the groups-of-3 strategy actually runs in $O(n)$ time. But by repeating the groups-of-3 idea on the middle group of medians, we can pick a pivot that guarantees $O(n)$ time. The SELECT3 algorithm on the next page determines the ith smallest of an input array of $n > 1$ distinct elements.

c. Describe in English how the SELECT3 algorithm works. Include in your description one or more suitable diagrams.

d. Show that SELECT3 runs in $O(n)$ time in the worst case.

Chapter notes

The worst-case linear-time median-finding algorithm was devised by Blum, Floyd, Pratt, Rivest, and Tarjan [62]. The fast randomized version is due to Hoare [218]. Floyd and Rivest [147] have developed an improved randomized version that partitions around an element recursively selected from a small sample of the elements.

```
SELECT3$(A, p, r, i)$
while $(r - p + 1) \bmod 9 \neq 0$
    for $j = p + 1$ to $r$                    // put the minimum into $A[p]$
        if $A[p] > A[j]$
            exchange $A[p]$ with $A[j]$
    // If we want the minimum of $A[p:r]$, we're done.
    if $i == 1$
        return $A[p]$
    // Otherwise, we want the $(i-1)$st element of $A[p+1:r]$.
    $p = p + 1$
    $i = i - 1$
$g = (r - p + 1)/3$                           // number of 3-element groups
for $j = p$ to $p + g - 1$                    // run through the groups
    sort $\langle A[j], A[j+g], A[j+2g] \rangle$ in place
// All group medians now lie in the middle third of $A[p:r]$.
$g' = g/3$                                    // number of 3-element subgroups
for $j = p + g$ to $p + g + g' - 1$           // sort the subgroups
    sort $\langle A[j], A[j+g']A[j+2g'] \rangle$ in place
// All subgroup medians now lie in the middle ninth of $A[p:r]$.
// Find the pivot $x$ recursively as the median of the subgroup medians.
$x =$ SELECT3$(A, p + 4g', p + 5g' - 1, \lceil g'/2 \rceil)$
$q =$ PARTITION-AROUND$(A, p, r, x)$          // partition around the pivot
// The rest is just like lines 19–24 of SELECT.
$k = q - p + 1$
if $i == k$
    return $A[q]$                             // the pivot value is the answer
elseif $i < k$
    return SELECT3$(A, p, q - 1, i)$
else return SELECT3$(A, q + 1, r, i - k)$
```

It is still unknown exactly how many comparisons are needed to determine the median. Bent and John [48] gave a lower bound of $2n$ comparisons for median finding, and Schönhage, Paterson, and Pippenger [397] gave an upper bound of $3n$. Dor and Zwick have improved on both of these bounds. Their upper bound [123] is slightly less than $2.95n$, and their lower bound [124] is $(2 + \epsilon)n$, for a small positive constant ϵ, thereby improving slightly on related work by Dor et al. [122]. Paterson [354] describes some of these results along with other related work.

Problem 9-6 was inspired by a paper by Chen and Dumitrescu [84].

Part III Data Structures

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets ***dynamic***. The next four chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. We call a dynamic set that supports these operations a ***dictionary***. Other algorithms require more complicated operations. For example, min-priority queues, which Chapter 6 introduced in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. The best way to implement a dynamic set depends upon the operations that you need to support.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated given a pointer to the object. Some kinds of dynamic sets assume that one of the object's attributes is an identifying ***key***. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain ***satellite data***, which are carried around in other object attributes but are otherwise unused by the set implementation. It may also have attributes that are manipulated by the set operations. These attributes may contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. A total ordering allows us to define the minimum element of the set, for example, or to speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations***, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(S, k)
: A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
: A modifying operation that adds the element pointed to by x to the set S. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
: A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S) and MAXIMUM(S)
: Queries on a totally ordered set S that return a pointer to the element of S with the smallest (for MINIMUM) or largest (for MAXIMUM) key.

SUCCESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

In some situations, we can extend the queries SUCCESSOR and PREDECESSOR so that they apply to sets with nondistinct keys. For a set on n keys, the normal presumption is that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in sorted order.

We usually measure the time taken to execute a set operation in terms of the size of the set. For example, Chapter 13 describes a data structure that can support any of the operations listed above on a set of size n in $O(\lg n)$ time.

Of course, you can always choose to implement a dynamic set with an array. The advantage of doing so is that the algorithms for the dynamic-set operations are simple. The downside, however, is that many of these operations have a worst-case running time of $\Theta(n)$. If the array is not sorted, INSERT and DELETE can take $\Theta(1)$ time, but the remaining operations take $\Theta(n)$ time. If instead the array is maintained in sorted order, then MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR take $\Theta(1)$ time; SEARCH takes $O(\lg n)$ time if implemented with binary search; but INSERT and DELETE take $\Theta(n)$ time in the worst case. The data structures studied in this part improve on the array implementation for many of the dynamic-set operations.

Overview of Part III

Chapters 10–13 describe several data structures that we can use to implement dynamic sets. We'll use many of these data structures later to construct efficient algorithms for a variety of problems. We already saw another important data structure —the heap—in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such as arrays, matrices, stacks, queues, linked lists, and rooted trees. If you have taken an introductory programming course, then much of this material should be familiar to you.

Chapter 11 introduces hash tables, a widely used data structure supporting the dictionary operations INSERT, DELETE, and SEARCH. In the worst case, hash tables require $\Theta(n)$ time to perform a SEARCH operation, but the expected time for hash-table operations is $O(1)$. We rely on probability to analyze hash-table operations, but you can understand how the operations work even without probability.

Binary search trees, which are covered in Chapter 12, support all the dynamic-set operations listed above. In the worst case, each operation takes $\Theta(n)$ time on a tree with n elements. Binary search trees serve as the basis for many other data structures.

Chapter 13 introduces red-black trees, which are a variant of binary search trees. Unlike ordinary binary search trees, red-black trees are guaranteed to perform well: operations take $O(\lg n)$ time in the worst case. A red-black tree is a balanced search tree. Chapter 18 in Part V presents another kind of balanced search tree, called a B-tree. Although the mechanics of red-black trees are somewhat intricate, you can glean most of their properties from the chapter without studying the mechanics in detail. Nevertheless, you probably will find walking through the code to be instructive.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although you can construct many complex data structures using pointers, we present only the rudimentary ones: arrays, matrices, stacks, queues, linked lists, and rooted trees.

10.1 Simple array-based data structures: arrays, matrices, stacks, queues

10.1.1 Arrays

We assume that, as in most programming languages, an array is stored as a contiguous sequence of bytes in memory. If the first element of an array has index s (for example, in an array with 1-origin indexing, $s = 1$), the array starts at memory address a, and each array element occupies b bytes, then the ith element occupies bytes $a + b(i - s)$ through $a + b(i - s + 1) - 1$. Since most of the arrays in this book are indexed starting at 1, and a few starting at 0, we can simplify these formulas a little. When $s = 1$, the ith element occupies bytes $a + b(i - 1)$ through $a + bi - 1$, and when $s = 0$, the ith element occupies bytes $a + bi$ through $a + b(i + 1) - 1$. Assuming that the computer can access all memory locations in the same amount of time (as in the RAM model described in Section 2.2), it takes constant time to access any array element, regardless of the index.

Most programming languages require each element of a particular array to be the same size. If the elements of a given array might occupy different numbers of bytes, then the above formulas fail to apply, since the element size b is not a constant. In such cases, the array elements are usually objects of varying sizes, and what actually appears in each array element is a pointer to the object. The number of bytes occupied by a pointer is typically the same, no matter what the pointer references, so that to access an object in an array, the above formulas give the address of the pointer to the object and then the pointer must be followed to access the object itself.

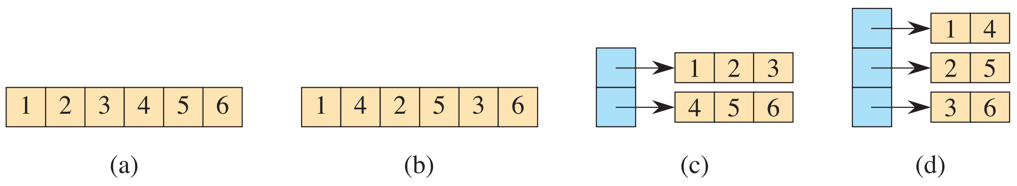


Figure 10.1 Four ways to store the 2×3 matrix M from equation (10.1). **(a)** In row-major order, in a single array. **(b)** In column-major order, in a single array. **(c)** In row-major order, with one array per row (tan) and a single array (blue) of pointers to the row arrays. **(d)** In column-major order, with one array per column (tan) and a single array (blue) of pointers to the column arrays.

10.1.2 Matrices

We typically represent a matrix or two-dimensional array by one or more one-dimensional arrays. The two most common ways to store a matrix are row-major and column-major order. Let's consider an $m \times n$ matrix—a matrix with m rows and n columns. In ***row-major order***, the matrix is stored row by row, and in ***column-major order***, the matrix is stored column by column. For example, consider the 2×3 matrix

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} . \tag{10.1}$$

Row-major order stores the two rows 1 2 3 and 4 5 6, whereas column-major order stores the three columns 1 4; 2 5; and 3 6.

Parts (a) and (b) of Figure 10.1 show how to store this matrix using a single one-dimensional array. It's stored in row-major order in part (a) and in column-major order in part (b). If the rows, columns, and the single array all are indexed starting at s, then $M[i, j]$—the element in row i and column j—is at array index $s + (n(i - s)) + (j - s)$ with row-major order and $s + (m(j - s)) + (i - s)$ with column-major order. When $s = 1$, the single-array indices are $n(i - 1) + j$ with row-major order and $i + m(j - 1)$ with column-major order. When $s = 0$, the single-array indices are simpler: $ni + j$ with row-major order and $i + mj$ with column-major order. For the example matrix M with 1-origin indexing, element $M[2, 1]$ is stored at index $3(2-1)+1 = 4$ in the single array using row-major order and at index $2 + 2(1 - 1) = 2$ using column-major order.

Parts (c) and (d) of Figure 10.1 show multiple-array strategies for storing the example matrix. In part (c), each row is stored in its own array of length n, shown in tan. Another array, with m elements, shown in blue, points to the m row arrays. If we call the blue array A, then $A[i]$ points to the array storing the entries for row i of M, and array element $A[i][j]$ stores matrix element $M[i, j]$. Part (d) shows the column-major version of the multiple-array representation, with n arrays, each of length m, representing the n columns. Matrix element $M[i, j]$ is stored in array element $A[j][i]$.

Single-array representations are typically more efficient on modern machines than multiple-array representations. But multiple-array representations can sometimes be more flexible, for example, allowing for "ragged arrays," in which the rows in the row-major version may have different lengths, or symmetrically for the column-major version, where columns may have different lengths.

Occasionally, other schemes are used to store matrices. In the ***block representation***, the matrix is divided into blocks, and each block is stored contiguously. For example, a 4×4 matrix that is divided into 2×2 blocks, such as

$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

might be stored in a single array in the order $\langle 1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16 \rangle$.

10.1.3 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. Here, you will see how to use an array with attributes to store them.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

Figure 10.2 shows how to implement a stack of at most n elements with an array $S[1:n]$. The stack has attributes $S.top$, indexing the most recently inserted element, and $S.size$, equaling the size n of the array. The stack consists of elements $S[1:S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

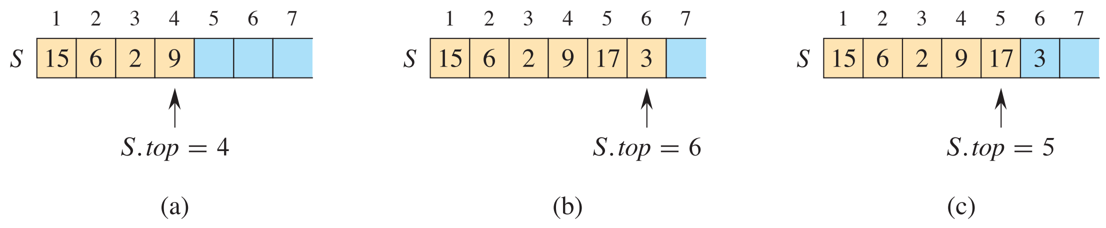


Figure 10.2 An array implementation of a stack S. Stack elements appear only in the tan positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack. The top is element 17.

When $S.top = 0$, the stack contains no elements and is ***empty***. We can test whether the stack is empty with the query operation STACK-EMPTY. Upon an attempt to pop an empty stack, the stack ***underflows***, which is normally an error. If $S.top$ exceeds $S.size$, the stack ***overflows***.

The procedures STACK-EMPTY, PUSH, and POP implement each of the stack operations with just a few lines of code. Figure 10.2 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

```
STACK-EMPTY($S$)
1  if $S.top == 0$
2      return TRUE
3  else return FALSE

PUSH($S, x$)
1  if $S.top == S.size$
2      error "overflow"
3  else $S.top = S.top + 1$
4      $S[S.top] = x$

POP($S$)
1  if STACK-EMPTY($S$)
2      error "underflow"
3  else $S.top = S.top - 1$
4      return $S[S.top + 1]$
```

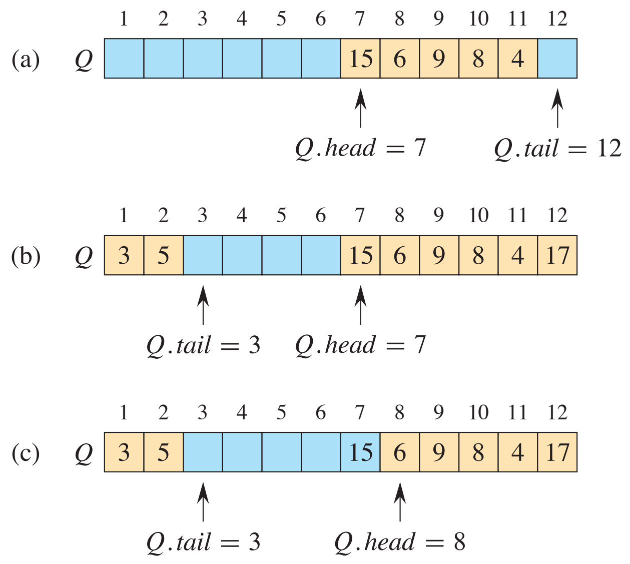


Figure 10.3 A queue implemented using an array $Q[1:12]$. Queue elements appear only in the tan positions. **(a)** The queue has 5 elements, in locations $Q[7:11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE. Like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting for service. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line, who has waited the longest.

Figure 10.3 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1:n]$, with the attribute $Q.size$ equaling the size n of the array. The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations $Q.head$, $Q.head + 1$, $\ldots$, $Q.tail - 1$, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When $Q.head = Q.tail$, the queue is empty. Initially, we have $Q.head = Q.tail = 1$. An attempt to dequeue an element from an empty queue causes the queue to underflow. When $Q.head = Q.tail + 1$ or both

$Q.head = 1$ and $Q.tail = Q.size$, the queue is full, and an attempt to enqueue an element causes the queue to overflow.

In the procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. (Exercise 10.1-5 asks you to supply these checks.) Figure 10.3 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

```
ENQUEUE($Q, x$)
1  $Q[Q.tail] = x$
2  if $Q.tail == Q.size$
3      $Q.tail = 1$
4  else $Q.tail = Q.tail + 1$

DEQUEUE($Q$)
1  $x = Q[Q.head]$
2  if $Q.head == Q.size$
3      $Q.head = 1$
4  else $Q.head = Q.head + 1$
5  return $x$
```

Exercises

10.1-1

Consider an $m \times n$ matrix in row-major order, where both m and n are powers of 2 and rows and columns are indexed from 0. We can represent a row index i in binary by the $\lg m$ bits $\langle i_{\lg m-1}, i_{\lg m-2}, \ldots, i_0 \rangle$ and a column index j in binary by the $\lg n$ bits $\langle j_{\lg n-1}, j_{\lg n-2}, \ldots, j_0 \rangle$. Suppose that this matrix is a 2×2 block matrix, where each block has $m/2$ rows and $n/2$ columns, and it is to be represented by a single array with 0-origin indexing. Show how to construct the binary representation of the $(\lg m + \lg n)$-bit index into the single array from the binary representations of i and j.

10.1-2

Using Figure 10.2 as a model, illustrate the result of each operation in the sequence PUSH$(S, 4)$, PUSH$(S, 1)$, PUSH$(S, 3)$, POP(S), PUSH$(S, 8)$, and POP(S) on an initially empty stack S stored in array $S[1:6]$

10.1-3

Explain how to implement two stacks in one array $A[1:n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.

10.1-4

Using Figure 10.3 as a model, illustrate the result of each operation in the sequence ENQUEUE$(Q, 4)$, ENQUEUE$(Q, 1)$, ENQUEUE$(Q, 3)$, DEQUEUE(Q), ENQUEUE$(Q, 8)$, and DEQUEUE(Q) on an initially empty queue Q stored in array $Q[1:6]$.

10.1-5

Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

10.1-6

Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue, pronounced like "deck") allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

10.1-7

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-8

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

10.2 Linked lists

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Since the elements of linked lists often contain keys that can be searched for, linked lists are sometimes called ***search lists***. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 250.

As shown in Figure 10.4, each element of a ***doubly linked list*** L is an object with an attribute *key* and two pointer attributes: *next* and *prev*. The object may

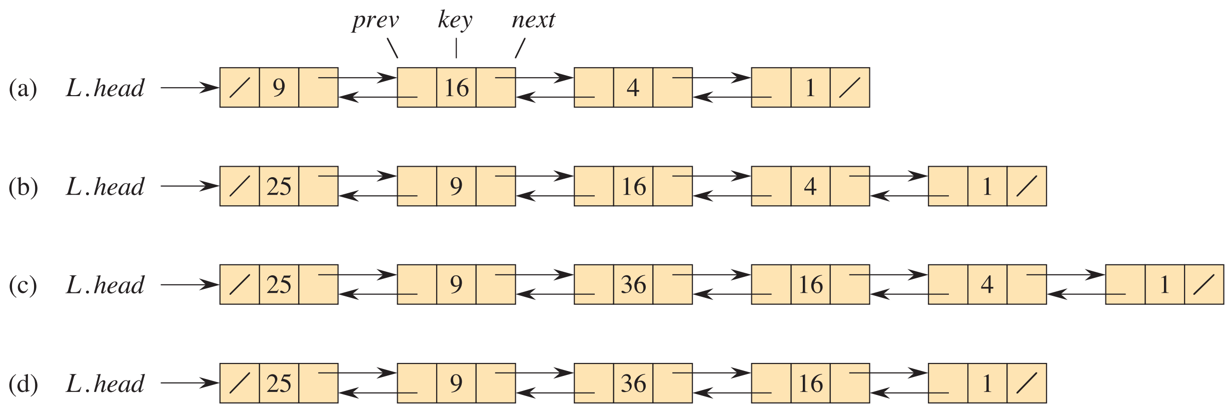


Figure 10.4 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-PREPEND(L, x), where $x.key = 25$, the linked list has an object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of calling LIST-INSERT(x, y), where $x.key = 36$ and y points to the object with key 9. **(d)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

also contain other satellite data. Given an element x in the list, $x.next$ points to its successor in the linked list, and $x.prev$ points to its predecessor. If $x.prev = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $x.next = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $L.head$ points to the first element of the list. If $L.head = \text{NIL}$, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is ***singly linked***, each element has a *next* pointer but not a *prev* pointer. If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list. The minimum element is then the head of the list, and the maximum element is the tail. If the list is ***unsorted***, the elements can appear in any order. In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. You can think of a circular list as a ring of elements. In the remainder of this section, we assume that the lists we are working with are unsorted and doubly linked.

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 10.4(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL. To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

```
LIST-SEARCH(L, k)
1  $x = L.head$
2  while $x \neq$ NIL and $x.key \neq k$
3      $x = x.next$
4  return $x$
```

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the LIST-PREPEND procedure adds x to the front of the linked list, as shown in Figure 10.4(b). (Recall that our attribute notation can cascade, so that $L.head.prev$ denotes the *prev* attribute of the object that $L.head$ points to.) The running time for LIST-PREPEND on a list of n elements is $O(1)$.

```
LIST-PREPEND(L, x)
1  $x.next = L.head$
2  $x.prev =$ NIL
3  if $L.head \neq$ NIL
4      $L.head.prev = x$
5  $L.head = x$
```

You can insert anywhere within a linked list. As Figure 10.4(c) shows, if you have a pointer y to an object in the list, the LIST-INSERT procedure on the facing page "splices" a new element x into the list, immediately following y, in $O(1)$ time. Since LIST-INSERT never references the list object L, it is not supplied as a parameter.

```
LIST-INSERT(x, y)
1  $x.next = y.next$
2  $x.prev = y$
3  if $y.next \neq$ NIL
4      $y.next.prev = x$
5  $y.next = x$
```

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "‘splices” x out of the list by updating pointers. To delete an element with a given key, first call LIST-SEARCH to retrieve a pointer to the element. Figure 10.4(d) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but to delete an element with a given key, the call to LIST-SEARCH makes the worst-case running time be $\Theta(n)$.

```
LIST-DELETE(L, x)
1  if $x.prev \neq$ NIL
2      $x.prev.next = x.next$
3  else $L.head = x.next$
4  if $x.next \neq$ NIL
5      $x.next.prev = x.prev$
```

Insertion and deletion are faster operations on doubly linked lists than on arrays. If you want to insert a new first element into an array or delete the first element in an array, maintaining the relative order of all the existing elements, then each of the existing elements needs to be moved by one position. In the worst case, therefore, insertion and deletion take $\Theta(n)$ time in an array, compared with $O(1)$ time for a doubly linked list. (Exercise 10.2-1 asks you to show that deleting an element from a singly linked list takes $\Theta(n)$ time in the worst case.) If, however, you want to find the kth element in the linear order, it takes just $O(1)$ time in an array regardless of k, but in a linked list, you'd have to traverse k elements, taking $\Theta(k)$ time.

Sentinels

The code for LIST-DELETE is simpler if you ignore the boundary conditions at the head and tail of the list:

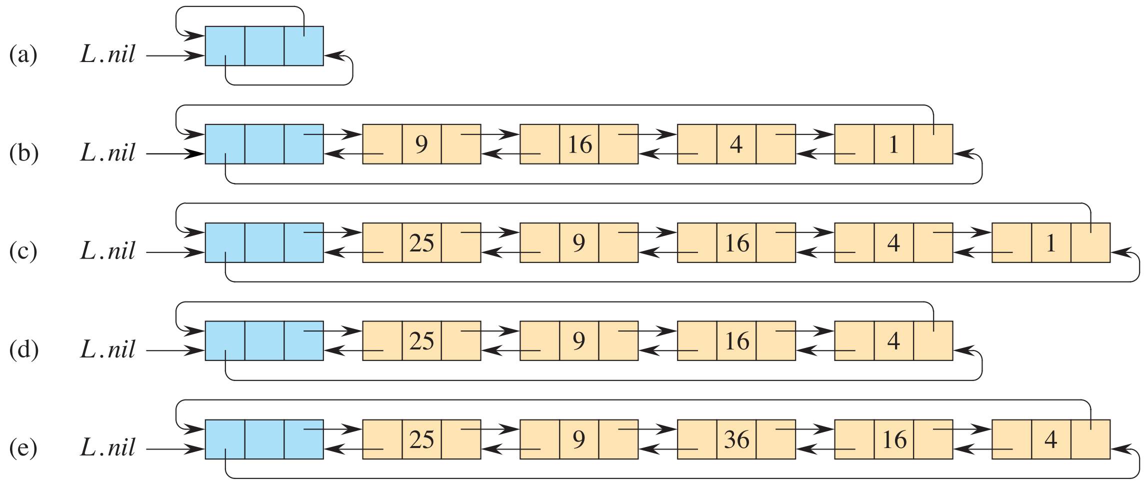


Figure 10.5 A circular, doubly linked list with a sentinel. The sentinel $L.nil$, in blue, appears between the head and tail. The attribute $L.head$ is no longer needed, since the head of the list is $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.4(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT′$(x, L.nil)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4. **(e)** The list after executing LIST-INSERT′(x, y), where $x.key = 36$ and y points to the object with key 9.

```
LIST-DELETE′(x)
1  $x.prev.next = x.next$
2  $x.next.prev = x.prev$
```

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. In a linked list L, the sentinel is an object $L.nil$ that represents NIL but has all the attributes of the other objects in the list. References to NIL are replaced by references to the sentinel $L.nil$. As shown in Figure 10.5, this change turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel $L.nil$ lies between the head and tail. The attribute $L.nil.next$ points to the head of the list, and $L.nil.prev$ points to the tail. Similarly, both the *next* attribute of the tail and the *prev* attribute of the head point to $L.nil$. Since $L.nil.next$ points to the head, the attribute $L.head$ is eliminated altogether, with references to it replaced by references to $L.nil.next$. Figure 10.5(a) shows that an empty list consists of just the sentinel, and both $L.nil.next$ and $L.nil.prev$ point to $L.nil$.

To delete an element from the list, just use the two-line procedure LIST-DELETE′ from before. Just as LIST-INSERT never references the list object L, neither does

LIST-DELETE′. You should never delete the sentinel $L.nil$ unless you are deleting the entire list!

The LIST-INSERT′ procedure inserts an element x into the list following object y. No separate procedure for prepending is necessary: to insert at the head of the list, let y be $L.nil$; and to insert at the tail, let y be $L.nil.prev$. Figure 10.5 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

```
LIST-INSERT′$(x, y)$
$x.next = y.next$
$x.prev = y$
$y.next.prev = x$
$y.next = x$
```

Searching a circular, doubly linked list with a sentinel has the same asymptotic running time as without a sentinel, but it is possible to decrease the constant factor. The test in line 2 of LIST-SEARCH makes two comparisons: one to check whether the search has run off the end of the list and, if not, one to check whether the key resides in the current element x. Suppose that you *know* that the key is somewhere in the list. Then you do not need to check whether the search runs off the end of the list, thereby eliminating one comparison in each iteration of the **while** loop.

The sentinel provides a place to put the key before starting the search. The search starts at the head $L.nil.next$ of list L, and it stops if it finds the key somewhere in the list. Now the search is guaranteed to find the key, either in the sentinel or before reaching the sentinel. If the key is found before reaching the sentinel, then it really is in the element where the search stops. If, however, the search goes through all the elements in the list and finds the key only in the sentinel, then the key is not really in the list, and the search returns NIL. The procedure LIST-SEARCH′ embodies this idea. (If your sentinel requires its *key* attribute to be NIL, then you might want to assign $L.nil.key =$ NIL before line 5.)

```
LIST-SEARCH′$(L, k)$
1  $L.nil.key = k$        // store the key in the sentinel to guarantee it is in list
2  $x = L.nil.next$       // start at the head of the list
3  while $x.key \neq k$
4      $x = x.next$
5  if $x == L.nil$        // found $k$ in the sentinel
6      return NIL         // $k$ was not really in the list
7  else return $x$        // found $k$ in element $x$
```

Sentinels often simplify code and, as in searching a linked list, they might speed up code by a small constant factor, but they don't typically improve the asymptotic running time. Use them judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they significantly simplify the code.

Exercises

10.2-1
Explain why the dynamic-set operation INSERT on a singly linked list can be implemented in $O(1)$ time, but the worst-case time for DELETE is $\Theta(n)$.

10.2-2
Implement a stack using a singly linked list. The operations PUSH and POP should still take $O(1)$ time. Do you need to add any attributes to the list?

10.2-3
Implement a queue using a singly linked list. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time. Do you need to add any attributes to the list?

10.2-4
The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-5
Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

★ ***10.2-6***
Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two (*next* and *prev*). Assume that all pointer values can be interpreted as k-bit integers, and define $x.np = x.next$ XOR $x.prev$, the k-bit "exclusive-or" of $x.next$ and $x.prev$. The value NIL is represented by 0. Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Representing rooted trees

Linked lists work well for representing linear relationships, but not all relationships are linear. In this section, we look specifically at the problem of representing rooted trees by linked data structures. We first look at binary trees, and then we present a method for rooted trees in which nodes can have an arbitrary number of children.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a *key* attribute. The remaining attributes of interest are pointers to other nodes, and they vary according to the type of tree.

Binary trees

Figure 10.6 shows how to use the attributes p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $x.p = \text{NIL}$, then x is the root. If node x has no left child, then $x.\mathit{left} = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $T.\mathit{root}$. If $T.\mathit{root} = \text{NIL}$, then the tree is empty.

Rooted trees with unbounded branching

It's simple to extend the scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant k: replace the *left* and *right* attributes by $\mathit{child}_1, \mathit{child}_2, \ldots, \mathit{child}_k$. This scheme no longer works when the number of children of a node is unbounded, however, since we do not know how many attributes to allocate in advance. Moreover, if k, the number of children, is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any n-node rooted tree. The ***left-child, right-sibling representation*** appears in Figure 10.7. As before, each node contains a parent pointer p, and $T.\mathit{root}$ points to the root of tree T. Instead of having a pointer to each of its children, however, each node x has only two pointers:

1. $x.\mathit{left\text{-}child}$ points to the leftmost child of node x, and
2. $x.\mathit{right\text{-}sibling}$ points to the sibling of x immediately to its right.

If node x has no children, then $x.\mathit{left\text{-}child} = \text{NIL}$, and if node x is the rightmost child of its parent, then $x.\mathit{right\text{-}sibling} = \text{NIL}$.

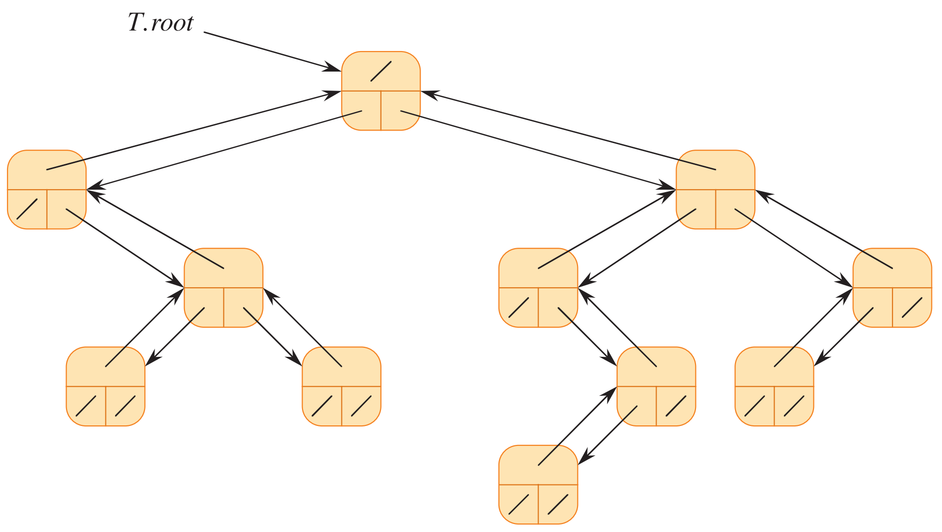


Figure 10.6 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

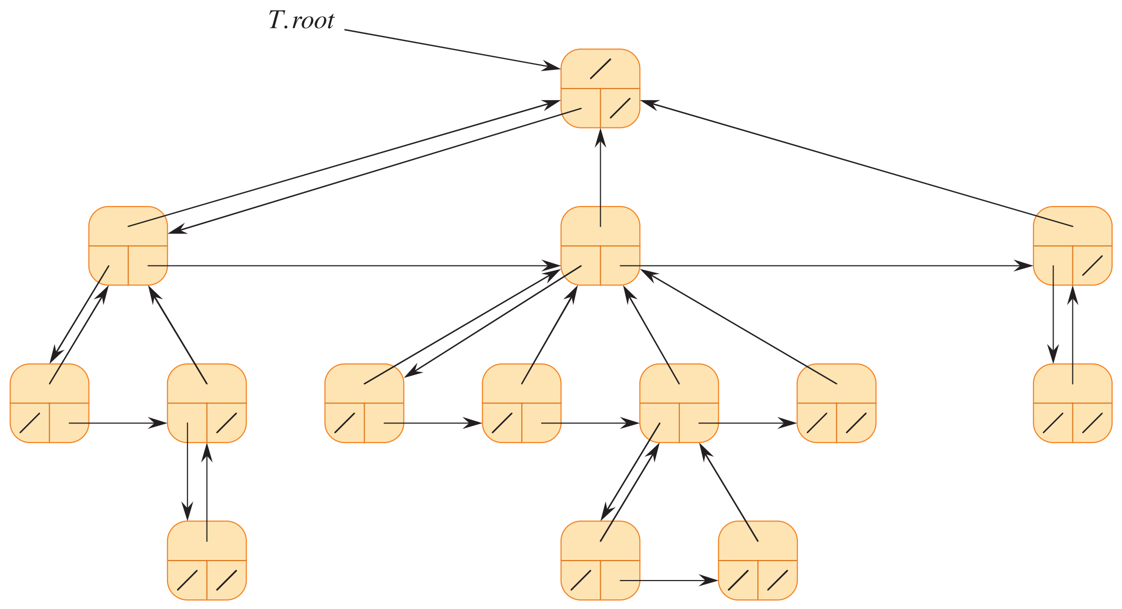


Figure 10.7 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example, we represented a heap, which is based on a complete binary tree, by a single array along with an attribute giving the index of the last node in the heap. The trees that appear in Chapter 19 are traversed only toward the root, and so only the parent pointers are present: there are no pointers to children. Many other schemes are possible. Which scheme is best depends on the application.

Exercises

10.3-1
Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	*key*	*left*	*right*
1	17	8	9
2	14	NIL	NIL
3	12	NIL	NIL
4	20	10	NIL
5	33	2	NIL
6	15	1	4
7	28	NIL	NIL
8	22	NIL	NIL
9	13	3	7
10	25	NIL	5

10.3-2
Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

10.3-3
Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.3-4
Write an $O(n)$-time procedure that prints out all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

★ ***10.3-5***
Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node. Use no more than constant extra space outside

of the tree itself and do not modify the tree, even temporarily, during the procedure.

★ ***10.3-6***

The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be accessed in constant time and all its children can be accessed in time linear in the number of children. Show how to use only two pointers and one boolean value in each node x so that x's parent or all of x's children can be accessed in time linear in the number of x's children.

Problems

10-1 Comparisons among lists

For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH				
INSERT				
DELETE				
SUCCESSOR				
PREDECESSOR				
MINIMUM				
MAXIMUM				

10-2 Mergeable heaps using linked lists

A ***mergeable heap*** supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION.[1]

[1] Because we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also refer to it as a ***mergeable min-heap***. Alternatively, if it supports MAXIMUM and EXTRACT-MAX, it is a ***mergeable max-heap***.

Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.

a. Lists are sorted.

b. Lists are unsorted.

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

10-3 Searching a sorted compact list

We can represent a singly linked list with two arrays, *key* and *next*. Given the index i of an element, its value is stored in $key[i]$, and the index of its successor is given by $next[i]$, where $next[i] = \text{NIL}$ for the last element. We also need the index *head* of the first element in the list. An n-element list stored in this way is ***compact*** if it is stored only in positions 1 through n of the *key* and *next* arrays.

Let's assume that all keys are distinct and that the compact list is also sorted, that is, $key[i] < key[next[i]]$ for all $i = 1, 2, \ldots, n$ such that $next[i] \neq \text{NIL}$. Under these assumptions, you will show that the randomized algorithm COMPACT-LIST-SEARCH searches the list for key k in $O(\sqrt{n})$ expected time.

```
COMPACT-LIST-SEARCH($key, next, head, n, k$)
1   $i = head$
2   while $i \neq$ NIL and $key[i] < k$
3       $j =$ RANDOM$(1, n)$
4       if $key[i] < key[j]$ and $key[j] \leq k$
5           $i = j$
6           if $key[i] == k$
7               return $i$
8       $i = next[i]$
9   if $i ==$ NIL or $key[i] > k$
10      return NIL
11  else return $i$
```

If you ignore lines 3–7 of the procedure, you can see that it's an ordinary algorithm for searching a sorted linked list, in which index i points to each position of the list in turn. The search terminates once the index i "falls off" the end of the list or once $key[i] \geq k$. In the latter case, if $key[i] = k$, the procedure has found a key with the value k. If, however, $key[i] > k$, then the search will never find a key with the value k, so that terminating the search was the correct action.

Lines 3–7 attempt to skip ahead to a randomly chosen position j. Such a skip helps if $key[j]$ is larger than $key[i]$ and no larger than k. In such a case, j marks a position in the list that i would reach during an ordinary list search. Because the list is compact, we know that any choice of j between 1 and n indexes some element in the list.

Instead of analyzing the performance of COMPACT-LIST-SEARCH directly, you will analyze a related algorithm, COMPACT-LIST-SEARCH′, which executes two separate loops. This algorithm takes an additional parameter t, which specifies an upper bound on the number of iterations of the first loop.

```
COMPACT-LIST-SEARCH′(key, next, head, n, k, t)
 1  i = head
 2  for q = 1 to t
 3      j = RANDOM(1, n)
 4      if key[i] < key[j] and key[j] ≤ k
 5          i = j
 6          if key[i] == k
 7              return i
 8  while i ≠ NIL and key[i] < k
 9      i = next[i]
10  if i == NIL or key[i] > k
11      return NIL
12  else return i
```

To compare the execution of the two algorithms, assume that the sequence of calls of RANDOM$(1, n)$ yields the same sequence of integers for both algorithms.

a. Argue that for any value of t, COMPACT-LIST-SEARCH$(key, next, head, n, k)$ and COMPACT-LIST-SEARCH′$(key, next, head, n, k, t)$ return the same result and that the number of iterations of the **while** loop of lines 2–8 in COMPACT-LIST-SEARCH is at most the total number of iterations of both the **for** and **while** loops in COMPACT-LIST-SEARCH′.

In the call COMPACT-LIST-SEARCH′$(key, next, head, n, k, t)$, let X_t be the random variable that describes the distance in the linked list (that is, through the chain of *next* pointers) from position i to the desired key k after t iterations of the **for** loop of lines 2–7 have occurred.

b. Argue that COMPACT-LIST-SEARCH′$(key, next, head, n, k, t)$ has an expected running time of $O(t + \mathrm{E}[X_t])$.

c. Show that $\mathrm{E}[X_t] = \sum_{r=1}^{n}(1-r/n)^t$. (*Hint:* Use equation (C.28) on page 1193.)

d. Show that $\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1)$. (*Hint:* Use inequality (A.18) on page 1150.)

e. Prove that $\mathrm{E}[X_t] \le n/(t+1)$.

f. Show that COMPACT-LIST-SEARCH$'(key, next, head, n, k, t)$ has an expected running time of $O(t + n/t)$.

g. Conclude that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.

h. Why do we assume that all keys are distinct in COMPACT-LIST-SEARCH? Argue that random skips do not necessarily help asymptotically when the list contains repeated key values.

Chapter notes

Aho, Hopcroft, and Ullman [6] and Knuth [259] are excellent references for elementary data structures. Many other texts cover both basic data structures and their implementation in a particular programming language. Examples of these types of textbooks include Goodrich and Tamassia [196], Main [311], Shaffer [406], and Weiss [452, 453, 454]. The book by Gonnet and Baeza-Yates [193] provides experimental data on the performance of many data-structure operations.

The origin of stacks and queues as data structures in computer science is unclear, since corresponding notions already existed in mathematics and paper-based business practices before the introduction of digital computers. Knuth [259] cites A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

11 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$. Indeed, the built-in dictionaries of Python are implemented with hash tables.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array takes advantage of the $O(1)$ access time for any array element. Section 11.1 discusses direct addressing in more detail. To use direct addressing, you must be able to allocate an array that contains a position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, we *compute* the array index from the key. Section 11.2 presents the main ideas, focusing on "chaining" as a way to handle "collisions," in which more than one key maps to the same array index. Section 11.3 describes how to compute array indices from keys using hash functions. We present and analyze several variations on the basic theme. Section 11.4 looks at "open addressing," which is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average. Section 11.5 discusses the hierarchical memory systems of modern computer systems have and illustrates how to design hash tables that work well in such systems.

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a distinct key drawn from the universe $U = \{0, 1, \ldots, m - 1\}$, where m is not too large.

To represent the dynamic set, you can use an array, or ***direct-address table***, denoted by $T[0:m-1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 11.1 illustrates this approach. Slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS-INSERT, and DIRECT-ADDRESS-DELETE on the following page are trivial to implement. Each takes only $O(1)$ time.

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, save space by storing the object directly in the slot. To indicate an empty slot, use a special key. Then again, why store the key of the object at all? The index of the object *is* its key! Of course, then you'd need some way to tell whether slots are empty.

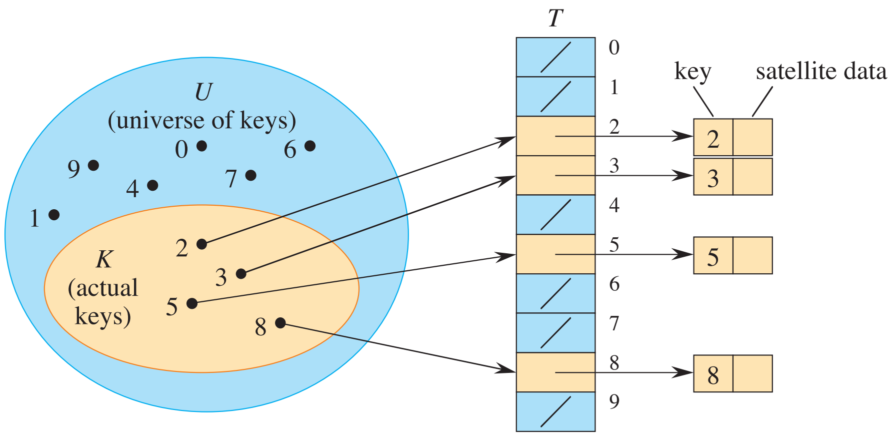


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index into the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, in blue, contain NIL.

```
DIRECT-ADDRESS-SEARCH$(T, k)$
1  return $T[k]$

DIRECT-ADDRESS-INSERT$(T, x)$
1  $T[x.key] = x$

DIRECT-ADDRESS-DELETE$(T, x)$
1  $T[x.key] =$ NIL
```

Exercises

11.1-1
A dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

11.1-2
A ***bit vector*** is simply an array of bits (each either 0 or 1). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements drawn from the set $\{0, 1, \ldots, m-1\}$ and with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3
Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

★ ***11.1-4***
Suppose that you want to implement a dictionary by using direct addressing on a *huge* array. That is, if the array size is m and the dictionary contains at most n elements at any one time, then $m \gg n$. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (*Hint:* Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2 Hash tables

The downside of direct addressing is apparent: if the universe U is large or infinite, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirement reduces to $\Theta(|K|)$ while maintaining the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*,[1] whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k, but with hashing, we use a ***hash function*** h to compute the slot number from the key k, so that the element goes into slot $h(k)$. The hash function h maps the universe U of keys into the slots of a ***hash table*** $T[0:m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\} ,$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k ***hashes*** to slot $h(k)$, and we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m. An example of a simple, but not particularly good, hash function is $h(k) = k \bmod m$.

There is one hitch, namely that two keys may hash to the same slot. We call this situation a ***collision***. Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution is to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be "random," thus avoiding collisions or at least minimizing their number. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k must always produce the same output $h(k)$.) Because $|U| > m$, however, there must be at least two keys that have the same hash value,

[1] The definition of "average-case" requires care—are we assuming an input distribution over the keys, or are we randomizing the choice of hash function itself? We'll consider both approaches, but with an emphasis on the use of a randomly chosen hash function.

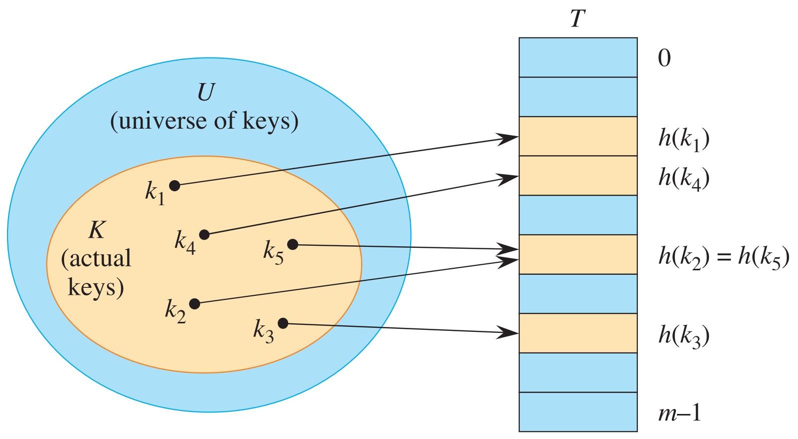


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

and avoiding collisions altogether is impossible. Thus, although a well-designed, "random"-looking hash function can reduce the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section first presents a definition of "independent uniform hashing," which captures the simplest notion of what it means for a hash function to be "random." It then presents and analyzes the simplest collision resolution technique, called chaining. Section 11.4 introduces an alternative method for resolving collisions, called open addressing.

Independent uniform hashing

An "ideal" hashing function h would have, for each possible input k in the domain U, an output $h(k)$ that is an element randomly and independently chosen uniformly from the range $\{0, 1, \ldots, m-1\}$. Once a value $h(k)$ is randomly chosen, each subsequent call to h with the same input k yields the same output $h(k)$.

We call such an ideal hash function an ***independent uniform hash function***. Such a function is also often called a ***random oracle*** [43]. When hash tables are implemented with an independent uniform hash function, we say we are using ***independent uniform hashing***.

Independent uniform hashing is an ideal theoretical abstraction, but it is not something that can reasonably be implemented in practice. Nonetheless, we'll analyze the efficiency of hashing under the assumption of independent uniform hashing and then present ways of achieving useful practical approximations to this ideal.

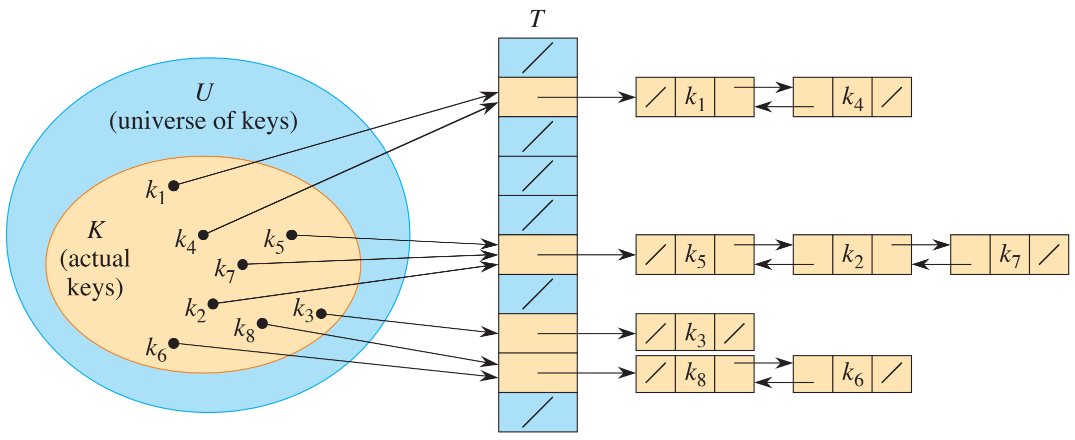


Figure 11.3 Collision resolution by chaining. Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The list can be either singly or doubly linked. We show it as doubly linked because deletion may be faster that way when the deletion procedure knows which list element (not just which key) is to be deleted.

Collision resolution by chaining

At a high level, you can think of hashing with chaining as a nonrecursive form of divide-and-conquer: the input set of n elements is divided randomly into m subsets, each of approximate size n/m. A hash function determines which subset an element belongs to. Each subset is managed independently as a list.

Figure 11.3 shows the idea behind ***chaining***: each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list. Slot j contains a pointer to the head of the list of all stored elements with hash value j. If there are no such elements, then slot j contains NIL.

When collisions are resolved by chaining, the dictionary operations are straightforward to implement. They appear on the next page and use the linked-list procedures from Section 10.2. The worst-case running time for insertion is $O(1)$. The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table. To enforce this assumption, you can search (at additional cost) for an element whose key is $x.key$ before inserting. For searching, the worst-case running time is proportional to the length of the list. (We'll analyze this operation more closely below.) Deletion takes $O(1)$ time if the lists are doubly linked, as in Figure 11.3. (Since CHAINED-HASH-DELETE takes as input an element x and not its key k, no search is needed. If the hash table supports deletion, then its linked lists should be doubly linked in order to delete an item quickly. If the lists were only singly linked, then by Exercise 10.2-1, deletion

```
CHAINED-HASH-INSERT$(T, x)$
1  LIST-PREPEND$(T[h(x.key)], x)$

CHAINED-HASH-SEARCH$(T, k)$
1  return LIST-SEARCH$(T[h(k)], k)$

CHAINED-HASH-DELETE$(T, x)$
1  LIST-DELETE$(T[h(x.key)], x)$
```

could take time proportional to the length of the list. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than using one linked list for all the elements. We clearly don't use hash tables for their worst-case performance.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average (meaning with respect to the distribution of keys to be hashed and with respect to the choice of hash function, if this choice is randomized). Section 11.3 discusses these issues, but for now we assume that any given element is equally likely to hash into any of the m slots. That is, the hash function is ***uniform***. We further assume that where a given element hashes to is *independent* of where any other elements hash to. In other words, we assume that we are using ***independent uniform hashing***.

Because hashes of distinct keys are assumed to be independent, independent uniform hashing is ***universal***: the chance that any two distinct keys k_1 and k_2 collide is at most $1/m$. Universality is important in our analysis and also in the specification of universal families of hash functions, which we'll see in Section 11.3.2.

For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j, so that

$$n = n_0 + n_1 + \cdots + n_{m-1} , \tag{11.1}$$

and the expected value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, we'll consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. We consider two cases. In the first, the search is unsuccessful: no element in the table has key k. In the second, the search successfully finds an element with key k.

Theorem 11.1
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof Under the assumption of independent uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$. ■

The situation for a successful search is slightly different. An unsuccessful search is equally likely to go to any slot of the hash table. A successful search, however, cannot go to an empty slot, since it is for an element that is present in one of the linked lists. We assume that the element searched for is equally likely to be any one of the elements in the table, so the longer the list, the more likely that the search is for one of its elements. Even so, the expected search time still turns out to be $\Theta(1 + \alpha)$.

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes $\Theta(1 + \alpha)$ time on average, under the assumption of independent uniform hashing.

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. Let x_i denote the ith element inserted into the table, for $i = 1, 2, \ldots, n$, and let $k_i = x_i.key$.

Our analysis uses indicator random variables extensively. For each slot q in the table and for each pair of distinct keys k_i and k_j, we define the indicator random variable

$$X_{ijq} = \text{I}\{\text{the search is for } x_i, h(k_i) = q, \text{and } h(k_j) = q\} .$$

That is, $X_{ijq} = 1$ when keys k_i and k_j collide at slot q and the search is for element x_i. Because $\Pr\{\text{the search is for } x_i\} = 1/n$, $\Pr\{h(k_i) = q\} = 1/m$, $\Pr\{h(k_j) = q\} = 1/m$, and these events are all independent, we have that $\Pr\{X_{ijq} = 1\} = 1/nm^2$. Lemma 5.1 on page 130 gives $\text{E}[X_{ijq}] = 1/nm^2$.

Next, we define, for each element x_j, the indicator random variable

$$\begin{aligned} Y_j &= \text{I}\{x_j \text{ appears in a list prior to the element being searched for}\} \\ &= \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq} , \end{aligned}$$

since at most one of the X_{ijq} equals 1, namely when the element x_i being searched for belongs to the same list as x_j (pointed to by slot q), and $i < j$ (so that x_i appears after x_j in the list).

Our final random variable is Z, which counts how many elements appear in the list prior to the element being searched for:

$$Z = \sum_{j=1}^{n} Y_j .$$

Because we must count the element being searched for as well as all those preceding it in its list, we wish to compute $\text{E}[Z + 1]$. Using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \text{E}[Z + 1] &= \text{E}\left[1 + \sum_{j=1}^{n} Y_j\right] \\ &= 1 + \text{E}\left[\sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \text{E}\left[\sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} X_{ijq}\right] \\ &= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \text{E}[X_{ijq}] \qquad \text{(by linearity of expectation)} \end{aligned}$$

$$
\begin{aligned}
&= 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \frac{1}{nm^2} \\
&= 1 + m \cdot \frac{n(n-1)}{2} \cdot \frac{1}{nm^2} \qquad \text{(by equation (A.2) on page 1141)} \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} .
\end{aligned}
$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

What does this analysis mean? If the number of elements in the table is at most proportional to the number of hash-table slots, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked (assuming that the list element to be deleted is known, and not just its key), we can support all dictionary operations in $O(1)$ time on average.

The analysis in the preceding two theorems depends only on two essential properties of independent uniform hashing: uniformity (each key is equally likely to hash to any one of the m slots), and independence (so any two distinct keys collide with probability $1/m$).

Exercises

11.2-1

You use a hash function h to hash n distinct keys into an array T of length m. Assuming independent uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k_1, k_2\} : k_1 \neq k_2 \text{ and } h(k_1) = h(k_2)\}$?

11.2-2

Consider a hash table with 9 slots and the hash function $h(k) = k \bmod 9$. Demonstrate what happens upon inserting the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ with collisions resolved by chaining.

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by creating a "free list": a linked list of all the unused slots. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

11.2-5

You need to store a set of n keys in a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > (n-1)m$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.2-6

You have stored n keys in a hash table of size m, with collisions resolved by chaining, and you know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

11.3 Hash functions

For hashing to work well, it needs a good hash function. Along with being efficiently computable, what properties does a good hash function have? How do you design good hash functions?

This section first attempts to answer these questions based on two ad hoc approaches for creating hash functions: hashing by division and hashing by multiplication. Although these methods work well for some sets of input keys, they are limited because they try to provide a single fixed hash function that works well on any data—an approach called ***static hashing***.

We then see that provably good average-case performance for *any* data can be obtained by designing a suitable *family* of hash functions and choosing a hash function at random from this family at runtime, independent of the data to be hashed. The approach we examine is called random hashing. A particular kind of random hashing, universal hashing, works well. As we saw with quicksort in Chapter 7, randomization is a powerful algorithmic design tool.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of independent uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other keys have hashed to. What does "equally likely" mean here? If the hash function is fixed, any probabilities would have to be based on the probability distribution of the input keys.

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally you might know the distribution. For example, if you know that the keys are random real numbers k independently and uniformly distributed in the range $0 \le k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of independent uniform hashing.

A good static hashing approach derives the hash value in a way that you expect to be independent of any patterns that might exist in the data. For example, the "division method" (discussed in Section 11.3.1) computes the hash value as the remainder when the key is divided by a specified prime number. This method may give good results, if you (somehow) choose a prime number that is unrelated to any patterns in the distribution of keys.

Random hashing, described in Section 11.3.2, picks the hash function to be used at random from a suitable family of hashing functions. This approach removes any need to know anything about the probability distribution of the input keys, as the randomization necessary for good average-case behavior then comes from the (known) random process used to pick the hash function from the family of hash functions, rather than from the (unknown) process used to create the input keys. We recommend that you use random hashing.

Keys are integers, vectors, or strings

In practice, a hash function is designed to handle keys that are one of the following two types:

- A short nonnegative integer that fits in a w-bit machine word. Typical values for w would be 32 or 64.

- A short vector of nonnegative integers, each of bounded size. For example, each element might be an 8-bit byte, in which case the vector is often called a (byte) string. The vector might be of variable length.

To begin, we assume that keys are short nonnegative integers. Handling vector keys is more complicated and discussed in Sections 11.3.5 and 11.5.2.

11.3.1 Static hashing

Static hashing uses a single, fixed hash function. The only randomization available is through the (usually unknown) distribution of input keys. This section discusses two standard approaches for static hashing: the division method and the multiplication method. Although static hashing is no longer recommended, the multiplication method also provides a good foundation for "nonstatic" hashing—better known as random hashing—where the hash function is chosen at random from a suitable family of hash functions.

The division method

The ***division method*** for creating hash functions maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \bmod m .$$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

The division method may work well when m is a prime not too close to an exact power of 2. There is no guarantee that this method provides good average-case performance, however, and it may complicate applications since it constrains the size of the hash tables to be prime.

The multiplication method

The general ***multiplication method*** for creating hash functions operates in two steps. First, multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, multiply this value by m and take the floor of the result. That is, the hash function is

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor ,$$

where "$kA \bmod 1$" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$. The general multiplication method has the advantage that the value of m is not critical and you can choose it independently of how you choose the multiplicative constant A.

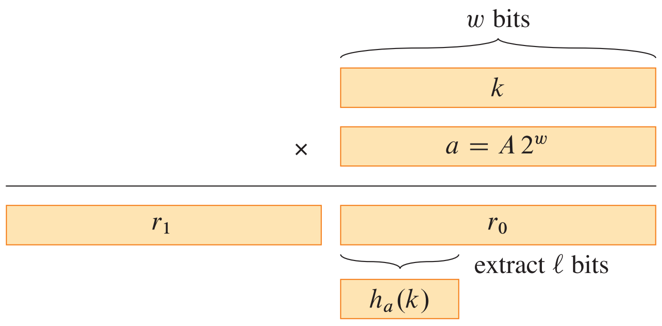


Figure 11.4 The multiply-shift method to compute a hash function. The w-bit representation of the key k is multiplied by the w-bit value $a = A \cdot 2^w$. The ℓ highest-order bits of the lower w-bit half of the product form the desired hash value $h_a(k)$.

The multiply-shift method

In practice, the multiplication method is best in the special case where the number m of hash-table slots is an exact power of 2, so that $m = 2^\ell$ for some integer ℓ, where $\ell \le w$ and w is the number of bits in a machine word. If you choose a fixed w-bit positive integer $a = A\,2^w$, where $0 < A < 1$ as in the multiplication method so that a is in the range $0 < a < 2^w$, you can implement the function on most computers as follows. We assume that a key k fits into a single w-bit word.

Referring to Figure 11.4, first multiply k by the w-bit integer a. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order w-bit word of the product and r_0 is the low-order w-bit word of the product. The desired ℓ-bit hash value consists of the ℓ most significant bits of r_0. (Since r_1 is ignored, the hash function can be implemented on a computer that produces only a w-bit product given two w-bit inputs, that is, where the multiplication operation computes modulo 2^w.)

In other words, you define the hash function $h = h_a$, where

$$h_a(k) = (ka \bmod 2^w) \ggg (w - \ell) \tag{11.2}$$

for a fixed nonzero w-bit value a. Since the product ka of two w-bit words occupies $2w$ bits, taking this product modulo 2^w zeroes out the high-order w bits (r_1), leaving only the low-order w bits (r_0). The $\ggg$ operator performs a logical right shift by $w - \ell$ bits, shifting zeros into the vacated positions on the left, so that the ℓ most significant bits of r_0 move into the ℓ rightmost positions. (It's the same as dividing by $2^{w-\ell}$ and taking the floor of the result.) The resulting value equals the ℓ most significant bits of r_0. The hash function h_a can be implemented with three machine instructions: multiplication, subtraction, and logical right shift.

As an example, suppose that $k = 123456$, $\ell = 14$, $m = 2^{14} = 16384$, and $w = 32$. Suppose further that we choose $a = 2654435769$ (following a suggestion of Knuth [261]). Then $ka = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h_a(k) = 67$.

Even though the multiply-shift method is fast, it doesn't provide any guarantee of good average-case performance. The universal hashing approach presented in the next section provides such a guarantee. A simple randomized variant of the multiply-shift method works well on the average, when the program begins by picking a as a randomly chosen odd integer.

11.3.2 Random hashing

Suppose that a malicious adversary chooses the keys to be hashed by some fixed hash function. Then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any static hash function is vulnerable to such terrible worst-case behavior. The only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach is called ***random hashing***. A special case of this approach, called ***universal hashing***, can yield provably good performance on average when collisions are handled by chaining, no matter which keys the adversary chooses.

To use random hashing, at the beginning of program execution you select the hash function at random from a suitable family of functions. As in the case of quicksort, randomization guarantees that no single input always evokes worst-case behavior. Because you randomly select the hash function, the algorithm can behave differently on each execution, even for the same set of keys to be hashed, guaranteeing good average-case performance.

Let $\mathcal{H}$ be a finite family of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a family is said to be ***universal*** if for each pair of distinct keys $k_1, k_2 \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k_1) = h(k_2)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k_1 and k_2 is no more than the chance $1/m$ of a collision if $h(k_1)$ and $h(k_2)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.

Independent uniform hashing is the same as picking a hash function uniformly at random from a family of m^n hash functions, each member of that family mapping the n keys to the m hash values in a different way.

Every independent uniform random family of hash function is universal, but the converse need not be true: consider the case where $U = \{0, 1, \ldots, m-1\}$ and the only hash function in the family is the identity function. The probability that two distinct keys collide is zero, even though each key is hashes to a fixed value.

The following corollary to Theorem 11.2 on page 279 says that universal hashing provides the desired payoff: it becomes impossible for an adversary to pick a sequence of operations that forces the worst-case running time.

Corollary 11.3
Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes $\Theta(s)$ expected time to handle any sequence of s INSERT, SEARCH, and DELETE operations containing $n = O(m)$ INSERT operations.

Proof The INSERT and DELETE operations take constant time. Since the number n of insertions is $O(m)$, we have that $\alpha = O(1)$. Furthermore, the expected time for each SEARCH operation is $O(1)$, which can be seen by examining the proof of Theorem 11.2. That analysis depends only on collision probabilities, which are $1/m$ for any pair k_1, k_2 of keys by the choice of an independent uniform hash function in that theorem. Using a universal family of hash functions here instead of using independent uniform hashing changes the probability of collision from $1/m$ to at most $1/m$. By linearity of expectation, therefore, the expected time for the entire sequence of s operations is $O(s)$. Since each operation takes $\Omega(1)$ time, the $\Theta(s)$ bound follows. ■

11.3.3 Achievable properties of random hashing

There is a rich literature on the properties a family $\mathcal{H}$ of hash functions can have, and how they relate to the efficiency of hashing. We summarize a few of the most interesting ones here.

Let $\mathcal{H}$ be a family of hash functions, each with domain U and range $\{0, 1, \ldots, m-1\}$, and let h be any hash function that is picked uniformly at random from $\mathcal{H}$. The probabilities mentioned are probabilities over the picks of h.

- The family $\mathcal{H}$ is ***uniform*** if for any key k in U and any slot q in the range $\{0, 1, \ldots, m-1\}$, the probability that $h(k) = q$ is $1/m$.
- The family $\mathcal{H}$ is ***universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most $1/m$.
- The family $\mathcal{H}$ of hash functions is ***ϵ-universal*** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$ is at most ϵ. Therefore, a universal family of hash functions is also $1/m$-universal.[2]

[2] In the literature, a (c/m)-universal hash function is sometimes called c-universal or c-approximately universal. We'll stick with the notation (c/m)-universal.

- The family $\mathcal{H}$ is ***d*-independent** if for any distinct keys $k_1, k_2, \ldots, k_d$ in U and any slots $q_1, q_2, \ldots, q_d$, not necessarily distinct, in $\{0, 1, \ldots, m-1\}$ the probability that $h(k_i) = q_i$ for $i = 1, 2, \ldots, d$ is $1/m^d$.

Universal hash-function families are of particular interest, as they are the simplest type supporting provably efficient hash-table operations for any input data set. Many other interesting and desirable properties, such as those noted above, are also possible and allow for efficient specialized hash-table operations.

11.3.4 Designing a universal family of hash functions

This section present two ways to design a universal (or ϵ-universal) family of hash functions: one based on number theory and another based on a randomized variant of the multiply-shift method presented in Section 11.3.1. The first method is a bit easier to prove universal, but the second method is newer and faster in practice.

A universal family of hash functions based on number theory

We can design a universal family of hash functions using a little number theory. You may wish to refer to Chapter 31 if you are unfamiliar with basic concepts in number theory.

Begin by choosing a prime number p large enough so that every possible key k lies in the range 0 to $p-1$, inclusive. We assume here that p has a "reasonable" length. (See Section 11.3.5 for a discussion of methods for handling long input keys, such as variable-length strings.) Let $\mathbb{Z}_p$ denote the set $\{0, 1, \ldots, p-1\}$, and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p-1\}$. Since p is prime, we can solve equations modulo p with the methods given in Chapter 31. Because the size of the universe of keys is greater than the number of slots in the hash table (otherwise, just use direct addressing), we have $p > m$.

Given any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$, define the hash function h_{ab} as a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m . \tag{11.3}$$

For example, with $p = 17$ and $m = 6$, we have

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 . \end{aligned}$$

Given p and m, the family of all such hash functions is

$$\mathcal{H}_{pm} = \left\{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\right\} . \tag{11.4}$$

Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$. This family of hash functions has the nice property that the size m of the output range (which is the size of the hash table) is arbitrary—it need not be prime. Since you can choose from among $p-1$ values for a and p values for b, the family $\mathcal{H}_{pm}$ contains $p(p-1)$ hash functions.

Theorem 11.4
The family $\mathcal{H}_{pm}$ of hash functions defined by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k_1 and k_2 from $\mathbb{Z}_p$, so that $k_1 \neq k_2$. For a given hash function h_{ab}, let

$$\begin{aligned} r_1 &= (ak_1 + b) \bmod p \ , \\ r_2 &= (ak_2 + b) \bmod p \ . \end{aligned}$$

We first note that $r_1 \neq r_2$. Why? Since we have $r_1 - r_2 = a(k_1 - k_2) \pmod p$, it follows that $r_1 \neq r_2$ because p is prime and both a and $(k_1 - k_2)$ are nonzero modulo p. By Theorem 31.6 on page 908, their product must also be nonzero modulo p. Therefore, when computing any $h_{ab} \in \mathcal{H}_{pm}$, distinct inputs k_1 and k_2 map to distinct values r_1 and r_2 modulo p, and there are no collisions yet at the "mod p level." Moreover, each of the possible $p(p-1)$ choices for the pair (a, b) with $a \neq 0$ yields a *different* resulting pair (r_1, r_2) with $r_1 \neq r_2$, since we can solve for a and b given r_1 and r_2:

$$\begin{aligned} a &= \big((r_1 - r_2)((k_1 - k_2)^{-1} \bmod p)\big) \bmod p \ , \\ b &= (r_1 - ak_1) \bmod p \ , \end{aligned}$$

where $((k_1 - k_2)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p, of $k_1 - k_2$. For each of the p possible values of r_1, there are only $p-1$ possible values of r_2 that do not equal r_1, making only $p(p-1)$ possible pairs (r_1, r_2) with $r_1 \neq r_2$. Therefore, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r_1, r_2) with $r_1 \neq r_2$. Thus, for any given pair of distinct inputs k_1 and k_2, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r_1, r_2) is equally likely to be any pair of distinct values modulo p.

Therefore, the probability that distinct keys k_1 and k_2 collide is equal to the probability that $r_1 = r_2 \pmod m$ when r_1 and r_2 are randomly chosen as distinct values modulo p. For a given value of r_1, of the $p-1$ possible remaining values for r_2, the number of values r_2 such that $r_2 \neq r_1$ and $r_2 = r_1 \pmod m$ is at most

$$\begin{aligned} \left\lceil \frac{p}{m} \right\rceil - 1 &\leq \frac{p+m-1}{m} - 1 \quad \text{(by inequality (3.7) on page 64)} \\ &= \frac{p-1}{m} \ . \end{aligned}$$

The probability that r_2 collides with r_1 when reduced modulo m is at most $((p-1)/m)/(p-1) = 1/m$, since r_2 is equally likely to be any of the $p-1$ values in Z_p that are different from r_1, but at most $(p-1)/m$ of those values are equivalent to r_1 modulo m.

Therefore, for any pair of distinct values $k_1, k_2 \in \mathbb{Z}_p$,

$$\Pr\{h_{ab}(k_1) = h_{ab}(k_2)\} \le 1/m ,$$

so that $\mathcal{H}_{pm}$ is indeed universal. ■

A $2/m$-universal family of hash functions based on the multiply-shift method

We recommend that in practice you use the following hash-function family based on the multiply-shift method. It is exceptionally efficient and (although we omit the proof) provably $2/m$-universal. Define $\mathcal{H}$ to be the family of multiply-shift hash functions with odd constants a:

$$\mathcal{H} = \{h_a : a \text{ is odd}, 1 \le a < m, \text{and } h_a \text{ is defined by equation (11.2)}\} . \tag{11.5}$$

Theorem 11.5
The family of hash functions $\mathcal{H}$ given by equation (11.5) is $2/m$-universal. ■

That is, the probability that any two distinct keys collide is at most $2/m$. In many practical situations, the speed of computing the hash function more than compensates for the higher upper bound on the probability that two distinct keys collide when compared with a universal hash function.

11.3.5 Hashing long inputs such as vectors or strings

Sometimes hash function inputs are so long that they cannot be easily encoded modulo a reasonably sized prime number p or encoded within a single word of, say, 64 bits. As an example, consider the class of vectors, such as vectors of 8-bit bytes (which is how strings in many programming languages are stored). A vector might have an arbitrary nonnegative length, in which case the length of the input to the hash function may vary from input to input.

Number-theoretic approaches

One way to design good hash functions for variable-length inputs is to extend the ideas used in Section 11.3.4 to design universal hash functions. Exercise 11.3-6 explores one such approach.

Cryptographic hashing

Another way to design a good hash function for variable-length inputs is to use a hash function designed for cryptographic applications. ***Cryptographic hash functions*** are complex pseudorandom functions, designed for applications requiring properties beyond those needed here, but are robust, widely implemented, and usable as hash functions for hash tables.

A cryptographic hash function takes as input an arbitrary byte string and returns a fixed-length output. For example, the NIST standard deterministic cryptographic hash function SHA-256 [346] produces a 256-bit (32-byte) output for any input.

Some chip manufacturers include instructions in their CPU architectures to provide fast implementations of some cryptographic functions. Of particular interest are instructions that efficiently implement rounds of the Advanced Encryption Standard (AES), the "AES-NI" instructions. These instructions execute in a few tens of nanoseconds, which is generally fast enough for use with hash tables. A message authentication code such as CBC-MAC based on AES and the use of the AES-NI instructions could be a useful and efficient hash function. We don't pursue the potential use of specialized instruction sets further here.

Cryptographic hash functions are useful because they provide a way of implementing an approximate version of a random oracle. As noted earlier, a random oracle is equivalent to an independent uniform hash function family. From a theoretical point of view, a random oracle is an unachievable ideal: a deterministic function that provides a randomly selected output for each input. Because it is deterministic, it provides the same output if queried again for the same input. From a practical point of view, constructions of hash function families based on cryptographic hash functions are sensible substitutes for random oracles.

There are many ways to use a cryptographic hash function as a hash function. For example, we could define

$$h(k) = \text{SHA-256}(k) \bmod m \; .$$

To define a family of such hash functions one may prepend a "salt" string a to the input before hashing it, as in

$$h_a(k) = \text{SHA-256}(a \parallel k) \bmod m \; ,$$

where $a \parallel k$ denotes the string formed by concatenating the strings a and k. The literature on message authentication codes (MACs) provides additional approaches.

Cryptographic approaches to hash-function design are becoming more practical as computers arrange their memories in hierarchies of differing capacities and speeds. Section 11.5 discusses one hash-function design based on the RC6 encryption method.

Exercises

11.3-1

You wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might you take advantage of the hash values when searching the list for an element with a given key?

11.3-2

You hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. You can represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can you apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-3

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string x can be converted to string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys $61, 62, 63, 64$, and 65 are mapped.

★ ***11.3-5***

Show that any ϵ-universal family $\mathcal{H}$ of hash functions from a finite set U to a finite set Q has $\epsilon \geq 1/|Q| - 1/|U|$.

★ ***11.3-6***

Let U be the set of d-tuples of values drawn from $\mathbb{Z}_p$, and let $Q = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \to Q$ for $b \in \mathbb{Z}_p$ on an input d-tuple $\langle a_0, a_1, \ldots, a_{d-1} \rangle$ from U as

$$h_b(\langle a_0, a_1, \ldots, a_{d-1} \rangle) = \left(\sum_{j=0}^{d-1} a_j b^j \right) \bmod p \;,$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that $\mathcal{H}$ is ϵ-universal for $\epsilon = (d-1)/p$. (*Hint:* See Exercise 31.4-4.)

11.4 Open addressing

This section describes open addressing, a method for collision resolution that, unlike chaining, does not make use of storage outside of the hash table itself. In ***open addressing***, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. No lists or elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made. One consequence is that the load factor α can never exceed 1.

Collisions are handled as follows: when a new element is to be inserted into the table, it is placed in its "first-choice" location if possible. If that location is already occupied, the new element is placed in its "second-choice" location. The process continues until an empty slot is found in which to place the new element. Different elements have different preference orders for the locations.

To search for an element, systematically examine the preferred table slots for that element, in order of decreasing preference, until either you find the desired element or you find an empty slot and thus verify that the element is not in the table.

Of course, you could use chaining and store the linked lists inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, you compute the sequence of slots to be examined. The memory freed by not storing pointers provides the hash table with a larger number of slots in the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, successively examine, or ***probe***, the hash table until you find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \ldots, m-1$ (which implies a $\Theta(n)$ search time), the sequence of positions probed depends upon the key being inserted. To determine which slots to probe, the hash function includes the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} \ .$$

Open addressing requires that for every key k, the ***probe sequence*** $\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m-1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. The HASH-INSERT procedure on the following page assumes that the elements in the hash table T are keys with no satellite information: the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k

that is assumed to be not already present in the hash table. It either returns the slot number where it stores key k or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == NIL
        T[q] = k
        return q
    else i = i + 1
until i == m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i = 0
repeat
    q = h(k, i)
    if T[q] == k
        return q
    i = i + 1
until T[q] == NIL or i == m
return NIL
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. The procedure HASH-SEARCH takes as input a hash table T and a key k, returning q if it finds that slot q contains key k, or NIL if key k is not present in table T.

Deletion from an open-address hash table is tricky. When you delete a key from slot q, it would be a mistake to mark that slot as empty by simply storing NIL in it. If you did, you might be unable to retrieve any key k for which slot q was probed and found occupied when k was inserted. One way to solve this problem is by marking the slot, storing in it the special value DELETED instead of NIL. The HASH-INSERT procedure then has to treat such a slot as empty so that it can insert a new key there. The HASH-SEARCH procedure passes over DELETED values while searching, since slots containing DELETED were filled when the key being searched for was inserted. Using the special value DELETED, however, means that search times no longer depend on the load factor α, and for this reason chaining is frequently selected as a collision resolution technique when keys must be deleted. There is a simple special case of open addressing, linear probing, that avoids the need to mark slots with DELETED. Section 11.5.1 shows how to delete from a hash table when using linear probing.

In our analysis, we assume ***independent uniform permutation hashing*** (also confusingly known as ***uniform hashing*** in the literature): the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$. Independent uniform permutation hashing generalizes the notion of independent uniform hashing defined earlier to a hash function that produces not just a single slot number, but a whole probe sequence. True independent uniform permutation hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We'll examine both double hashing and its special case, linear probing. These techniques guarantee that $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ for each key k. (Recall that the second parameter to the hash function h is the probe number.) Neither double hashing nor linear probing meets the assumption of independent uniform permutation hashing, however. Double hashing cannot generate more than m^2 different probe sequences (instead of the $m!$ that independent uniform permutation hashing requires). Nonetheless, double hashing has a large number of possible probe sequences and, as you might expect, seems to give good results. Linear probing is even more restricted, capable of generating only m different probe sequences.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m ,$$

where both h_1 and h_2 are ***auxiliary hash functions***. The initial probe goes to position $T[h_1(k)]$, and successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus, the probe sequence here depends in two ways upon the key k, since the initial probe position $h_1(k)$, the step size $h_2(k)$, or both, may vary. Figure 11.5 gives an example of insertion by double hashing.

In order for the entire hash table to be searched, the value $h_2(k)$ must be relatively prime to the hash-table size m. (See Exercise 11.4-5.) A convenient way to ensure this condition is to let m be an exact power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, you

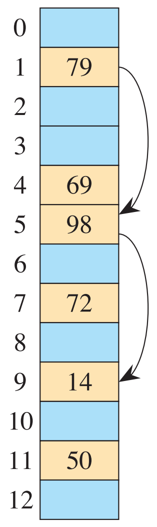


Figure 11.5 Insertion by double hashing. The hash table has size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 = 1 \pmod{13}$ and $14 = 3 \pmod{11}$, the key 14 goes into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

could choose m prime and let

$$h_1(k) = k \bmod m ,$$
$$h_2(k) = 1 + (k \bmod m') ,$$

where m' is chosen to be slightly less than m (say, $m - 1$). For example, if $k = 123456$, $m = 701$, and $m' = 700$, then $h_1(k) = 80$ and $h_2(k) = 257$, so that the first probe goes to position 80, and successive probes examine every 257th slot (modulo m) until the key has been found or every slot has been examined.

Although values of m other than primes or exact powers of 2 can in principle be used with double hashing, in practice it becomes more difficult to efficiently generate $h_2(k)$ (other than choosing $h_2(k) = 1$, which gives linear probing) in a way that ensures that it is relatively prime to m, in part because the relative density $\phi(m)/m$ of such numbers for general m may be small (see equation (31.25) on page 921).

When m is prime or an exact power of 2, double hashing produces $\Theta(m^2)$ probe sequences, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, for such values of m, double hashing appears to perform close to the "ideal" scheme of independent uniform permutation hashing.

Linear probing

Linear probing, a special case of double hashing, is the simplest open-addressing approach to resolving collisions. As with double hashing, an auxiliary hash function h_1 determines the first probe position $h_1(k)$ for inserting an element. If slot $T[h_1(k)]$ is already occupied, probe the next position $T[h_1(k)+1]$. Keep going as necessary, on up to slot $T[m-1]$, and then wrap around to slots $T[0]$, $T[1]$, and so on, but never going past slot $T[h_1(k)-1]$. To view linear probing as a special case of double hashing, just set the double-hashing step function h_2 to be fixed at 1: $h_2(k) = 1$ for all k. That is, the hash function is

$$h(k,i) = (h_1(k) + i) \bmod m \tag{11.6}$$

for $i = 0, 1, \ldots, m-1$. The value of $h_1(k)$ determines the entire probe sequence, and so assuming that $h_1(k)$ can take on any value in $\{0, 1, \ldots, m-1\}$, linear probing allows only m distinct probe sequences.

We'll revisit linear probing in Section 11.5.1.

Analysis of open-address hashing

As in our analysis of chaining in Section 11.2, we analyze open addressing in terms of the load factor $\alpha = n/m$ of the hash table. With open addressing, at most one element occupies each slot, and thus $n \le m$, which implies $\alpha \le 1$. The analysis below requires α to be strictly less than 1, and so we assume that at least one slot is empty. Because deleting from an open-address hash table does not really free up a slot, we assume as well that no deletions occur.

For the hash function, we assume independent uniform permutation hashing. In this idealized scheme, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \ldots, m-1\rangle$. Of course, any given key has a unique fixed probe sequence associated with it. What we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of independent uniform permutation hashing, beginning with the expected number of probes made in an unsuccessful search (assuming, as stated above, that $\alpha < 1$).

The bound proven, of $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$, has an intuitive interpretation. The first probe always occurs. With probability approximately α, the first probe finds an occupied slot, so that a second probe happens. With probability approximately α^2, the first two slots are occupied so that a third probe ensues, and so on.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming independent uniform permutation hashing and no deletions.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let the random variable X denote the number of probes made in an unsuccessful search, and define the event A_i, for $i = 1, 2, \ldots$, as the event that an ith probe occurs and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$. We bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$. By Exercise C.2-5 on page 1190,

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} \ .\end{aligned}$$

Since there are n elements and m slots, $\Pr\{A_1\} = n/m$. For $j > 1$, the probability that there is a jth probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$. This probability follows because the jth probe would be finding one of the remaining $(n-(j-1))$ elements in one of the $(m-(j-1))$ unexamined slots, and by the assumption of independent uniform permutation hashing, the probability is the ratio of these quantities. Since $n < m$ implies that $(n-j)/(m-j) \leq n/m$ for all j in the range $0 \leq j < m$, it follows that for all i in the range $1 \leq i \leq m$, we have

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \ .\end{aligned}$$

The product in the first line has $i-1$ factors. When $i = 1$, the product is 1, the identity for multiplication, and we get $\Pr\{X \geq 1\} = 1$, which makes sense, since there must always be at least 1 probe. If each of the first n probes is to an occupied slot, then all occupied slots have been probed. Then, the $(n+1)$st probe must be to an empty slot, which gives $\Pr\{X \geq i\} = 0$ for $i > n+1$. Now, we use equation (C.28) on page 1193 to bound the expected number of probes:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &= \sum_{i=1}^{n+1} \Pr\{X \geq i\} + \sum_{i>n+1} \Pr\{X \geq i\}\end{aligned}$$

$$
\begin{aligned}
&\le \sum_{i=1}^{\infty} \alpha^{i-1} + 0 \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha} \qquad \text{(by equation (A.7) on page 1142 because } 0 \le \alpha < 1) . \qquad \blacksquare
\end{aligned}
$$

If α is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1-.5) = 2$. If it is 90% full, the average number of probes is at most $1/(1-.9) = 10$.

Theorem 11.6 yields almost immediately how well the HASH-INSERT procedure performs.

Corollary 11.7
Inserting an element into an open-address hash table with load factor α, where $\alpha < 1$, requires at most $1/(1-\alpha)$ probes on average, assuming independent uniform permutation hashing and no deletions.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1-\alpha)$. ■

It takes a little more work to compute the expected number of probes for a successful search.

Theorem 11.8
Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} ,$$

assuming independent uniform permutation hashing with no deletions and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k reproduces the same probe sequence as when the element with key k was inserted. If k was the $(i+1)$st key inserted into the hash table, then the load factor at the time it was inserted was i/m, and so by Corollary 11.7, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$. Averaging over all n keys in the hash table gives us the expected number of probes in a successful search:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}\,dx \qquad \text{(by inequality (A.19) on page 1150)} \\
&= \frac{1}{\alpha}(\ln m - \ln(m-n)) \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\,. \qquad \blacksquare
\end{aligned}$$

If the hash table is half full, the expected number of probes in a successful search is less than 1.387. If the hash table is 90% full, the expected number of probes is less than 2.559. If $\alpha = 1$, then in an unsuccessful search, all m slots must be probed. Exercise 11.4-4 asks you to analyze a successful search when $\alpha = 1$.

Exercises

11.4-1

Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using open addressing. Illustrate the result of inserting these keys using linear probing with $h(k,i) = (k+i) \bmod m$ and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m-1))$.

11.4-2

Write pseudocode for HASH-DELETE that fills the deleted key's slot with the special value DELETED, and modify HASH-SEARCH and HASH-INSERT as needed to handle DELETED.

11.4-3

Consider an open-address hash table with independent uniform permutation hashing and no deletions. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

11.4-4
Show that the expected number of probes required for a successful search when $\alpha = 1$ (that is, when $n = m$), is H_m, the mth harmonic number.

★ ***11.4-5***
Show that, with double hashing, if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k, then an unsuccessful search for key k examines $(1/d)$th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (*Hint:* See Chapter 31.)

★ ***11.4-6***
Consider an open-address hash table with a load factor α. Approximate the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

11.5 Practical considerations

Efficient hash table algorithms are not only of theoretical interest, but also of immense practical importance. Constant factors can matter. For this reason, this section discusses two aspects of modern CPUs that are not included in the standard RAM model presented in Section 2.2:

Memory hierarchies: The memory of modern CPUs has a number of levels, from the fast registers, through one or more levels of ***cache memory***, to the main-memory level. Each successive level stores more data than the previous level, but access is slower. As a consequence, a complex computation (such as a complicated hash function) that works entirely within the fast registers can take less time than a single read operation from main memory. Furthermore, cache memory is organized in ***cache blocks*** of (say) 64 bytes each, which are always fetched together from main memory. There is a substantial benefit for ensuring that memory usage is local: reusing the same cache block is much more efficient than fetching a different cache block from main memory.

The standard RAM model measures efficiency of a hash-table operation by counting the number of hash-table slots probed. In practice, this metric is only a crude approximation to the truth, since once a cache block is in the cache, successive probes to that cache block are much faster than probes that must access main memory.

Advanced instruction sets: Modern CPUs may have sophisticated instruction sets that implement advanced primitives useful for encryption or other forms of cryptography. These instructions may be useful in the design of exceptionally efficient hash functions.

Section 11.5.1 discusses linear probing, which becomes the collision-resolution method of choice in the presence of a memory hierarchy. Section 11.5.2 suggests how to construct "advanced" hash functions based on cryptographic primitives, suitable for use on computers with hierarchical memory models.

11.5.1 Linear probing

Linear probing is often disparaged because of its poor performance in the standard RAM model. But linear probing excels for hierarchical memory models, because successive probes are usually to the same cache block of memory.

Deletion with linear probing

Another reason why linear probing is often not used in practice is that deletion seems complicated or impossible without using the special DELETED value. Yet we'll now see that deletion from a hash table based on linear probing is not all that difficult, even without the DELETED marker. The deletion procedure works for linear probing, but not for open-address probing in general, because with linear probing keys all follow the same simple cyclic probing sequence (albeit with different starting points).

The deletion procedure relies on an "inverse" function to the linear-probing hash function $h(k, i) = (h_1(k) + i) \bmod m$, which maps a key k and a probe number i to a slot number in the hash table. The inverse function g maps a key k and a slot number q, where $0 \le q < m$, to the probe number that reaches slot q:

$$g(k, q) = (q - h_1(k)) \bmod m \; .$$

If $h(k, i) = q$, then $g(k, q) = i$, and so $h(k, g(k, q)) = q$.

The procedure LINEAR-PROBING-HASH-DELETE on the facing page deletes the key stored in position q from hash table T. Figure 11.6 shows how it works. The procedure first deletes the key in position q by setting $T[q]$ to NIL in line 2. It then searches for a slot q' (if any) that contains a key that should be moved to the slot q just vacated by key k. Line 9 asks the critical question: does the key k' in slot q' need to be moved to the vacated slot q in order to preserve the accessibility of k'? If $g(k', q) < g(k', q')$, then during the insertion of k' into the table, slot q was examined but found to be already occupied. But now slot q, where a search will look for k', is empty. In this case, key k' moves to slot q in line 10, and the

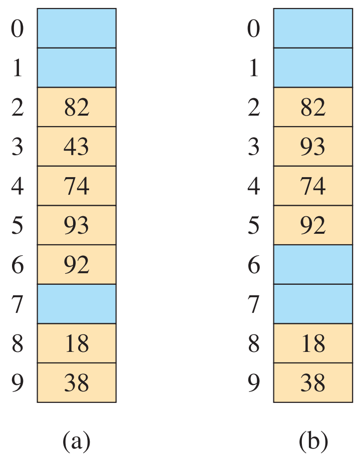


Figure 11.6 Deletion in a hash table that uses linear probing. The hash table has size 10 with $h_1(k) = k \bmod 10$. **(a)** The hash table after inserting keys in the order 74, 43, 93, 18, 82, 38, 92. **(b)** The hash table after deleting the key 43 from slot 3. Key 93 moves up to slot 3 to keep it accessible, and then key 92 moves up to slot 5 just vacated by key 93. No other keys need to be moved.

search continues, to see whether any later key also needs to be moved to the slot q' that was just freed up when k' moved.

```
LINEAR-PROBING-HASH-DELETE$(T, q)$
while TRUE
    $T[q]$ = NIL                        // make slot $q$ empty
    $q' = q$                            // starting point for search
    repeat
        $q' = (q' + 1) \bmod m$         // next slot number with linear probing
        $k' = T[q']$                    // next key to try to move
        if $k'$ == NIL
            return                      // return when an empty slot is found
    until $g(k', q) < g(k', q')$        // was empty slot $q$ probed before $q'$?
    $T[q] = k'$                         // move $k'$ into slot $q$
    $q = q'$                            // free up slot $q'$
```

Analysis of linear probing

Linear probing is popular to implement, but it exhibits a phenomenon known as ***primary clustering***. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i + 1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

In the standard RAM model, primary clustering is a problem, and general double hashing usually performs better than linear probing. By contrast, in a hierarchical memory model, primary clustering is a beneficial property, as elements are often stored together in the same cache block. Searching proceeds through one cache block before advancing to search the next cache block. With linear probing, the running time for a key k of HASH-INSERT, HASH-SEARCH, or LINEAR-PROBING-HASH-DELETE is at most proportional to the distance from $h_1(k)$ to the next empty slot.

The following theorem is due to Pagh et al. [351]. A more recent proof is given by Thorup [438]. We omit the proof here. The need for 5-independence is by no means obvious; see the cited proofs.

Theorem 11.9
If h_1 is 5-independent and $\alpha \le 2/3$, then it takes expected constant time to search for, insert, or delete a key in a hash table using linear probing. ■

(Indeed, the expected operation time is $O(1/\epsilon^2)$ for $\alpha = 1 - \epsilon$.)

★ 11.5.2 Hash functions for hierarchical memory models

This section illustrates an approach for designing efficient hash tables in a modern computer system having a memory hierarchy.

Because of the memory hierarchy, linear probing is a good choice for resolving collisions, as probe sequences are sequential and tend to stay within cache blocks. But linear probing is most efficient when the hash function is complex (for example, 5-independent as in Theorem 11.9). Fortunately, having a memory hierarchy means that complex hash functions can be implemented efficiently.

As noted in Section 11.3.5, one approach is to use a cryptographic hash function such as SHA-256. Such functions are complex and sufficiently random for hash table applications. On machines with specialized instructions, cryptographic functions can be quite efficient.

Instead, we present here a simple hash function based only on addition, multiplication, and swapping the halves of a word. This function can be implemented entirely within the fast registers, and on a machine with a memory hierarchy, its latency is small compared with the time taken to access a random slot of the hash table. It is related to the RC6 encryption algorithm and can for practical purposes be considered a "random oracle."

The wee hash function

Let w denote the word size of the machine (e.g., $w = 64$), assumed to be even, and let a and b be w-bit unsigned (nonnegative) integers such that a is odd. Let $\text{swap}(x)$ denote the w-bit result of swapping the two $w/2$-bit halves of w-bit input x. That is,

$$\text{swap}(x) = (x \ggg (w/2)) + (x \lll (w/2))$$

where "$\ggg$" is "logical right shift" (as in equation (11.2)) and "$\lll$ is "left shift." Define

$$f_a(k) = \text{swap}((2k^2 + ak) \bmod 2^w) \ .$$

Thus, to compute $f_a(k)$, evaluate the quadratic function $2k^2 + ak$ modulo 2^w and then swap the left and right halves of the result.

Let r denote a desired number of "rounds" for the computation of the hash function. We'll use $r = 4$, but the hash function is well defined for any nonnegative r. Denote by $f_a^{(r)}(k)$ the result of iterating f_a a total of r times (that is, r rounds) starting with input value k. For any odd a and any $r \geq 0$, the function $f_a^{(r)}$, although complicated, is one-to-one (see Exercise 11.5-1). A cryptographer would view $f_a^{(r)}$ as a simple block cipher operating on w-bit input blocks, with r rounds and key a.

We first define the wee hash function h for short inputs, where by "short" we means "whose length t is at most w-bits," so that the input fits within one computer word. We would like inputs of different lengths to be hashed differently. The ***wee hash function*** $h_{a,b,t,r}(k)$ for parameters a, b, and r on t-bit input k is defined as

$$h_{a,b,t,r}(k) = \left(f_{a+2t}^{(r)}(k + b)\right) \bmod m \ . \qquad (11.7)$$

That is, the hash value for t-bit input k is obtained by applying $f_{a+2t}^{(r)}$ to $k + b$, then taking the final result modulo m. Adding the value b provides hash-dependent randomization of the input, in a way that ensures that for variable-length inputs the 0-length input does not have a fixed hash value. Adding the value $2t$ to a ensures that the hash function acts differently for inputs of different lengths. (We use $2t$ rather than t to ensure that the key $a + 2t$ is odd if a is odd.) We call this hash function "wee" because it uses a tiny amount of memory—more precisely, it can be implemented efficiently using only the computer's fast registers. (This hash function does not have a name in the literature; it is a variant we developed for this textbook.)

Speed of the wee hash function

It is surprising how much efficiency can be bought with locality. Experiments (unpublished, by the authors) suggest that evaluating the wee hash function takes less time than probing a *single* randomly chosen slot in a hash table. These experiments were run on a laptop (2019 MacBook Pro) with $w = 64$ and $a = 123$. For large hash tables, evaluating the wee hash function was 2 to 10 times faster than performing a single probe of the hash table.

The wee hash function for variable-length inputs

Sometimes inputs are long—more than one w-bit word in length—or have variable length, as discussed in Section 11.3.5. We can extend the wee hash function, defined above for inputs that are at most single w-bit word in length, to handle long or variable-length inputs. Here is one method for doing so.

Suppose that an input k has length t (measured in bits). Break k into a sequence $\langle k_1, k_2, \ldots, k_u \rangle$ of w-bit words, where $u = \lceil t/w \rceil$, k_1 contains the least-significant w bits of k, and k_u contains the most significant bits. If t is not a multiple of w, then k_u contains fewer than w bits, in which case, pad out the unused high-order bits of k_u with 0-bits. Define the function chop to return a sequence of the w-bit words in k:

$$\text{chop}(k) = \langle k_1, k_2, \ldots, k_u \rangle \ .$$

The most important property of the chop operation is that it is one-to-one, given t: for any two t-bit keys k and k', if $k \neq k'$ then $\text{chop}(k) \neq \text{chop}(k')$, and k can be derived from $\text{chop}(k)$ and t. The chop operation also has the useful property that a single-word input key yields a single-word output sequence: $\text{chop}(k) = \langle k \rangle$.

With the chop function in hand, we specify the wee hash function $h_{a,b,t,r}(k)$ for an input k of length t bits as follows:

$$h_{a,b,t,r}(k) = \text{WEE}(k, a, b, t, r, m) \ ,$$

where the procedure WEE defined on the facing page iterates through the elements of the w-bit words returned by $\text{chop}(k)$, applying f_a^r to the sum of the current word k_i and the previously computed hash value so far, finally returning the result obtained modulo m. This definition for variable-length and long (multiple-word) inputs is a consistent extension of the definition in equation (11.7) for short (single-word) inputs. For practical use, we recommend that a be a randomly chosen odd w-bit word, b be a randomly chosen w-bit word, and that $r = 4$.

Note that the wee hash function is really a hash function family, with individual hash functions determined by parameters a, b, t, r, and m. The (approximate) 5-independence of the wee hash function family for variable-length inputs can be argued based on the assumption that the 1-word wee hash function is a random oracle and on the security of the cipher-block-chaining message authentication code (CBC-MAC), as studied by Bellare et al. [42]. The case here is actually simpler than that studied in the literature, since if two messages have different lengths t and t', then their "keys" are different: $a + 2t \neq a + 2t'$. We omit the details.

```
WEE($k, a, b, t, r, m$)
1  $u = \lceil t/w \rceil$
2  $\langle k_1, k_2, \ldots, k_u \rangle = \text{chop}(k)$
3  $q = b$
4  for $i = 1$ to $u$
5      $q = f^{(r)}_{a+2t}(k_i + q)$
6  return $q \bmod m$
```

This definition of a cryptographically inspired hash-function family is meant to be realistic, yet only illustrative, and many variations and improvements are possible. See the chapter notes for suggestions.

In summary, we see that when the memory system is hierarchical, it becomes advantageous to use linear probing (a special case of double hashing), since successive probes tend to stay in the same cache block. Furthermore, hash functions that can be implemented using only the computer's fast registers are exceptionally efficient, so they can be quite complex and even cryptographically inspired, providing the high degree of independence needed for linear probing to work most efficiently.

Exercises

★ ***11.5-1***

Complete the argument that for any odd positive integer a and any integer $r \geq 0$, the function $f_a^{(r)}$ is one-to-one. Use a proof by contradiction and make use of the fact that the function f_a works modulo 2^w.

★ ***11.5-2***

Argue that a random oracle is 5-independent.

★ ***11.5-3***

Consider what happens to the value $f_a^{(r)}(k)$ as we flip a single bit k_i of the input value k, for various values of r. Let $k = \sum_{i=0}^{w-1} k_i 2^i$ and $g_a(k) = \sum_{j=0}^{w-1} b_j 2^j$ define the bit values k_i in the input (with k_0 the least-significant bit) and the bit values b_j in $g_a(k) = (2k^2 + ak) \bmod 2^w$ (where $g_a(k)$ is the value that, when its halves are swapped, becomes $f_a(k)$). Suppose that flipping a single bit k_i of the input k may cause any bit b_j of $g_a(k)$ to flip, for $j \geq i$. What is the least value of r for which flipping the value of any single bit k_i may cause *any* bit of the output $f_a^{(r)}(k)$ to flip? Explain.

Problems

11-1 Longest-probe bound for hashing

Suppose you are using an open-addressed hash table of size m to store $n \le m/2$ items.

a. Assuming independent uniform permutation hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-p} that the ith insertion requires strictly more than p probes.

b. Show that for $i = 1, 2, \ldots, n$, the probability is $O(1/n^2)$ that the ith insertion requires more than $2 \lg n$ probes.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max\{X_i : 1 \le i \le n\}$ denote the maximum number of probes required by any of the n insertions.

c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.

d. Show that the expected length $\mathrm{E}[X]$ of the longest probe sequence is $O(\lg n)$.

11-2 Searching a static set

You are asked to implement a searchable set of n elements in which the keys are numbers. The set is static (no INSERT or DELETE operations), and the only operation required is SEARCH. You are given an arbitrary amount of time to preprocess the n elements so that SEARCH operations run quickly.

a. Show how to implement SEARCH in $O(\lg n)$ worst-case time using no extra storage beyond what is needed to store the elements of the set themselves.

b. Consider implementing the set by open-address hashing on m slots, and assume independent uniform permutation hashing. What is the minimum amount of extra storage $m - n$ required to make the average performance of an unsuccessful SEARCH operation be at least as good as the bound in part (a)? Your answer should be an asymptotic bound on $m - n$ in terms of n.

11-3 Slot-size bound for chaining

Given a hash table with n slots, with collisions resolved by chaining, suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $\mathrm{E}[M]$, the expected value of M.

a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \le n Q_k$.

c. Show that $Q_k < e^k / k^k$. *Hint:* Use Stirling's approximation, equation (3.25) on page 67.

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^{3}$ for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \ge k_0 = c \lg n / \lg \lg n$.

e. Argue that

$$\mathrm{E}[M] \le \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n} .$$

Conclude that $\mathrm{E}[M] = O(\lg n / \lg \lg n)$.

11-4 Hashing and authentication

Let $\mathcal{H}$ be a family of hash functions in which each hash function $h \in \mathcal{H}$ maps the universe U of keys to $\{0, 1, \ldots, m-1\}$.

a. Show that if the family $\mathcal{H}$ of hash functions is 2-independent, then it is universal.

b. Suppose that the universe U is the set of n-tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. For any n-tuple $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j\right) \bmod p .$$

Let $\mathcal{H} = \{h_a : a \in U\}$. Show that $\mathcal{H}$ is universal, but not 2-independent. (*Hint:* Find a key for which all hash functions in $\mathcal{H}$ produce the same value.)

c. Suppose that we modify $\mathcal{H}$ slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b\right) \bmod p$$

and $\mathcal{H}' = \{h'_{ab} : a \in U \text{ and } b \in \mathbb{Z}_p\}$. Argue that $\mathcal{H}'$ is 2-independent. (*Hint:* Consider fixed n-tuples $x \in U$ and $y \in U$, with $x_i \neq y_i$ for some i. What happens to $h'_{ab}(x)$ and $h'_{ab}(y)$ as a_i and b range over $\mathbb{Z}_p$?)

d. Alice and Bob secretly agree on a hash function h from a 2-independent family $\mathcal{H}$ of hash functions. Each $h \in \mathcal{H}$ maps from a universe of keys U to $\mathbb{Z}_p$, where p is prime. Later, Alice sends a message m to Bob over the internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t'). Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, even if the adversary knows the family $\mathcal{H}$ of hash functions used.

Chapter notes

The books by Knuth [261] and Gonnet and Baeza-Yates [193] are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing. The notion of a random oracle was introduced by Bellare et al. [43]. Carter and Wegman [80] introduced the notion of universal families of hash functions in 1979.

Dietzfelbinger et al. [113] invented the multiply-shift hash function and gave a proof of Theorem 11.5. Thorup [437] provides extensions and additional analysis. Thorup [438] gives a simple proof that linear probing with 5-independent hashing takes constant expected time per operation. Thorup also describes the method for deletion in a hash table using linear probing.

Fredman, Komlós, and Szemerédi [154] developed a perfect hashing scheme for static sets—"perfect" because all collisions are avoided. An extension of their method to dynamic sets, handling insertions and deletions in amortized expected time $O(1)$, has been given by Dietzfelbinger et al. [114].

The wee hash function is based on the RC6 encryption algorithm [379]. Leiserson et al. [292] propose an "RC6MIX" function that is essentially the same as the wee hash function. They give experimental evidence that it has good randomness, and they also give a "DOTMIX" function for dealing with variable-length inputs. Bellare et al. [42] provide an analysis of the security of the cipher-block-chaining message authentication code. This analysis implies that the wee hash function has the desired pseudorandomness properties.

12 Binary Search Trees

The search tree data structure supports each of the dynamic-set operations listed on page 250: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, you can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. In Chapter 13, we'll see a variation of binary search trees, red-black trees, whose operations guarantee a height of $O(\lg n)$. We won't prove it here, but if you build a binary search tree on a random set of n keys, its expected height is $O(\lg n)$ even if you don't try to limit its height.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. You can represent such a tree with a linked data structure, as in Section 10.3. In addition to a *key* and satellite data, each node object contains attributes *left*, *right*, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The tree itself has an attribute *root*

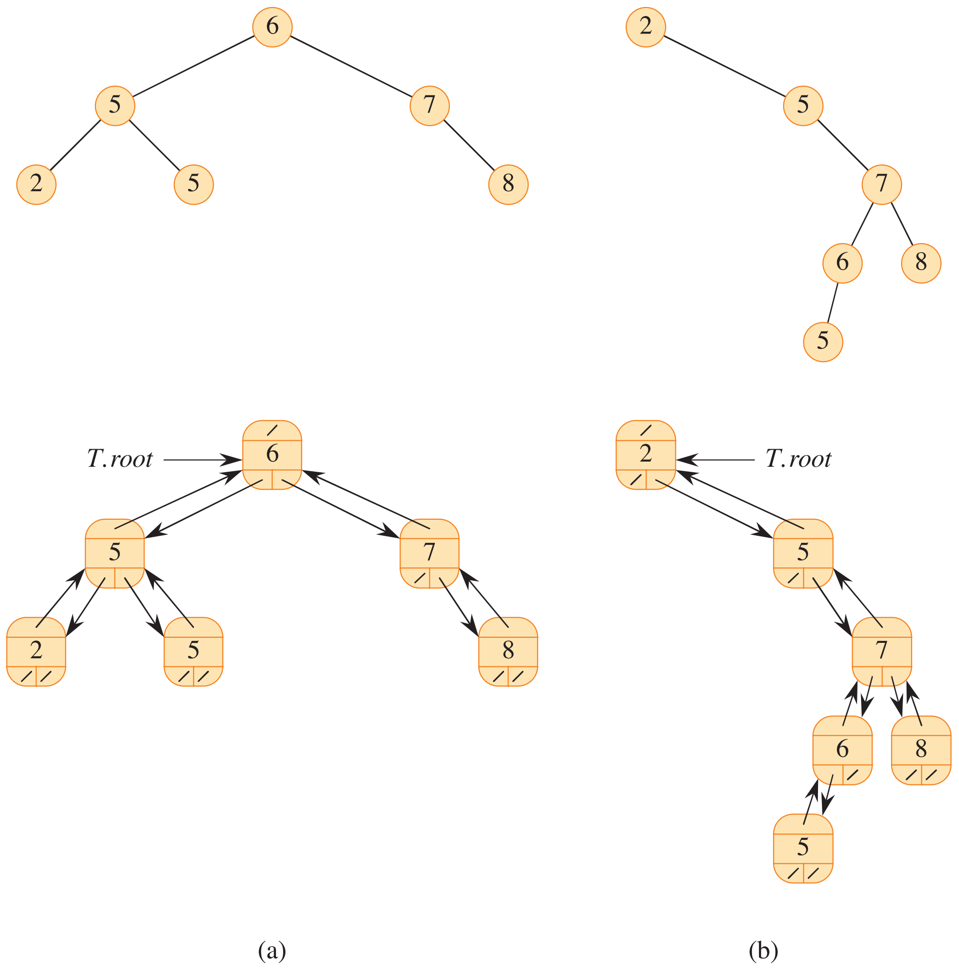


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. The top figure shows how to view the tree conceptually, and the bottom figure shows the *left*, *right*, and p attributes in each node, in the style of Figure 10.6 on page 266. **(b)** A less efficient binary search tree, with height 4, that contains the same keys.

that points to the root node, or NIL if the tree is empty. The root node $T.root$ is the only node in a tree T whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, looking at the root's left child as the root of a subtree, this subtree root has the key 5, the key 2 in its left subtree is no larger than 5, and the key 5 in its right subtree is no smaller than 5.

Because of the binary-search-tree property, you can print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***, given by the procedure INORDER-TREE-WALK. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To print all the elements in a binary search tree T, call INORDER-TREE-WALK$(T.root)$. For example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order $2, 5, 5, 6, 7, 8$. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

```
INORDER-TREE-WALK($x$)
1  if $x \neq$ NIL
2      INORDER-TREE-WALK($x.left$)
3      print $x.key$
4      INORDER-TREE-WALK($x.right$)
```

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq \text{NIL}$), we have $T(0) = c$ for some constant $c > 0$.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k) + T(n-k-1) + d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c + d)n + c$. For $n = 0$, we have $(c + d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &\leq ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c \,, \end{aligned}$$

which completes the proof. ■

Exercises

12.1-1

For the set $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

12.1-2

What is the difference between the binary-search-tree property and the min-heap property on page 163? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that you can test two pointers for equality.)

12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

12.1-5

Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

Binary search trees can support the queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR, as well as SEARCH. This section examines these operations and shows how to support each one in $O(h)$ time on any binary search tree of height h.

Searching

To search for a node with a given key in a binary search tree, call the TREE-SEARCH procedure. Given a pointer x to the root of a subtree and a key k, TREE-SEARCH(x, k) returns a pointer to a node with key k if one exists in the subtree; otherwise, it returns NIL. To search for key k in the entire binary search tree T, call TREE-SEARCH$(T.root, k)$.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

The TREE-SEARCH procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2(a). For each node x it encounters, it compares the key k with $x.key$. If the two keys are equal, the search terminates. If k is smaller than $x.key$, the search continues in the left subtree of x, since the binary-search-tree property implies that k cannot reside in the right subtree. Symmetrically, if k is larger than $x.key$, the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

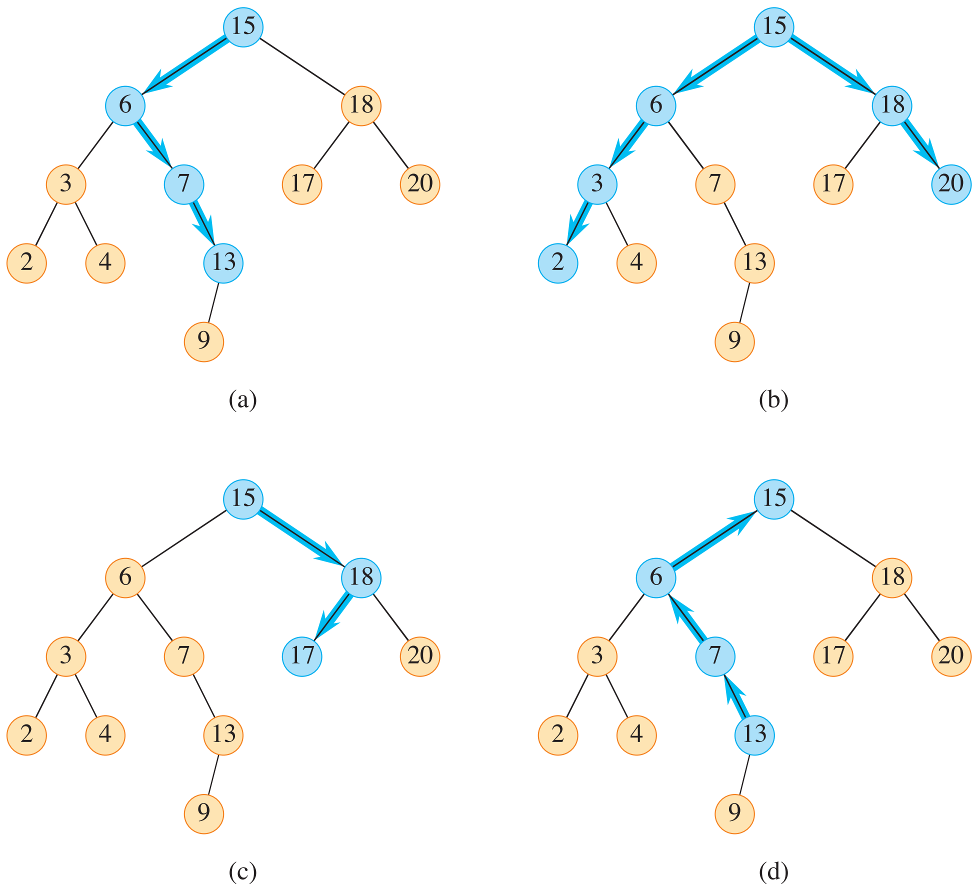


Figure 12.2 Queries on a binary search tree. Nodes and paths followed in each query are colored blue. **(a)** A search for the key 13 in the tree follows the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. **(b)** The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. **(c)** The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. **(d)** The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Since the TREE-SEARCH procedure recurses on either the left subtree or the right subtree, but not both, we can rewrite the algorithm to “unroll” the recursion into a **while** loop. On most computers, the ITERATIVE-TREE-SEARCH procedure on the facing page is more efficient.

Minimum and maximum

To find an element in a binary search tree whose key is a minimum, just follow *left* child pointers from the root until you encounter a NIL, as shown in Figure 12.2(b).

The TREE-MINIMUM procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL.

```
TREE-MINIMUM(x)
1  while $x.left \neq$ NIL
2      $x = x.left$
3  return $x$

TREE-MAXIMUM(x)
1  while $x.right \neq$ NIL
2      $x = x.right$
3  return $x$
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If node x has no left subtree, then since every key in the right subtree of x is at least as large as $x.key$, the minimum key in the subtree rooted at x is $x.key$. If node x has a left subtree, then since no key in the right subtree is smaller than $x.key$ and every key in the left subtree is not larger than $x.key$, the minimum key in the subtree rooted at x resides in the subtree rooted at $x.left$.

The pseudocode for TREE-MAXIMUM is symmetric. Both TREE-MINIMUM and TREE-MAXIMUM run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and predecessor

Given a node in a binary search tree, how can you find its successor in the sorted order determined by an inorder tree walk? If all keys are distinct, the successor of a node x is the node with the smallest key greater than $x.key$. Regardless of whether the keys are distinct, we define the ***successor*** of a node as the next node visited in an inorder tree walk. The structure of a binary search tree allows you to determine the successor of a node without comparing keys. The TREE-SUCCESSOR procedure on the facing page returns the successor of a node x in a binary search tree if it exists, or NIL if x is the last node that would be visited during an inorder walk.

The code for TREE-SUCCESSOR has two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree, which line 2 finds by calling TREE-MINIMUM$(x.right)$. For example, the successor of the node with key 15 in Figure 12.2(c) is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose

```
TREE-SUCCESSOR($x$)
if $x.right \neq$ NIL
    return TREE-MINIMUM($x.right$)    // leftmost node in right subtree
else // find the lowest ancestor of $x$ whose left child is an ancestor of $x$
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$
        $x = y$
        $y = y.p$
    return $y$
```

left child is also an ancestor of x. In Figure 12.2(d), the successor of the node with key 13 is the node with key 15. To find y, go up the tree from x until you encounter either the root or a node that is the left child of its parent. Lines 4–8 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since it either follows a simple path up the tree or follows a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in $O(h)$ time.

In summary, we have proved the following theorem.

Theorem 12.2
The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h. ∎

Exercises

12.2-1
You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences *cannot* be the sequence of nodes examined?

a. $2, 252, 401, 398, 330, 344, 397, 363$.

b. $924, 220, 911, 244, 898, 258, 362, 363$.

c. $925, 202, 911, 240, 912, 245, 363$.

d. $2, 399, 387, 219, 266, 382, 381, 278, 363$.

e. $935, 278, 347, 621, 299, 392, 358, 363$.

12.2-2
Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

12.2-3
Write the TREE-PREDECESSOR procedure.

12.2-4
Professor Kilmer claims to have discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up at a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Kilmer claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

12.2-5
Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6
Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

12.2-7
An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8
Prove that no matter what node you start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

12.2-9
Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. We'll see that modifying the tree to insert a new element is relatively straightforward, but deleting a node from a binary search tree is more complicated.

Insertion

The TREE-INSERT procedure inserts a new node into a binary search tree. The procedure takes a binary search tree T and a node z for which $z.key$ has already been filled in, $z.left = \text{NIL}$, and $z.right = \text{NIL}$. It modifies T and some of the attributes of z so as to insert z into an appropriate position in the tree.

```
TREE-INSERT($T, z$)
1   $x = T.root$                // node being compared with $z$
2   $y = \text{NIL}$            // $y$ will be parent of $z$
3   while $x \neq \text{NIL}$   // descend until reaching a leaf
4       $y = x$
5       if $z.key < x.key$
6           $x = x.left$
7       else $x = x.right$
8   $z.p = y$                   // found the location—insert $z$ with parent $y$
9   if $y == \text{NIL}$
10      $T.root = z$            // tree $T$ was empty
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
```

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input node z. The procedure maintains the ***trailing pointer*** y as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $z.key$ with $x.key$, until x becomes NIL. This NIL occupies the position where node z will go. More precisely, this NIL is a *left* or *right* attribute of the node that will become z's parent, or it is $T.root$ if tree T is currently empty. The procedure needs the

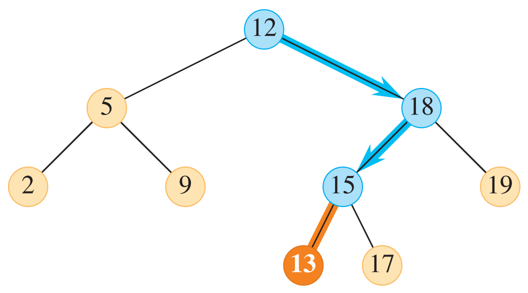


Figure 12.3 Inserting a node with key 13 into a binary search tree. The simple path from the root down to the position where the node is inserted is shown in blue. The new node and the link to its parent are highlighted in orange.

trailing pointer y, because by the time it finds the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases and, as we'll see, one of the cases is a bit tricky.

- If z has no children, then simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
- If z has two children, find z's successor y—which must belong to z's right subtree—and move y to take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. Because y is z's successor, it cannot have a left child, and y's original right child moves into y's original position, with the rest of y's original right subtree following automatically. This case is the tricky one because, as we'll see, it matters whether y is z's right child.

The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases shown in Figure 12.4.

- If z has no left child, then as in part (a) of the figure, replace z by its right child, which may or may not be NIL. When z's right child is NIL, this case deals with

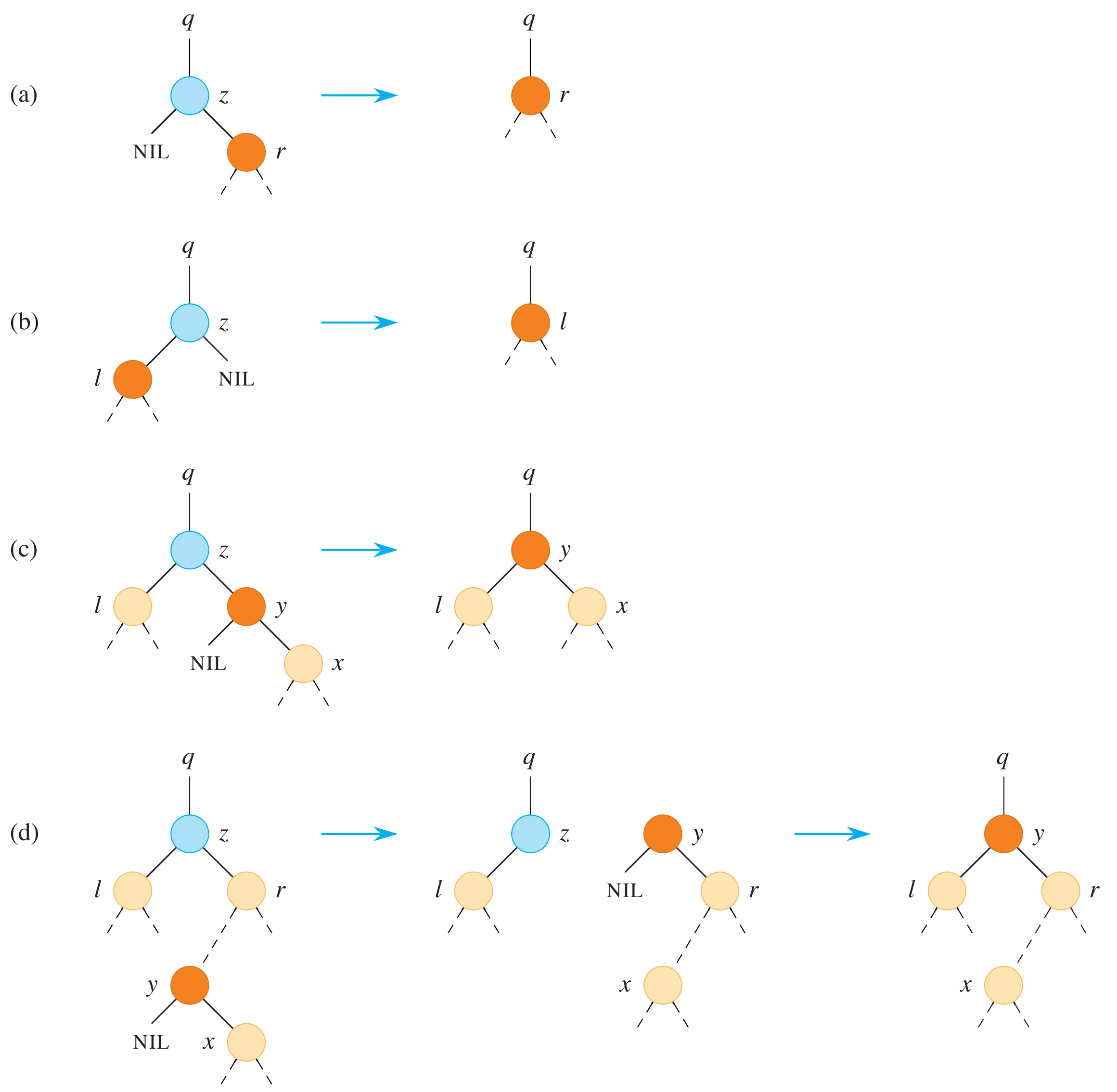


Figure 12.4 Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange. **(a)** Node z has no left child. Replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. Replace z by l. **(c)** Node z has two children. Its left child is node l, its right child is its successor y (which has no left child), and y's right child is node x. Replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. First replace y by its own right child x, and set y to be r's parent. Then set y to be q's child and the parent of l.

the situation in which z has no children. When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.

- Otherwise, if z has just one child, then that child is a left child. As in part (b) of the figure, replace z by its left child.
- Otherwise, z has both a left and a right child. Find z's successor y, which lies in z's right subtree and has no left child (see Exercise 12.2-5). Splice node y out of its current location and replace z by y in the tree. How to do so depends on whether y is z's right child:
    - If y is z's right child, then as in part (c) of the figure, replace z by y, leaving y's right child alone.
    - Otherwise, y lies within z's right subtree but is not z's right child. In this case, as in part (d) of the figure, first replace y by its own right child, and then replace z by y.

As part of the process of deleting a node, subtrees need to move around within the binary search tree. The subroutine TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child. TRANSPLANT allows v to be NIL instead of a pointer to a node.

```
TRANSPLANT(T, u, v)
1  if $u.p$ == NIL
2      $T.root = v$
3  elseif $u == u.p.left$
4      $u.p.left = v$
5  else $u.p.right = v$
6  if $v \neq$ NIL
7      $v.p = u.p$
```

Here is how TRANSPLANT works. Lines 1–2 handle the case in which u is the root of T. Otherwise, u is either a left child or a right child of its parent. Lines 3–4 take care of updating $u.p.left$ if u is a left child, and line 5 updates $u.p.right$ if u is a right child. Because v may be NIL, lines 6–7 update $v.p$ only if v is non-NIL. The procedure TRANSPLANT does not attempt to update $v.left$ and $v.right$. Doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

The procedure TREE-DELETE on the facing page uses TRANSPLANT to delete node z from binary search tree T. It executes the four cases as follows. Lines 1–2 handle the case in which node z has no left child (Figure 12.4(a)), and lines 3–4 handle the case in which z has a left child but no right child (Figure 12.4(b)). Lines 5–12 deal with the remaining two cases, in which z has two children. Line 5 finds node y, which is the successor of z. Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key; hence the call to TREE-MINIMUM($z.right$). As we noted before, y has no left child. The procedure needs to splice y out of its current location and replace z by y in the tree. If y is z's right child (Figure 12.4(c)), then lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child. Node y retains its right child (x in Figure 12.4(c)), and so no change to $y.right$ needs to occur. If y is not z's right child (Figure 12.4(d)), then two nodes have to move. Lines 7–9 replace y as a child of its parent by y's right child (x in Figure 12.4(c)) and make z's right child (r in the figure) become y's right child instead. Finally, lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)          // replace z by its right child
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)           // replace z by its left child
 5  else y = TREE-MINIMUM(z.right)         // y is z's successor
 6      if y ≠ z.right                     // is y farther down the tree?
 7          TRANSPLANT(T, y, y.right)      // replace y by its right child
 8          y.right = z.right              // z's right child becomes
 9          y.right.p = y                  //     y's right child
10      TRANSPLANT(T, z, y)                // replace z by its successor y
11      y.left = z.left                    // and give z's left child to y,
12      y.left.p = y                       //     which had no left child
```

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3

The dynamic-set operations INSERT and DELETE can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.3-1
Give a recursive version of the TREE-INSERT procedure.

12.3-2
Suppose that you construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.

12.3-3
You can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-4
When TREE-DELETE calls TRANSPLANT, under what circumstances can the parameter v of TRANSPLANT be NIL?

12.3-5
Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

12.3-6
Suppose that instead of each node x keeping the attribute $x.p$, pointing to x's parent, it keeps $x.succ$, pointing to x's successor. Give pseudocode for TREE-SEARCH, TREE-INSERT, and TREE-DELETE on a binary search tree T using this representation. These procedures should operate in $O(h)$ time, where h is the height of the tree T. You may assume that all keys in the binary search tree are distinct. (*Hint:* You might wish to implement a subroutine that returns the parent of a node.)

12.3-7
When node z in TREE-DELETE has two children, you can choose node y to be its predecessor rather than its successor. What other changes to TREE-DELETE are necessary if you do so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be minimally changed to implement such a fair strategy?

Problems

12-1 Binary search trees with equal keys

Equal keys pose a problem for the implementation of binary search trees.

a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

Consider changing TREE-INSERT to test whether $z.key = x.key$ before line 5 and to test whether $z.key = y.key$ before line 11. If equality holds, implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, which compares the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

b. Keep a boolean flag $x.b$ at node x, and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time TREE-INSERT visits x while inserting a node with the same key as x.

c. Keep a list of nodes with equal keys at x, and insert z into the list.

d. Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

12-2 Radix trees

Given two strings $a = a_0a_1 \ldots a_p$ and $b = b_0b_1 \ldots b_q$, where each a_i and each b_j belongs to some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \le j \le \min\{p, q\}$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j - 1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 12.5 (also known as a ***trie***) stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0a_1 \ldots a_p$, go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

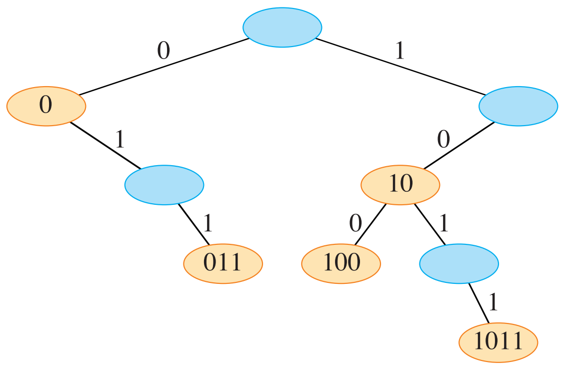


Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. To determine each node's key, traverse the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes. The keys appear here for illustrative purposes only. Keys corresponding to blue nodes are not in the tree. Such nodes are present only to establish a path to other nodes.

12-3 Average node depth in a randomly built binary search tree

A ***randomly built binary search tree*** on n keys is a binary search tree created by starting with an empty tree and inserting the keys in random order, where each of the $n!$ permutations of the keys is equally likely. In this problem, you will prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. The technique reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

Denote the depth of any node x in tree T by $d(x,T)$. Then the ***total path length*** $P(T)$ of a tree T is the sum, over all nodes x in T, of $d(x,T)$.

a. Argue that the average depth of a node in T is

$$\frac{1}{n}\sum_{x \in T} d(x,T) = \frac{1}{n}P(T) .$$

Thus, you need to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

d. Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1}P(k) + \Theta(n) .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$.

Each recursive invocation of randomized quicksort chooses a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.

a. Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Referring to Problem 4-5 on page 121 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right) .$$

The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

c. Show that

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial theorem, equation (C.4) on page 1181, to noninteger exponents n, where for any real number n and for any integer k, you can interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \geq 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}(1 + O(1/n)).$$

Chapter notes

Knuth [261] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950s. Radix trees are often called "tries," which comes from the middle letters in the word *retrieval*. Knuth [261] also discusses them.

Many texts, including the first two editions of this book, describe a somewhat simpler method of deleting a node from a binary search tree when both of its children are present. Instead of replacing node z by its successor y, delete node y but copy its key and satellite data into node z. The downside of this approach is that the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with "stale" pointers to nodes that have been deleted. Although the deletion method presented in this edition of this book is a bit more complicated, it guarantees that a call to delete node z deletes node z and only node z.

Section 14.5 will show how to construct an optimal binary search tree when you know the search frequencies before constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, a set of searches in the constructed binary search tree examines the minimum number of nodes.

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***. Indeed, as we're about to see, the height of a red-black tree with n keys is at most $2\lg(n+1)$, which is $O(\lg n)$.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. Think of these NILs as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree.

A red-black tree is a binary search tree that satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.

4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 262). For a red-black tree T, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its *color* attribute is BLACK, and its other attributes—p, *left*, *right*, and *key*—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

Why use the sentinel? The sentinel makes it possible to treat a NIL child of a node x as an ordinary node whose parent is x. An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly wastes space, however. Instead, just the one sentinel $T.nil$ represents all the NILs—all leaves and the root's parent. The values of the attributes p, *left*, *right*, and *key* of the sentinel are immaterial. The red-black tree procedures can place whatever values in the sentinel that yield simpler code.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. The remainder of this chapter omits the leaves in drawings of red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the ***black-height*** of the node, denoted $\text{bh}(x)$. By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. The black-height of a red-black tree is the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Proof We start by showing that the subtree rooted at any node x contains at least $2^{\text{bh}(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node. Then node x has two children, either or both of which may be a leaf. If a child is black, then it contributes 1 to x's black-height but not to its own. If a child is red, then it contributes to neither x's black-height nor its own. Therefore, each child has a black-height of either $\text{bh}(x) - 1$ (if it's black) or $\text{bh}(x)$ (if it's red). Since the height of a child of x is less than the height of x itself, we can apply the inductive

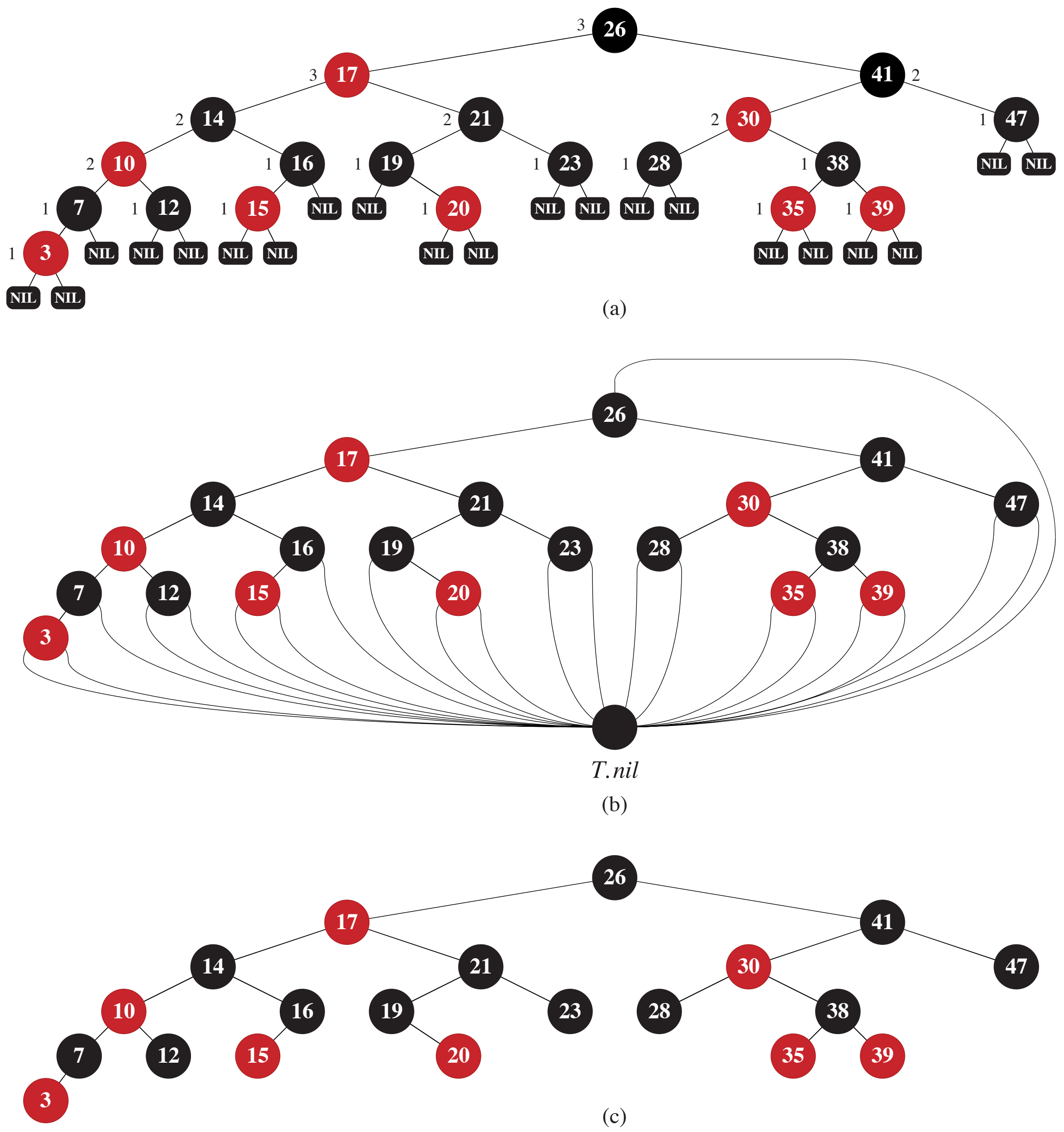


Figure 13.1 A red-black tree. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height, where NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. The remainder of this chapter uses this drawing style.

hypothesis to conclude that each child has at least $2^{\text{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$, and thus,

$$n \geq 2^{h/2} - 1 .$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n+1) \geq h/2$, or $h \leq 2\lg(n+1)$. ■

As an immediate consequence of this lemma, each of the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR runs in $O(\lg n)$ time on a red-black tree, since each can run in $O(h)$ time on a binary search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 have to be replaced by $T.nil$.) Although the procedures TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, you cannot just use them to implement the dynamic-set operations INSERT and DELETE. They do not necessarily maintain the red-black properties, so you might not end up with a legal red-black tree. The remainder of this chapter shows how to insert into and delete from a red-black tree in $O(\lg n)$ time.

Exercises

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

13.1-3

Define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5, but whose root may be either red or black. Consider a relaxed red-black tree T whose root is red. If the root of T is changed to black but no other changes occur, is the resulting tree a red-black tree?

13.1-4
Suppose that every black node in a red-black tree "absorbs" all of its red children, so that the children of any red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5
Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

13.1-6
What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

13.1-7
Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

13.1-8
Argue that in a red-black tree, a red node cannot have exactly one non-NIL child.

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, colors and pointers within nodes need to change.

The pointer structure changes through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. Let's look at a left rotation on a node x, which transforms the structure on the right side of the figure to the structure on the left. Node x has a right child y, which must not be $T.nil$. The left rotation changes the subtree originally rooted at x by "twisting" the link between x and y to the left. The new root of the subtree is node y, with x as y's left child and y's original left child (the subtree represented by β in the figure) as x's right child.

The pseudocode for LEFT-ROTATE appearing on the following page assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$. Figure 13.3 shows an

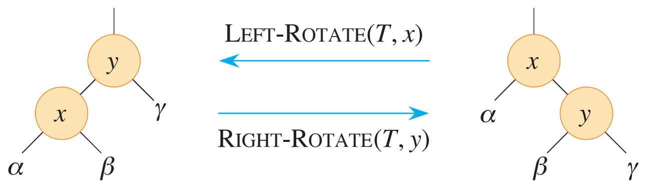


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation, and all other attributes in a node remain the same.

```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left        // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil       // if y's left subtree is not empty ...
 4      y.left.p = x        // ... then x becomes the parent of the subtree's root
 5  y.p = x.p               // x's parent becomes y's parent
 6  if x.p == T.nil         // if x was the root ...
 7      T.root = y          // ... then y becomes the root
 8  elseif x == x.p.left    // otherwise, if x was a left child ...
 9      x.p.left = y        // ... then y becomes a left child
10  else x.p.right = y      // otherwise, x was a right child, and now y is
11  y.left = x              // make x become y's left child
12  x.p = y
```

Exercises

13.2-1
Write pseudocode for RIGHT-ROTATE.

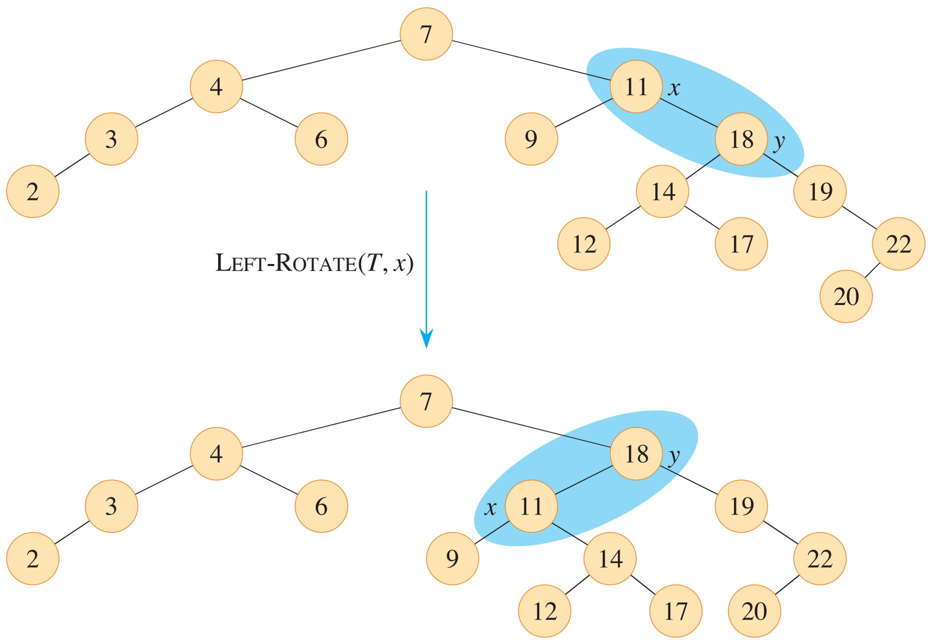


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-2
Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

13.2-3
Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the right tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

13.2-4
Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. (*Hint:* First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

★ ***13.2-5***
We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

13.3 Insertion

In order to insert a node into a red-black tree with n internal nodes in $O(\lg n)$ time and maintain the red-black properties, we'll need to slightly modify the TREE-INSERT procedure on page 321. The procedure RB-INSERT starts by inserting node z into the tree T as if it were an ordinary binary search tree, and then it colors z red. (Exercise 13.3-1 asks you to explain why to make node z red rather than black.) To guarantee that the red-black properties are preserved, an auxiliary procedure RB-INSERT-FIXUP on the facing page recolors nodes and performs rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

```
RB-INSERT(T, z)
 1  $x = T.root$                    // node being compared with $z$
 2  $y = T.nil$                     // $y$ will be parent of $z$
 3  while $x \neq T.nil$            // descend until reaching the sentinel
 4      $y = x$
 5      if $z.key < x.key$
 6          $x = x.left$
 7      else $x = x.right$
 8  $z.p = y$                       // found the location—insert $z$ with parent $y$
 9  if $y == T.nil$
10      $T.root = z$                // tree $T$ was empty
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
14  $z.left = T.nil$                // both of $z$'s children are the sentinel
15  $z.right = T.nil$
16  $z.color$ = RED                 // the new node starts out red
17  RB-INSERT-FIXUP$(T, z)$         // correct any violations of red-black properties
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by $T.nil$. Second, lines 14–15 of RB-INSERT set $z.left$ and $z.right$ to $T.nil$, in order to maintain the proper tree structure. (TREE-INSERT assumed that z's children were already NIL.) Third, line 16 colors z red. Fourth, because coloring z red may cause a violation of one of the red-black properties, line 17 of RB-INSERT calls RB-INSERT-FIXUP(T, z) in order to restore the red-black properties.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left            // is z's parent a left child?
 3          y = z.p.p.right             // y is z's uncle
 4          if y.color == RED           // are z's parent and uncle both red?
 5              z.p.color = BLACK             }
 6              y.color = BLACK               }
 7              z.p.p.color = RED             }  case 1
 8              z = z.p.p                     }
 9          else
10              if z == z.p.right
11                  z = z.p                   }  case 2
12                  LEFT-ROTATE(T, z)         }
13              z.p.color = BLACK             }
14              z.p.p.color = RED             }  case 3
15              RIGHT-ROTATE(T, z.p.p)        }
16      else // same as lines 3–15, but with "right" and "left" exchanged
17          y = z.p.p.left
18          if y.color == RED
19              z.p.color = BLACK
20              y.color = BLACK
21              z.p.p.color = RED
22              z = z.p.p
23          else
24              if z == z.p.left
25                  z = z.p
26                  RIGHT-ROTATE(T, z)
27              z.p.color = BLACK
28              z.p.p.color = RED
29              LEFT-ROTATE(T, z.p.p)
30  T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, let's examine the code in three major steps. First, we'll determine which violations of the red-black properties might arise in RB-INSERT upon inserting node z and coloring it red. Second, we'll consider the overall goal of the **while** loop in lines 1–29. Finally, we'll explore each of the three cases within the **while** loop's body (case 2 falls through into case 3, so these two cases are not mutually exclusive) and see how they accomplish the goal.

In describing the structure of a red-black tree, we'll often need to refer to the sibling of a node's parent. We use the term ***uncle*** for such a node.[1] Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree, with cases depending in part on the colors of a node, its parent, and its uncle.

What violations of the red-black properties might occur upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold (every node is either red or black), as does property 3 (every leaf is black), since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations may arise because z is colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

The **while** loop of lines 1–29 has two symmetric possibilities: lines 3–15 deal with the situation in which node z's parent $z.p$ is a left child of z's grandparent $z.p.p$, and lines 17–29 apply when z's parent is a right child. Our proof will focus only on lines 3–15, relying on the symmetry in lines 17–29.

We'll show that the **while** loop maintains the following three-part invariant at the start of each iteration of the loop:

a. Node z is red.

b. If $z.p$ is the root, then $z.p$ is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4, but not both. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we'll use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. Part (b) will help show that z's grandparent $z.p.p$ exists when it's referenced in lines 2, 3, 7, 8, 14, and 15 (recall that we're focusing only on lines 3–15).

[1] Although we try to avoid gendered language in this book, the English language lacks a gender-neutral word for a parent's sibling.

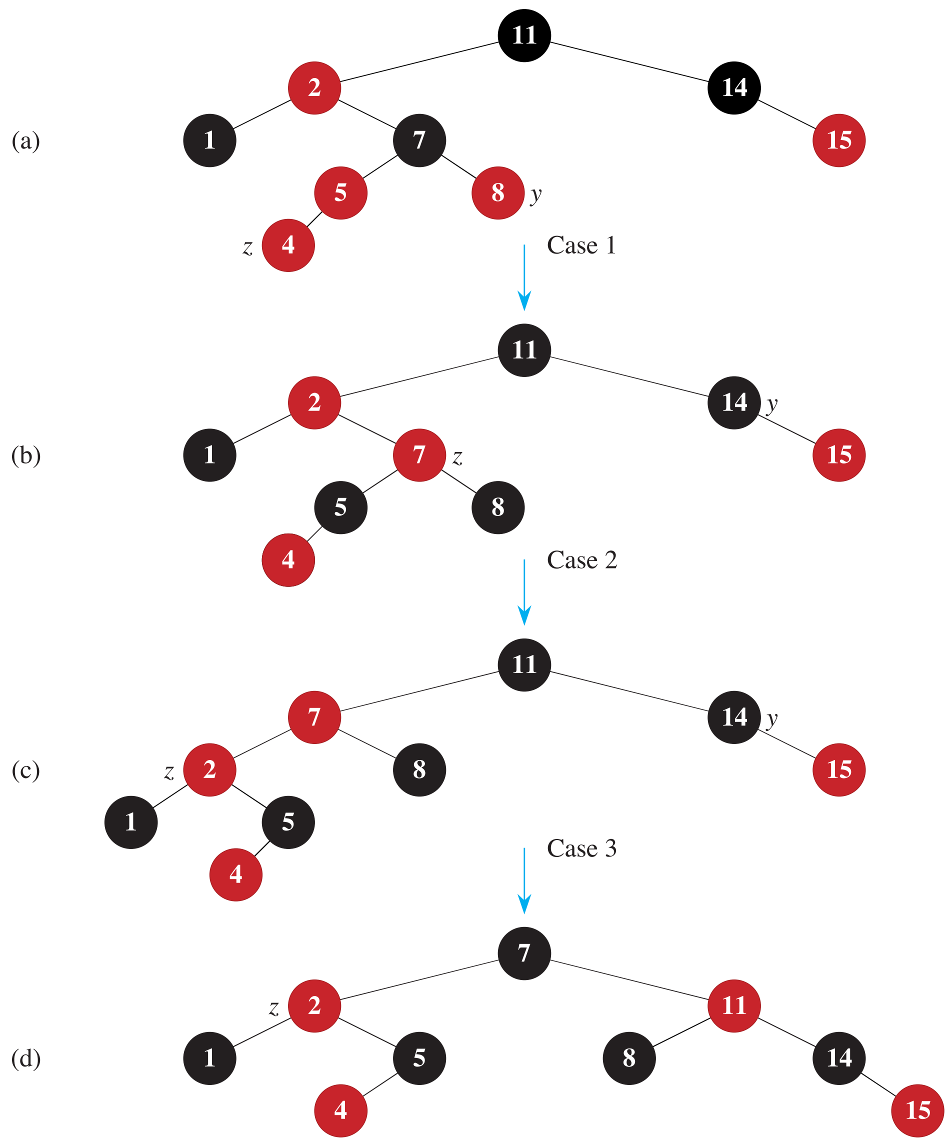


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. Node z's grandparent $z.p.p$ must be black, and its blackness transfers down one level to z's parent and uncle. Once the pointer z moves up two levels in the tree, the tree shown in **(b)** results. Once again, z and its parent are both red, but this time z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. Performing a left rotation results in the tree in **(c)**. Now z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

Recall that to use a loop invariant, we need to show that the invariant is true upon entering the first iteration of the loop, that each iteration maintains it, that the loop terminates, and that the loop invariant gives us a useful property at loop termination. We'll see that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or some rotations occur and then the loop terminates.

Initialization: Before RB-INSERT is called, the red-black tree has no violations. RB-INSERT adds a red node z and calls RB-INSERT-FIXUP. We'll show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

a. When RB-INSERT-FIXUP is called, z is the red node that was added.

b. If $z.p$ is the root, then $z.p$ started out black and did not change before the call of RB-INSERT-FIXUP.

c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-FIXUP is called.

   If the tree violates property 2 (the root must be black), then the red root must be the newly added node z, which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4 (both children of a red node are black). Thus this violation of property 2 is the only violation of red-black properties in the entire tree.

   If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Maintenance: There are six cases within the **while** loop, but we'll examine only the three cases in lines 3–15, when node z's parent $z.p$ is a left child of z's grandparent $z.p.p$. The proof for lines 17–29 is symmetric. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since RB-INSERT-FIXUP enters a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

Case 1 differs from cases 2 and 3 by the color of z's uncle y. Line 3 makes y point to z's uncle $z.p.p.right$, and line 4 tests y's color. If y is red, then case 1 executes. Otherwise, control passes to cases 2 and 3. In all three cases, z's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

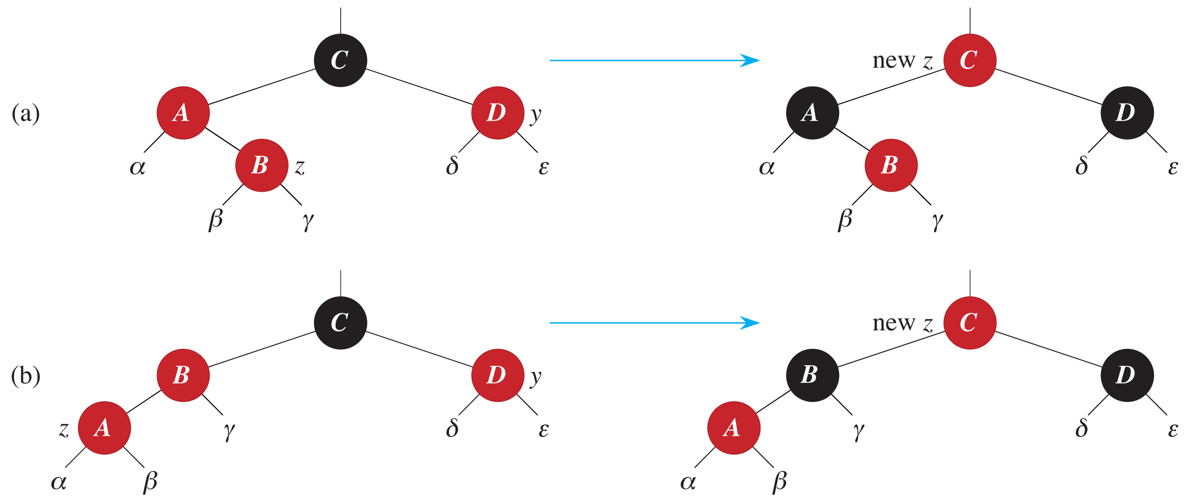


Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Both z and its parent $z.p$ are red, violating property 4. In case 1, z's uncle y is red. The same action occurs regardless of whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root—possibly the sentinel—and each has the same black-height. The code for case 1 moves the blackness of z's grandparent down to z's parent and uncle, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. If the action of case 1 causes a new violation of property 4 to occur, it must be only between the new z, which is red, and its parent, if it is red as well.

Case 1: z's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because z's grandparent $z.p.p$ is black, its blackness can transfer down one level to both $z.p$ and y, thereby fixing the problem of z and $z.p$ both being red. Having had its blackness transferred down one level, z's grandparent becomes red, thereby maintaining property 5. The **while** loop repeats with $z.p.p$ as the new node z, so that the pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

a. Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.

b. The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

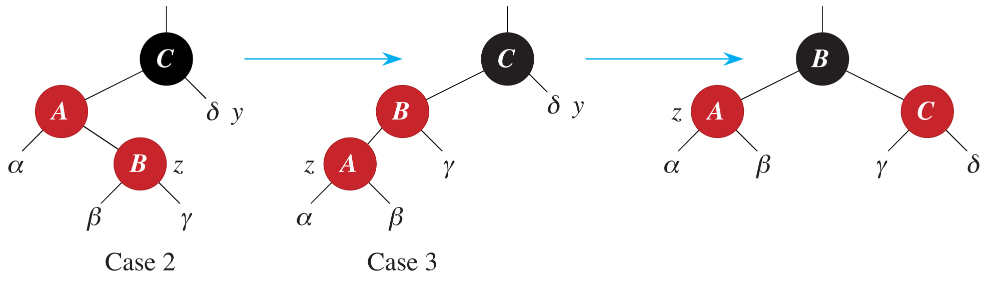


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise case 1 would apply), and each has the same black-height. Case 2 transforms into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

   If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z'.

   If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4, between z' and $z'.p$.

Case 2: z's uncle y is black and z is a right child
Case 3: z's uncle y is black and z is a left child

In cases 2 and 3, the color of z's uncle y is black. We distinguish the two cases, which assume that z's parent $z.p$ is red and a left child, according to whether z is a right or left child of $z.p$. Lines 11–12 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. A left rotation immediately transforms the situation into case 3 (lines 13–15), in which node z is a left child. Because both z and $z.p$ are red, the rotation affects neither the black-heights of nodes nor property 5. Whether case 3 executes directly or through case 2, z's uncle y is black, since otherwise case 1 would have run. Additionally, the node $z.p.p$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 11 and then down one level in line 12, the identity of $z.p.p$ remains unchanged. Case 3 performs some color changes and a right rotation, which preserve property 5. At this point, there are no longer two red nodes in a row. The **while** loop terminates upon the next test in line 1, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.

b. Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

   Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

   Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Termination: To see that the loop terminates, observe that if only case 1 occurs, then the node pointer z moves toward the root in each iteration, so that eventually $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) If either case 2 or case 3 occurs, then we've seen that the loop terminates. Since the loop terminates because $z.p$ is black, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 30 restores this property by coloring the root black, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Thus, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, lines 1–16 of RB-INSERT take $O(\lg n)$ time. In RB-INSERT-FIXUP, the **while** loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Moreover, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Exercises

13.3-1
Line 16 of RB-INSERT sets the color of the newly inserted node z to red. If instead z's color were set to black, then property 4 of a red-black tree would not be violated. Why not set z's color to black?

13.3-2
Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

13.3-3
Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4
Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

13.3-5
Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

13.3-6
Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes $O(\lg n)$ time. Deleting a node from a red-black tree is more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure on page 325. First, we need to customize the TRANSPLANT subroutine on page 324 that TREE-DELETE calls so that it applies to a red-black tree. Like TRANSPLANT, the new procedure RB-TRANSPLANT replaces the subtree rooted at node u by the subtree rooted at node v. The RB-TRANSPLANT procedure differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: the procedure can assign to $v.p$ even if v points to the sentinel. We'll take advantage of the ability to assign to $v.p$ when $v = T.nil$.

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

The procedure RB-DELETE on the next page is like the TREE-DELETE procedure, but with additional lines of pseudocode. The additional lines deal with nodes x and y that may be involved in violations of the red-black properties. When the node z being deleted has at most one child, then y will be z. When z has two children, then, as in TREE-DELETE, y will be z's successor, which has no left child and moves into z's position in the tree. Additionally, y takes on z's color. In either case, node y has at most one child: node x, which takes y's place in the tree. (Node x will be the sentinel $T.nil$ if y has no children.) Since node y will be either removed from the tree or moved within the tree, the procedure needs to keep track of y's original color. If the red-black properties might be violated after deleting node z, RB-DELETE calls the auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by $T.nil$ and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

In detail, here are the other differences between the two procedures:

- Lines 1 and 9 set node y as described above: line 1 when node z has at most one child and line 9 when z has two children.
- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When node z has two children, then nodes y and z are

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)        // replace z by its right child
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)         // replace z by its left child
 9  else y = TREE-MINIMUM(z.right)          // y is z's successor
10      y-original-color = y.color
11      x = y.right
12      if y ≠ z.right                      // is y farther down the tree?
13          RB-TRANSPLANT(T, y, y.right)    // replace y by its right child
14          y.right = z.right               // z's right child becomes
15          y.right.p = y                   //     y's right child
16      else x.p = y                        // in case x is T.nil
17      RB-TRANSPLANT(T, z, y)              // replace z by its successor y
18      y.left = z.left                     // and give z's left child to y,
19      y.left.p = y                        //     which had no left child
20      y.color = z.color
21  if y-original-color == BLACK      // if any red-black violations occurred,
22      RB-DELETE-FIXUP(T, x)         //     correct them
```

distinct. In this case, line 17 moves y into z's original position in the tree (that is, z's location in the tree at the time RB-DELETE was called), and line 20 gives y the same color as z. When node y was originally black, removing or moving it could cause violations of the red-black properties, which are corrected by the call of RB-DELETE-FIXUP in line 22.

- As discussed, the procedure keeps track of the node x that moves into node y's original position at the time of call. The assignments in lines 4, 7, and 11 set x to point to either y's only child or, if y has no children, the sentinel $T.nil$.
- Since node x moves into node y's original position, the attribute $x.p$ must be set correctly. If node z has two children and y is z's right child, then y just moves into z's position, with x remaining a child of y. Line 12 checks for this case. Although you might think that setting $x.p$ to y in line 16 is unnecessary since x is a child of y, the call of RB-DELETE-FIXUP relies on $x.p$ being y even if x is $T.nil$. Thus, when z has two children and y is z's right child, executing

line 16 is necessary if y's right child is $T.nil$, and otherwise it does not change anything.

Otherwise, node z is either the same as node y or it is a proper ancestor of y's original parent. In these cases, the calls of RB-TRANSPLANT in lines 5, 8, and 13 set $x.p$ correctly in line 6 of RB-TRANSPLANT. (In these calls of RB-TRANSPLANT, the third parameter passed is the same as x.)

- Finally, if node y was black, one or more violations of the red-black properties might arise. The call of RB-DELETE-FIXUP in line 22 restores the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

  1. No black-heights in the tree have changed. (See Exercise 13.4-1.)
  2. No red nodes have been made adjacent. If z has at most one child, then y and z are the same node. That node is removed, with a child taking its place. If the removed node was red, then neither its parent nor its children can also be red, so moving a child to take its place cannot cause two red nodes to become adjacent. If, on the other hand, z has two children, then y takes z's place in the tree, along with z's color, so there cannot be two adjacent red nodes at y's new position in the tree. In addition, if y was not z's right child, then y's original right child x replaces y in the tree. Since y is red, x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
  3. Because y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y was the root and a red child of y became the new root, property 2 is violated. Second, if both x and its new parent are red, then a violation of property 4 occurs. Third, moving y within the tree causes any simple path that previously contained y to have one less black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that when the black node y is removed or moved, its blackness transfers to the node x that moves into y's original position, giving x an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. But now another problem emerges: node x is neither red nor black, thereby violating property 1. Instead, node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

The procedure RB-DELETE-FIXUP on the next page restores properties 1, 2, and 4. Exercises 13.4-2 and 13.4-3 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we focus on property 1. The goal of the **while** loop in lines 1–43 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case line 44 colors x (singly) black;
2. x points to the root, in which case the extra black simply vanishes; or
3. having performed suitable rotations and recolorings, the loop exits.

Like RB-INSERT-FIXUP, the RB-DELETE-FIXUP procedure handles two symmetric situations: lines 3–22 for when node x is a left child, and lines 24–43 for when x is a right child. Our proof focuses on the four cases shown in lines 3–22.

Within the **while** loop, x always points to a nonroot doubly black node. Line 2 determines whether x is a left child or a right child of its parent $x.p$ so that either lines 3–22 or 24–43 will execute in a given iteration. The sibling of x is always denoted by a pointer w. Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x.

Recall that the RB-DELETE procedure always assigns to $x.p$ before calling RB-DELETE-FIXUP (either within the call of RB-TRANSPLANT in line 13 or the assignment in line 16), even when node x is the sentinel $T.nil$. That is because RB-DELETE-FIXUP references x's parent $x.p$ in several places, and this attribute must point to the node that became x's parent in RB-DELETE—even if x is $T.nil$.

Figure 13.7 demonstrates the four cases in the code when node x is a left child. (As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.) Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x's extra black) from (and including) the root of the subtree shown to the roots of each of the subtrees $\alpha, \beta, \ldots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to the root of either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to the root of any of γ, δ, ε, and ζ is 2, both before and after the transformation.[2] In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK.

[2] If property 5 holds, we can assume that paths from the roots of γ, δ, ε, and ζ down to leaves contain one more black than do paths from the roots of α and β down to leaves.

```
RB-DELETE-FIXUP$(T, x)$
 1  while $x \neq T.root$ and $x.color$ == BLACK
 2      if $x == x.p.left$                    // is $x$ a left child?
 3          $w = x.p.right$                   // $w$ is $x$'s sibling
 4          if $w.color$ == RED
 5              $w.color$ = BLACK                          ⎫
 6              $x.p.color$ = RED                          ⎬ case 1
 7              LEFT-ROTATE$(T, x.p)$                      ⎪
 8              $w = x.p.right$                            ⎭
 9          if $w.left.color$ == BLACK and $w.right.color$ == BLACK
10              $w.color$ = RED                            ⎫ case 2
11              $x = x.p$                                  ⎭
12          else
13              if $w.right.color$ == BLACK
14                  $w.left.color$ = BLACK                 ⎫
15                  $w.color$ = RED                        ⎬ case 3
16                  RIGHT-ROTATE$(T, w)$                   ⎪
17                  $w = x.p.right$                        ⎭
18              $w.color = x.p.color$                      ⎫
19              $x.p.color$ = BLACK                        ⎪
20              $w.right.color$ = BLACK                    ⎬ case 4
21              LEFT-ROTATE$(T, x.p)$                      ⎪
22              $x = T.root$                               ⎭
23      else // same as lines 3–22, but with "right" and "left" exchanged
24          $w = x.p.left$
25          if $w.color$ == RED
26              $w.color$ = BLACK
27              $x.p.color$ = RED
28              RIGHT-ROTATE$(T, x.p)$
29              $w = x.p.left$
30          if $w.right.color$ == BLACK and $w.left.color$ == BLACK
31              $w.color$ = RED
32              $x = x.p$
33          else
34              if $w.left.color$ == BLACK
35                  $w.right.color$ = BLACK
36                  $w.color$ = RED
37                  LEFT-ROTATE$(T, w)$
38                  $w = x.p.left$
39              $w.color = x.p.color$
40              $x.p.color$ = BLACK
41              $w.left.color$ = BLACK
42              RIGHT-ROTATE$(T, x.p)$
43              $x = T.root$
44  $x.color$ = BLACK
```

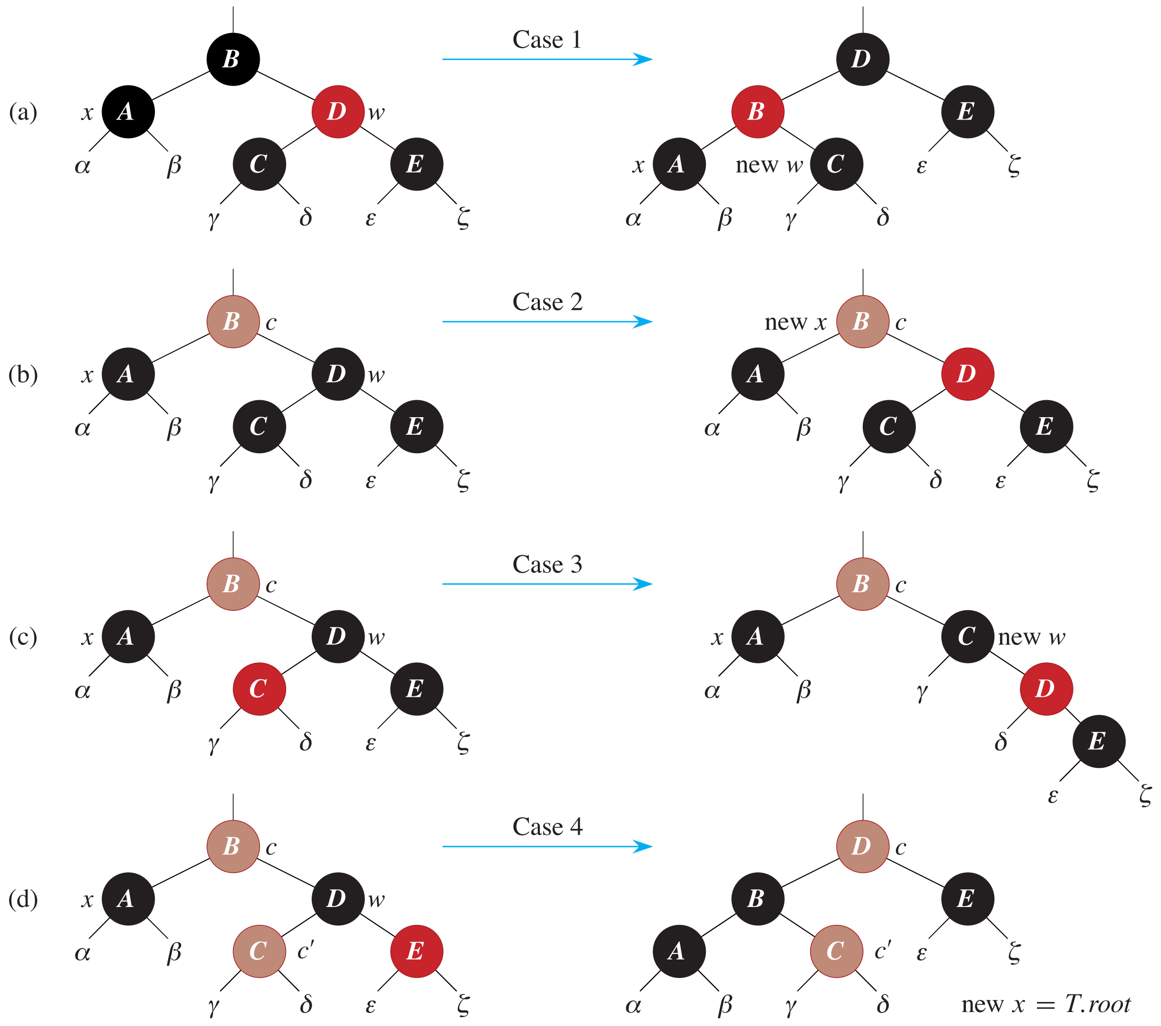


Figure 13.7 The cases in lines 3–22 of the procedure RB-DELETE-FIXUP. Brown nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed into case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If case 2 is entered through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is RED. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

If we define count(RED) $= 0$ and count(BLACK) $= 1$, then the number of black nodes from the root to α is $2 + \text{count}(c)$, both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c, but this node is really either red-and-black (if $c =$ RED) or doubly black (if $c =$ BLACK). You can verify the other cases similarly (see Exercise 13.4-6).

Case 1: x's sibling w is red

Case 1 (lines 5–8 and Figure 13.7(a)) occurs when node w, the sibling of node x, is red. Because w is red, it must have black children. This case switches the colors of w and $x.p$ and then performs a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus case 1 converts into one of cases 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black and are distinguished by the colors of w's children.

Case 2: x's sibling w is black, and both of w's children are black

In case 2 (lines 10–11 and Figure 13.7(b)), both of w's children are black. Since w is also black, this case removes one black from both x and w, leaving x with only one black and leaving w red. To compensate for x and w each losing one black, x's parent $x.p$ can take on an extra black. Line 11 does so by moving x up one level, so that the **while** loop repeats with $x.p$ as the new node x. If case 2 enters through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. Line 44 then colors the new node x (singly) black.

Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (lines 14–17 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. This case switches the colors of w and its left child $w.\mathit{left}$ and then performs a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus case 3 falls through into case 4.

Case 4: x's sibling w is black, and w's right child is red

Case 4 (lines 18–22 and Figure 13.7(d)) occurs when node x's sibling w is black and w's right child is red. Some color changes and a left rotation on $x.p$ allow the extra black on x to vanish, making it singly black, without violating any of the red-black properties. Line 22 sets x to be the root, and the **while** loop terminates when it next tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

Exercises

13.4-1
Show that if node y in RB-DELETE is red, then no black-heights change.

13.4-2
Argue that after RB-DELETE-FIXUP executes, the root of the tree must be black.

13.4-3
Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

13.4-4
In Exercise 13.3-2 on page 346, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

13.4-5
Which lines of the code for RB-DELETE-FIXUP might examine or modify the sentinel $T.nil$?

13.4-6
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to the roots of each of the subtrees $\alpha, \beta, \ldots, \zeta$, and verify that each count remains the same after the transformation. When a node has a *color* attribute c or c', use the notation count(c) or count(c') symbolically in your count.

13.4-7
Professors Skelton and Baron worry that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If $x.p$ is not black, then lines 5–6 are

wrong. Show that $x.p$ must be black at the start of case 1, so that the professors need not be concerned.

13.4-8
A node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree always the same as the initial red-black tree? Justify your answer.

★ ***13.4-9***
Consider the operation RB-ENUMERATE(T, r, a, b), which outputs all the keys k such that $a \le k \le b$ in a subtree rooted at node r in an n-node red-black tree T. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output. Assume that the keys in T are unique and that the values a and b appear as keys in T. How does your solution change if a and b might not appear in T?

Problems

13-1 Persistent dynamic sets
During the course of an algorithm, you sometimes find that you need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume a lot of space. Sometimes, you can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which you implement using binary search trees as shown in Figure 13.8(a). Maintain a separate root for every version of the set. In order to insert the key 5 into the set, create a new node with key 5. This node becomes the left child of a new node with key 7, since you cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. Thus, you copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes *key*, *left*, and *right* but no parent. (See also Exercise 13.3-6 on page 346.)

a. For a persistent binary search tree (not a red-black tree, just a binary search tree), identify the nodes that need to change to insert or delete a node.

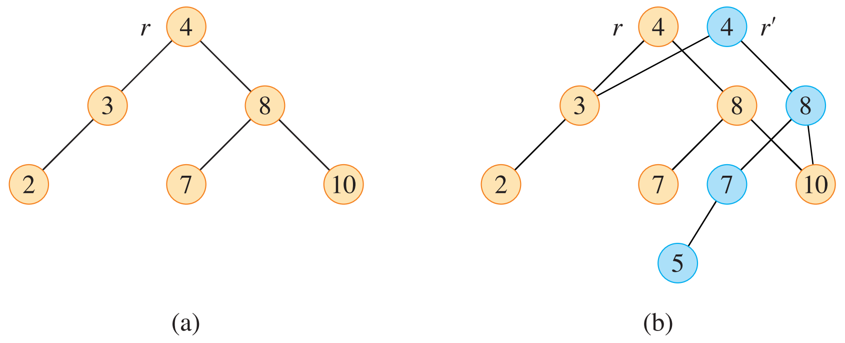


Figure 13.8 **(a)** A binary search tree with keys $2, 3, 4, 7, 8, 10$. **(b)** The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Blue nodes are added when key 5 is inserted.

b. Write a procedure PERSISTENT-TREE-INSERT(T, z) that, given a persistent binary search tree T and a node z to insert, returns a new persistent tree T' that is the result of inserting z into T. Assume that you have a procedure COPY-NODE(x) that makes a copy of node x, including all of its attributes.

c. If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of nodes that are copied.)

d. Suppose that you include the parent attribute in each node. In this case, the PERSISTENT-TREE-INSERT procedure needs to perform additional copying. Prove that PERSISTENT-TREE-INSERT then requires $\Omega(n)$ time and space, where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion. You may assume that all keys are distinct.

13-2 Join operation on red-black trees

The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \le x.key \le x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Suppose that you store the black-height of a red-black tree T as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the *bh* attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show how to determine the black-height of each node visited while descending through T, using $O(1)$ time per node visited.

Let T_1 and T_2 be red-black trees and x be a key value such that for any nodes x_1 in T_1 and x_2 in T_2, we have $x_1.key \leq x.key \leq x_2.key$. You will show how to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $T_1.bh \geq T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should you make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

13-3 AVL trees

An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, maintain an extra attribute h in each node such that $x.h$ is the height of node x. As for any other binary search tree T, assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint:* Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, so that $|x.right.h - x.left.h| \le 2$, and alters the subtree rooted at x to be height balanced. The procedure should return a pointer to the node that is the root of the subtree after alterations occur. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(T, z) that takes an AVL tree T and a newly created node z (whose key has already been filled in), and adds z into T, maintaining the property that T is an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$. Assume as well that $z.h = 0$.

d. Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(\lg n)$ rotations.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called "AVL trees" in 1962, described in Problem 13-3. Another class of search trees, called "2-3 trees," was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree, where each node has either two or three children. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [39], called "B-trees."

Red-black trees were invented by Bayer [38] under the name "symmetric binary B-trees." Guibas and Sedgewick [202] studied their properties at length and introduced the red/black color convention. Andersson [16] gives a simpler-to-code variant of red-black trees. Weiss [451] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children can never be red.

Sedgewick and Wayne [402] present red-black trees as a modified version of 2-3 trees in which a node with three children is split into two nodes with two children each. One of these nodes becomes the left child of the other, and only left children can be red. They call this structure a "left-leaning red-black binary search tree." Although the code for left-leaning red-black binary search trees is more concise than the red-black tree pseudocode in this chapter, operations on left-leaning red-black binary search trees do not limit the number of rotations per operation to a constant. This distinction will matter in Chapter 17.

Treaps, a hybrid of binary search trees and heaps, were proposed by Seidel and Aragon [404]. They are the default implementation of a dictionary in LEDA [324], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [344], k-neighbor trees [318], and scapegoat trees [174]. Perhaps the most intriguing are the "splay trees" introduced by Sleator and Tarjan [418], which are "self-adjusting." (See Tarjan [429] for a good description of splay trees.) Splay trees maintain balance without any explicit balance condition such as color. Instead, "splay operations" (which involve rotations) are performed within the tree every time an access is made. The amortized cost (see Chapter 16) of each operation on an n-node tree is $O(\lg n)$. Splay trees have been conjectured to perform within a constant factor of the best offline rotation-based tree. The best known competitive ratio (see Chapter 27) for a rotation-based tree is the Tango Tree of Demaine et al. [109].

Skip lists [369] provide an alternative to balanced binary trees. A skip list is a linked list that is augmented with a number of additional pointers. Each dictionary operation runs in $O(\lg n)$ expected time on a skip list of n items.

Part IV Advanced Design and Analysis Techniques

Introduction

This part covers three important techniques used in designing and analyzing efficient algorithms: dynamic programming (Chapter 14), greedy algorithms (Chapter 15), and amortized analysis (Chapter 16). Earlier parts have presented other widely applicable techniques, such as divide-and-conquer, randomization, and how to solve recurrences. The techniques in this part are somewhat more sophisticated, but you will be able to use them solve many computational problems. The themes introduced in this part will recur later in this book.

Dynamic programming typically applies to optimization problems in which you make a set of choices in order to arrive at an optimal solution, each choice generates subproblems of the same form as the original problem, and the same subproblems arise repeatedly. The key strategy is to store the solution to each such subproblem rather than recompute it. Chapter 14 shows how this simple idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which you make a set of choices in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner, resulting in a faster algorithm than you get with dynamic programming. Chapter 15 will help you determine when the greedy approach works.

The technique of amortized analysis applies to certain algorithms that perform a sequence of similar operations. Instead of bounding the cost of the sequence of operations by bounding the actual cost of each operation separately, an amortized analysis provides a worst-case bound on the actual cost of the entire sequence. One advantage of this approach is that although some operations might be expensive, many others might be cheap. You can use amortized analysis when designing algorithms, since the design of an algorithm and the analysis of its running time are often closely intertwined. Chapter 16 introduces three ways to perform an amortized analysis of an algorithm.

14 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Dynamic programming typically applies to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

To develop a dynamic-programming algorithm, follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If you need only the value of an optimal solution, and not the solution itself, then you can omit step 4. When you do perform step 4, it often pays to maintain additional information during step 3 so that you can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 14.1 examines the problem of cutting a rod into rods of smaller length in a way that maximizes their total value. Section 14.2 shows how to multiply a chain of matrices while performing the fewest total scalar multiplications. Given these examples of dynamic programming, Section 14.3 discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. Section 14.4 then shows how to find the longest common subsequence of two sequences via dynamic programming. Finally, Section 14.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

14.1 Rod cutting

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

Serling Enterprises has a table giving, for $i = 1, 2, \ldots$, the price p_i in dollars that they charge for a rod of length i inches. The length of each rod in inches is always an integer. Figure 14.1 gives a sample price table.

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. If the price p_n for a rod of length n is large enough, an optimal solution might require no cutting at all.

Consider the case when $n = 4$. Figure 14.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. Cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

Serling Enterprises can cut up a rod of length n in 2^{n-1} different ways, since they have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1, 2, \ldots, n-1$.[1] We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3. If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

[1] If pieces are required to be cut in order of monotonically increasing size, there are fewer ways to consider. For $n = 4$, only 5 such ways are possible: parts (a), (b), (c), (e), and (h) in Figure 14.2. The number of ways is called the ***partition function***, which is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. This quantity is less than 2^{n-1}, but still much greater than any polynomial in n. We won't pursue this line of inquiry further, however.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 14.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

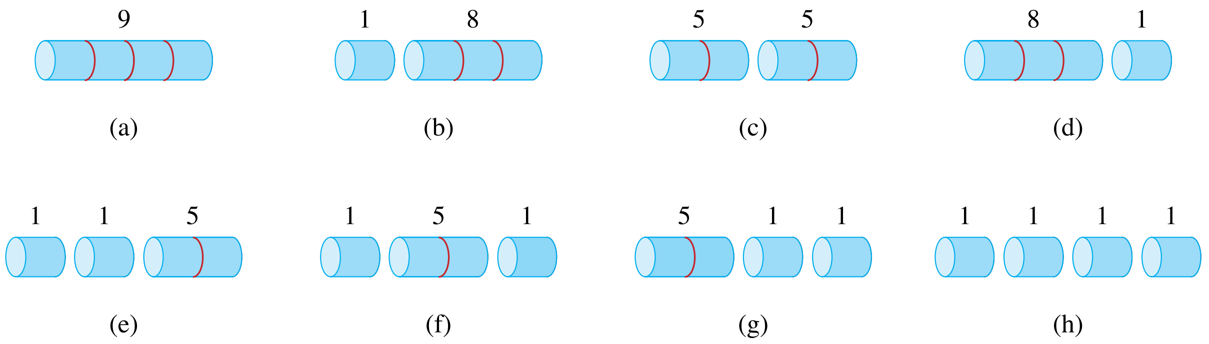


Figure 14.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 14.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} .$$

For the sample problem in Figure 14.1, you can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

$$\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)} ,\\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)} ,\\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)} ,\\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 ,\\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 ,\\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)} ,\\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 ,\\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 ,\\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 ,\\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)} .
\end{aligned}$$

More generally, we can express the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} . \tag{14.1}$$

The first argument, p_n, corresponds to making no cuts at all and selling the rod of length n as is. The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1, 2, \ldots, n-1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces. Since you don't know ahead of time which value of i optimizes revenue, you have to consider all possible values for i and pick the one that maximizes revenue. You also have the option of picking no i at all if the greatest revenue comes from selling the rod uncut.

To solve the original problem of size n, you solve smaller problems of the same type. Once you make the first cut, the two resulting pieces form independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two resulting subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting problem exhibits ***optimal substructure***: optimal solutions to a problem incorporate optimal solutions to related subproblems, which you may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, let's view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. Think of every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. Then we can express the solution with no cuts at all by saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation (14.1):

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\} . \tag{14.2}$$

In this formulation, an optimal solution embodies the solution to only *one* related subproblem—the remainder—rather than two.

Recursive top-down implementation

The CUT-ROD procedure on the following page implements the computation implicit in equation (14.2) in a straightforward, top-down, recursive manner. It takes as input an array $p[1:n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. For length $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes

$q = \max\{p_i + \text{CUT-ROD}(p, n-i) : 1 \le i \le n\}$. Line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using equation (14.2).

```
CUT-ROD(p, n)
1 if n == 0
2     return 0
3 q = −∞
4 for i = 1 to n
5     q = max {q, p[i] + CUT-ROD(p, n − i)}
6 return q
```

If you code up CUT-ROD in your favorite programming language and run it on your computer, you'll find that once the input size becomes moderately large, your program takes a long time to run. For $n = 40$, your program may take several minutes and possibly more than an hour. For large values of n, you'll also discover that each time you increase n by 1, your program's running time approximately doubles.

Why is CUT-ROD so inefficient? The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values, which means that it solves the same subproblems repeatedly. Figure 14.3 shows a recursion tree demonstrating what happens for $n = 4$: CUT-ROD(p, n) calls CUT-ROD$(p, n - i)$ for $i = 1, 2, \ldots, n$. Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \ldots, n - 1$. When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

To analyze the running time of CUT-ROD, let $T(n)$ denote the total number of calls made to CUT-ROD(p, n) for a particular value of n. This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. The count includes the initial call at its root. Thus, $T(0) = 1$ and

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \ . \tag{14.3}$$

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call CUT-ROD$(p, n - i)$, where $j = n - i$. As Exercise 14.1-1 asks you to show,

$$T(n) = 2^n \ , \tag{14.4}$$

and so the running time of CUT-ROD is exponential in n.

In retrospect, this exponential running time is not so surprising. CUT-ROD explicitly considers all possible ways of cutting up a rod of length n. How many ways

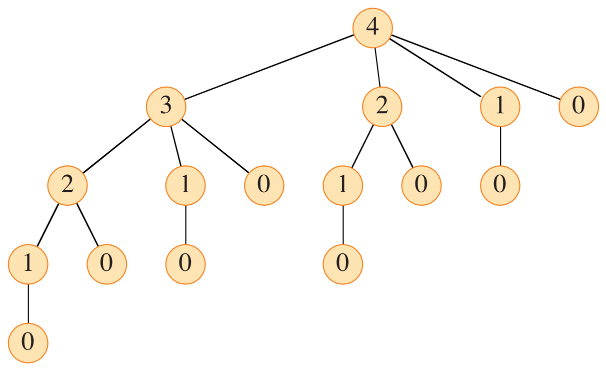


Figure 14.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

are there? A rod of length n has $n - 1$ potential locations to cut. Each possible way to cut up the rod makes a cut at some subset of these $n - 1$ locations, including the empty set, which makes for no cuts. Viewing each cut location as a distinct member of a set of $n - 1$ elements, you can see that there are 2^{n-1} subsets. Each leaf in the recursion tree of Figure 14.3 corresponds to one possible way to cut up the rod. Hence, the recursion tree has 2^{n-1} leaves. The labels on the simple path from the root to a leaf give the sizes of each remaining right-hand piece before making each cut. That is, the labels give the corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

Now, let's see how to use dynamic programming to convert CUT-ROD into an efficient algorithm.

The dynamic-programming method works as follows. Instead of solving the same subproblems repeatedly, as in the naive recursion solution, arrange for each subproblem to be solved *only once*. There's actually an obvious way to do so: the first time you solve a subproblem, *save its solution*. If you need to refer to this subproblem's solution again later, just look it up, rather than recomputing it.

Saving subproblem solutions comes with a cost: the additional memory needed to store solutions. Dynamic programming thus serves as an example of a ***time-memory trade-off***. The savings may be dramatic. For example, we're about to use dynamic programming to go from the exponential-time algorithm for rod cutting down to a $\Theta(n^2)$-time algorithm. A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and you can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. Solutions to the rod-cutting problem illustrate both of them.

The first approach is ***top-down*** with ***memoization***.[2] In this approach, you write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level. If not, the procedure computes the value in the usual manner but also saves it. We say that the recursive procedure has been ***memoized***: it "remembers" what results it has computed previously.

The second approach is the ***bottom-up method***. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. Solve the subproblems in size order, smallest first, storing the solution to each subproblem when it is first solved. In this way, when solving a particular subproblem, there are already saved solutions for all of the smaller subproblems its solution depends upon. You need to solve each subproblem only once, and when you first see it, you have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has lower overhead for procedure calls.

The procedures MEMOIZED-CUT-ROD and MEMOIZED-CUT-ROD-AUX on the facing page demonstrate how to memoize the top-down CUT-ROD procedure. The main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0:n]$ with the value $-\infty$ which, since known revenue values are always nonnegative, is a convenient choice for denoting "unknown." MEMOIZED-CUT-ROD then calls its helper procedure, MEMOIZED-CUT-ROD-AUX, which is just the memoized version of the exponential-time procedure, CUT-ROD. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

The bottom-up version, BOTTOM-UP-CUT-ROD on the next page, is even simpler. Using the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD takes advantage of the natural ordering of the subproblems: a subproblem of

[2] The technical term "memoization" is not a misspelling of "memorization." The word "memoization" comes from "memo," since the technique consists of recording a value to be looked up later.

```
MEMOIZED-CUT-ROD$(p, n)$
let $r[0:n]$ be a new array          // will remember solution values in $r$
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
if $r[n] \geq 0$                 // already have a solution for length $n$?
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$      // $i$ is the position of the first cut
        $q = \max\{q, p[i] + $MEMOIZED-CUT-ROD-AUX$(p, n-i, r)\}$
$r[n] = q$                      // remember the solution value for length $n$
return $q$
```

```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0:n]$ be a new array          // will remember solution values in $r$
$r[0] = 0$
for $j = 1$ to $n$                   // for increasing rod length $j$
    $q = -\infty$
    for $i = 1$ to $j$               // $i$ is the position of the first cut
        $q = \max\{q, p[i] + r[j-i]\}$
    $r[j] = q$                       // remember the solution value for length $j$
return $r[n]$
```

size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Line 1 of BOTTOM-UP-CUT-ROD creates a new array $r[0:n]$ in which to save the results of the subproblems, and line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now directly references array entry $r[j-i]$ instead of making a recursive call to solve the subproblem of size $j-i$. Line 7 saves in $r[j]$ the solution to the subproblem of size j. Finally, line 8 returns $r[n]$, which equals the optimal value r_n.

The bottom-up and top-down versions have the same asymptotic running time. The running time of BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly nested

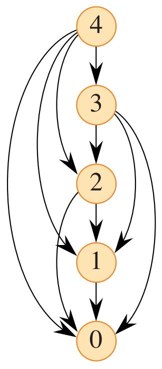


Figure 14.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y. This graph is a reduced version of the recursion tree of Figure 14.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

loop structure. The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series. The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop of lines 6–7 iterates n times. Thus, the total number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations, just like the inner **for** loop of BOTTOM-UP-CUT-ROD. (We actually are using a form of aggregate analysis here. We'll see aggregate analysis in detail in Section 16.1.)

Subproblem graphs

When you think about a dynamic-programming problem, you need to understand the set of subproblems involved and how subproblems depend on one another.

The ***subproblem graph*** for the problem embodies exactly this information. Figure 14.4 shows the subproblem graph for the rod-cutting problem with $n = 4$. It is a directed graph, containing one vertex for each distinct subproblem. The subproblem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y. For example, the subproblem graph contains an edge from x to y if a top-down recursive procedure for solving x directly calls itself to solve y. You can think of the subproblem graph as a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method, with all nodes for the same subproblem coalesced into a single vertex and all edges directed from parent to child.

The bottom-up method for dynamic programming considers the vertices of the subproblem graph in such an order that you solve the subproblems y adjacent to a given subproblem x before you solve subproblem x. (As Section B.4 notes, the adjacency relation in a directed graph is not necessarily symmetric.) Using terminology that we'll see in Section 20.4, in a bottom-up dynamic-programming algorithm, you consider the vertices of the subproblem graph in an order that is a "reverse topological sort," or a "topological sort of the transpose" of the subproblem graph. In other words, no subproblem is considered until all of the subproblems it depends upon have been solved. Similarly, using notions that we'll visit in Section 20.3, you can view the top-down method (with memoization) for dynamic programming as a "depth-first search" of the subproblem graph.

The size of the subproblem graph $G = (V, E)$ can help you determine the running time of the dynamic-programming algorithm. Since you solve each subproblem just once, the running time is the sum of the times needed to solve each subproblem. Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph. In this common case, the running time of dynamic programming is linear in the number of vertices and edges.

Reconstructing a solution

The procedures MEMOIZED-CUT-ROD and BOTTOM-UP-CUT-ROD return the *value* of an optimal solution to the rod-cutting problem, but they do not return the solution *itself*: a list of piece sizes.

Let's see how to extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, you can readily print an optimal solution. The procedure EXTENDED-BOTTOM-UP-CUT-ROD on the next page computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off. It's similar to BOTTOM-UP-CUT-ROD, except that it creates the array s in line 1, and it updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

The procedure PRINT-CUT-ROD-SOLUTION on the following page takes as input an array $p[1:n]$ of prices and a rod size n. It calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1:n]$ of optimal first-piece sizes. Then it prints out the complete list of piece sizes in an optimal decomposition of a rod of length n. For the sample price chart appearing in Figure 14.1, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ returns the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$		1	2	3	2	2	6	1	2	3	10

A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ prints just 10, but a call with $n = 7$ prints the cuts 1 and 6, which correspond to the first optimal decomposition for r_7 given earlier.

```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0:n]$ and $s[1:n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$                    // for increasing rod length $j$
    $q = -\infty$
    for $i = 1$ to $j$                // $i$ is the position of the first cut
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$                // best cut location so far for length $j$
    $r[j] = q$                        // remember the solution value for length $j$
return $r$ and $s$
```

```
PRINT-CUT-ROD-SOLUTION$(p, n)$
$(r, s)$ = EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
while $n > 0$
    print $s[n]$          // cut location for length $n$
    $n = n - s[n]$        // length of the remainder of the rod
```

Exercises

14.1-1
Show that equation (14.4) follows from equation (14.3) and the initial condition $T(0) = 1$.

14.1-2
Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i / i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum

density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

14.1-3
Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

14.1-4
Modify CUT-ROD and MEMOIZED-CUT-ROD-AUX so that their **for** loops go up to only $\lfloor n/2 \rfloor$, rather than up to n. What other changes to the procedures do you need to make? How are their running times affected?

14.1-5
Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution.

14.1-6
The Fibonacci numbers are defined by recurrence (3.31) on page 69. Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges does the graph contain?

14.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, where the matrices aren't necessarily square, the goal is to compute the product

$$A_1 A_2 \cdots A_n . \tag{14.5}$$

using the standard algorithm[3] for multiplying rectangular matrices, which we'll see in a moment, while minimizing the number of scalar multiplications.

You can evaluate the expression (14.5) using the algorithm for multiplying pairs of rectangular matrices as a subroutine once you have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of

[3] None of the three methods from Sections 4.1 and Section 4.2 can be used directly, because they apply only to square matrices.

matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then you can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$$
\begin{aligned}
&(A_1(A_2(A_3 A_4)))\ ,\\
&(A_1((A_2 A_3) A_4))\ ,\\
&((A_1 A_2)(A_3 A_4))\ ,\\
&((A_1(A_2 A_3)) A_4)\ ,\\
&(((A_1 A_2) A_3) A_4)\ .
\end{aligned}
$$

How you parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two rectangular matrices. The standard algorithm is given by the procedure RECTANGULAR-MATRIX-MULTIPLY, which generalizes the square-matrix multiplication procedure MATRIX-MULTIPLY on page 81. The RECTANGULAR-MATRIX-MULTIPLY procedure computes $C = C + A \cdot B$ for three matrices $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$, where A is $p \times q$, B is $q \times r$, and C is $p \times r$.

```
RECTANGULAR-MATRIX-MULTIPLY($A, B, C, p, q, r$)
1  for $i = 1$ to $p$
2      for $j = 1$ to $r$
3          for $k = 1$ to $q$
4              $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
```

The running time of RECTANGULAR-MATRIX-MULTIPLY is dominated by the number of scalar multiplications in line 4, which is pqr. Therefore, we'll consider the cost of multiplying matrices to be the number of scalar multiplications. (The number of scalar multiplications dominates even if we consider initializing $C = 0$ to perform just $C = A \cdot B$.)

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. Multiplying according to the parenthesization $((A_1 A_2) A_3)$ performs $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. Multiplying according to the alternative parenthesization $(A_1(A_2 A_3))$ performs $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

The matrix-chain multiplication problem does not entail actually multiplying matrices. The goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations is not an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, the sequence consists of just one matrix, and therefore there is only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \, , \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \, . \end{cases} \tag{14.6}$$

Problem 12-4 on page 329 asked you to show that the solution to a similar recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n/n^{3/2})$. A simpler exercise (see Exercise 14.2-3) is to show that the solution to the recurrence (14.6) is $\Omega(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

Let's use the dynamic-programming method to determine how to optimally parenthesize a matrix chain, by following the four-step sequence that we stated at the beginning of this chapter:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

We'll go through these steps in order, demonstrating how to apply each step to the problem.

Step 1: The structure of an optimal parenthesization

In the first step of the dynamic-programming method, you find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. To perform this step for the matrix-chain multiplication problem, it's convenient to first introduce some notation. Let $A_{i:j}$, where $i \le j$, denote the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. If the problem is nontrivial, that is, $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, the product must split between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, first compute the matrices $A_{i:k}$ and $A_{k+1:j}$, and then multiply them together to produce the final product $A_{i:j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i:k}$, plus the cost of computing $A_{k+1:j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, you split the product between A_k and A_{k+1}. Then the way you parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then you could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost is lower than the optimum: a contradiction. A similar observation holds for how to parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Now let's use the optimal substructure to show how to construct an optimal solution to the problem from optimal solutions to subproblems. Any solution to a nontrivial instance of the matrix-chain multiplication problem requires splitting the product, and any optimal solution contains within it optimal solutions to subproblem instances. Thus, to build an optimal solution to an instance of the matrix-chain multiplication problem, split the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), find optimal solutions to the two subproblem instances, and then combine these optimal subproblem solutions. To ensure that you've examined the optimal split, you must consider all possible splits.

Step 2: A recursive solution

The next step is to define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, a subproblem is to determine the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Given the input dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$, an index pair i, j specifies a subproblem. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i:j}$. For the full problem, the lowest-cost way to compute $A_{1:n}$ is thus $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial: the chain consists of just one matrix $A_{i:i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Suppose that an optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i, j]$ equals the minimum cost $m[i, k]$ for computing the subproduct $A_{i:k}$, plus the minimum cost $m[k+1, j]$ for computing the subproduct, $A_{k+1:j}$, plus the cost of multiplying these two matrices together. Because each matrix A_i is $p_{i-1} \times p_i$, computing the matrix product $A_{i:k} A_{k+1:j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \ .$$

This recursive equation assumes that you know the value of k. But you don't, at least not yet. You have to try all possible values of k. How many are there? Just $j - i$, namely $k = i, i+1, \ldots, j-1$. Since the optimal parenthesization must use one of these values for k, you need only check them all to find the best. Thus, the recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min \{ m[i, k] + m[k+1, j] + p_{i-1} p_k p_j : i \le k < j \} & \text{if } i < j \ . \end{cases} \tag{14.7}$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information you need to construct an optimal solution. To help you do so, let's define $s[i, j]$ to be a value of k at which you split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$.

Step 3: Computing the optimal costs

At this point, you could write a recursive algorithm based on recurrence (14.7) to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 \cdots A_n$. But as we saw for the rod-cutting problem, and as we shall see in Section 14.3, this recursive algorithm takes exponential time. That's no better than the brute-force method of checking each way of parenthesizing the product.

Fortunately, there aren't all that many distinct subproblems: just one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all.[4] A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence (14.7) recursively, let's compute the optimal cost by using a tabular, bottom-up approach, as in the procedure MATRIX-CHAIN-ORDER. (The corresponding top-down approach using memoization appears in Section 14.3.) The input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of matrix dimensions, along with n, so that for $i = 1, 2, \ldots, n$, matrix A_i has dimensions $p_{i-1} \times p_i$. The procedure uses an auxiliary table $m[1:n, 1:n]$ to store the $m[i, j]$ costs and another auxiliary table $s[1:n-1, 2:n]$ that records which index k achieved the optimal cost in computing $m[i, j]$. The table s will help in constructing an optimal solution.

```
MATRIX-CHAIN-ORDER(p, n)
1  let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
2  for $i = 1$ to $n$                      // chain length 1
3      $m[i,i] = 0$
4  for $l = 2$ to $n$                      // $l$ is the chain length
5      for $i = 1$ to $n - l + 1$          // chain begins at $A_i$
6          $j = i + l - 1$                 // chain ends at $A_j$
7          $m[i, j] = \infty$
8          for $k = i$ to $j - 1$          // try $A_{i:k} A_{k+1:j}$
9              $q = m[i,k] + m[k+1, j] + p_{i-1} p_k p_j$
10             if $q < m[i, j]$
11                 $m[i, j] = q$           // remember this cost
12                 $s[i, j] = k$           // remember this index
13 return $m$ and $s$
```

In what order should the algorithm fill in the table entries? To answer this question, let's see which entries of the table need to be accessed when computing the

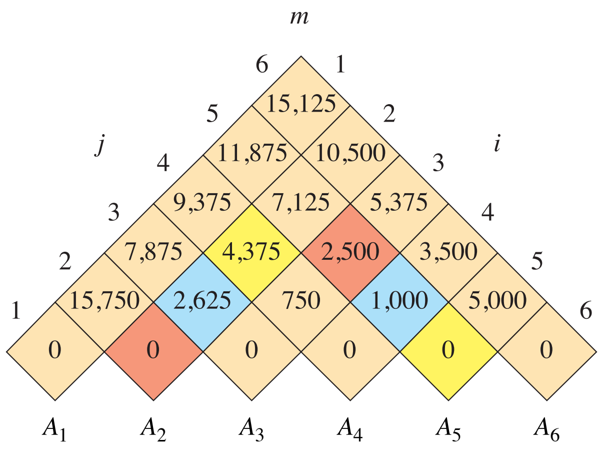
[4] The $\binom{n}{2}$ term counts all pairs in which $i < j$. Because i and j may be equal, we need to add in the n term.

cost $m[i, j]$. Equation (14.7) tells us that to compute the cost of matrix product $A_{i:j}$, first the costs of the products $A_{i:k}$ and $A_{k+1:j}$ need to have been computed for all $k = i, i+1, \ldots, j-1$. The chain $A_i A_{i+1} \cdots A_j$ consists of $j - i + 1$ matrices, and the chains $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$ consist of $k - i + 1$ and $j - k$ matrices, respectively. Since $k < j$, a chain of $k - i + 1$ matrices consists of fewer than $j - i + 1$ matrices. Likewise, since $k \geq i$, a chain of $j - k$ matrices consists of fewer than $j - i + 1$ matrices. Thus, the algorithm should fill in the table m from shorter matrix chains to longer matrix chains. That is, for the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, it makes sense to consider the subproblem size as the length $j - i + 1$ of the chain.

Now, let's see how the MATRIX-CHAIN-ORDER procedure fills in the $m[i, j]$ entries in order of increasing chain length. Lines 2–3 initialize $m[i, i] = 0$ for $i = 1, 2, \ldots, n$, since any matrix chain with just one matrix requires no scalar multiplications. In the **for** loop of lines 4–12, the loop variable l denotes the length of matrix chains whose minimum costs are being computed. Each iteration of this loop uses recurrence (14.7) to compute $m[i, i+l-1]$ for $i = 1, 2, \ldots, n-l+1$. In the first iteration, $l = 2$, and so the loop computes $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$: the minimum costs for chains of length $l = 2$. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$: the minimum costs for chains of length $l = 3$. And so on, ending with a single matrix chain of length $l = n$ and computing $m[1, n]$. When lines 7–12 compute an $m[i, j]$ cost, this cost depends only on table entries $m[i, k]$ and $m[k+1, j]$, which have already been computed.

Figure 14.5 illustrates the m and s tables, as filled in by the MATRIX-CHAIN-ORDER procedure on a chain of $n = 6$ matrices. Since $m[i, j]$ is defined only for $i \leq j$, only the portion of the table m on or above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, the minimum cost $m[i, j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices appears at the intersection of lines running northeast from A_i and northwest from A_j. Reading across, each diagonal in the table contains the entries for matrix chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1} p_k p_j$ for $k = i, i+1, \ldots, j-1$ and all entries southwest and southeast from $m[i, j]$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n-1$ values. Exercise 14.2-5 asks you to show that the running time of this algorithm is in fact also $\Omega(n^3)$. The algorithm requires $\Theta(n^2)$ space to store the m and s tables. Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.



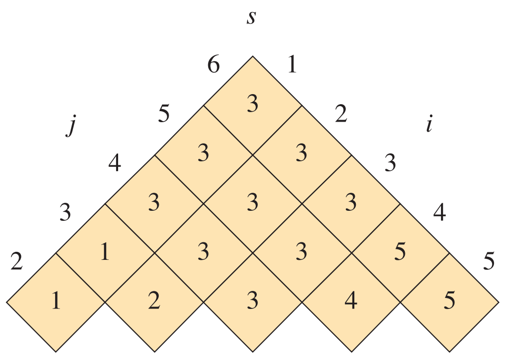


Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the entries that are not tan, the pairs that have the same color are taken together in line 9 when computing

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125 .
\end{aligned}
$$

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1:n-1, 2:n]$ provides the information needed to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. The final matrix multiplication in computing $A_{1:n}$ optimally is $A_{1:s[1,n]} A_{s[1,n]+1:n}$. The s table contains the information needed to determine the earlier matrix multiplications as well, using recursion: $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_{1:s[1,n]}$ and $s[s[1,n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1:n}$. The recursive procedure PRINT-OPTIMAL-PARENS on the facing page prints an optimal parenthesization of the matrix chain product $A_i A_{i+1} \cdots A_j$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of the full matrix chain product $A_1 A_2 \cdots A_n$. In the example of Figure 14.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the optimal parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

Exercises

14.2-1
Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

14.2-2
Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call is MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.) Assume that the call RECTANGULAR-MATRIX-MULTIPLY(A, B) returns the product of matrices A and B.

14.2-3
Use the substitution method to show that the solution to the recurrence (14.6) is $\Omega(2^n)$.

14.2-4
Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

14.2-5
Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3-n}{3} .$$

(*Hint:* You may find equation (A.4) on page 1141 useful.)

14.2-6

Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

14.3 Elements of dynamic programming

Although you have just seen two complete examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should you look for a dynamic-programming solution to a problem? In this section, we'll examine the two key ingredients that an optimization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems. We'll also revisit and discuss more fully how memoization might help you take advantage of the overlapping-subproblems property in a top-down recursive approach.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. When a problem exhibits optimal substructure, that gives you a good clue that dynamic programming might apply. (As Chapter 15 discusses, it also might mean that a greedy strategy applies, however.) Dynamic programming builds an optimal solution to the problem from optimal solutions to subproblems. Consequently, you must take care to ensure that the range of subproblems you consider includes those used in an optimal solution.

Optimal substructure was key to solving both of the previous problems in this chapter. In Section 14.1, we observed that the optimal way of cutting up a rod of length n (if Serling Enterprises makes any cuts at all) involves optimally cutting up the two pieces resulting from the first cut. In Section 14.2, we noted that an optimal parenthesization of the matrix chain product $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by "cutting out" the nonoptimal solution to each subproblem and "pasting in" the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

To characterize the space of subproblems, a good rule of thumb says to try to keep the space as simple as possible and then expand it as necessary. For example, the space of subproblems for the rod-cutting problem contained the problems of optimally cutting up a rod of length i for each size i. This subproblem space worked well, and it was not necessary to try a more general space of subproblems.

Conversely, suppose that you tried to constrain the subproblem space for matrix-chain multiplication to matrix products of the form $A_1 A_2 \cdots A_j$. As before, an optimal parenthesization must split this product between A_k and A_{k+1} for some $1 \le k < j$. Unless you can guarantee that k always equals $j - 1$, you will find that you have subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. Moreover, the latter subproblem does not have the form $A_1 A_2 \cdots A_j$. To solve this problem by dynamic programming, you need to allow the subproblems to vary at "both ends." That is, both i and j need to vary in the subproblem of parenthesizing the product $A_i A_{i+1} \cdots A_j$.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems an optimal solution to the original problem uses, and
2. how many choices you have in determining which subproblem(s) to use in an optimal solution.

In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n-i$), but we have to consider n choices for i in order to determine which one yields an optimal solution. Matrix-chain multiplication for the subchain $A_i A_{i+1} \cdots A_j$ serves an example with two subproblems and $j-i$ choices. For a given matrix A_k where the product splits, two subproblems arise—parenthesizing $A_i A_{i+1} \cdots A_k$ and parenthesizing $A_{k+1} A_{k+2} \cdots A_j$—and we have to solve *both* of them optimally. Once we determine the optimal solutions to subproblems, we choose from among $j-i$ candidates for the index k.

Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices you look at for each subproblem. In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O(n^2)$ running time. Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and each had at most $n-1$ choices, giving an $O(n^3)$ running time (actually, a $\Theta(n^3)$ running time, by Exercise 14.2-5).

Usually, the subproblem graph gives an alternative way to perform the same analysis. Each vertex corresponds to a subproblem, and the choices for a subproblem are the edges incident from that subproblem. Recall that in rod cutting, the subproblem graph has n vertices and at most n edges per vertex, yielding an $O(n^2)$ running time. For matrix-chain multiplication, if you were to draw the subproblem graph, it would have $\Theta(n^2)$ vertices and each vertex would have degree at most $n-1$, giving a total of $O(n^3)$ vertices and edges.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, you first find optimal solutions to subproblems and, having solved the subproblems, you find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among subproblems as to which you will use in solving the problem. The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself. In rod cutting, for example, first we solved the subproblems of determining optimal ways to cut up rods of length i for $i = 0, 1, \ldots, n-1$, and then we determined which of these subproblems yielded an optimal solution for a rod of length n, using equation (14.2). The cost attributable to the choice itself is the term p_i in equation (14.2). In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i-1} p_k p_j$.

Chapter 15 explores "greedy algorithms," which have many similarities to dynamic programming. In particular, problems to which greedy algorithms apply have optimal substructure. One major difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a "greedy" choice—the choice that looks best at the time—and then solve a resulting subproblem, without bothering to solve all possible related smaller subproblems. Surprisingly, in some cases this strategy works!

Subtleties

You should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems whose input consists of a directed graph $G = (V, E)$ and vertices $u, v \in V$.

Unweighted shortest path:[5] Find a path from u to v consisting of the fewest edges. Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. (Without the requirement that the path must be simple, the problem is undefined, since repeatedly traversing a cycle creates paths with an arbitrarily large number of edges.)

The unweighted shortest-path problem exhibits optimal substructure. Here's how. Suppose that $u \neq v$, so that the problem is nontrivial. Then, any path p from u to v must contain an intermediate vertex, say w. (Note that w may be u or v.) Then, we can decompose the path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$. The number of edges in p equals the number of edges in p_1 plus the number of edges in p_2. We claim that if p is an optimal (i.e., shortest) path from u to v, then p_1 must be a shortest path from u to w. Why? As suggested earlier, use a "cut-and-paste" argument: if there were another path, say p'_1, from u to w with fewer edges than p_1, then we could cut out p_1 and paste in p'_1 to produce a path $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$ with fewer edges than p, thus contradicting p's optimality. Likewise, p_2 must be a shortest path from w to v. Thus, to find a shortest path from u to v, consider all intermediate vertices w, find a shortest path from u to w and a shortest path from w to v, and choose an intermediate vertex w that yields the overall shortest path. Section 23.2 uses a variant of this observation of optimal substructure to find a shortest path between every pair of vertices on a weighted, directed graph.

You might be tempted to assume that the problem of finding an unweighted longest simple path exhibits optimal substructure as well. After all, if we decompose a longest simple path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, then mustn't p_1 be a longest simple path from u to w, and mustn't p_2 be a longest simple path from w to v? The answer is no! Figure 14.6 supplies an example. Consider the

[5] We use the term "unweighted" to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 22 and 23. You can use the breadth-first search technique of Chapter 20 to solve the unweighted problem.

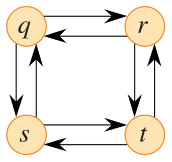


Figure 14.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

path $q \to r \to t$, which is a longest simple path from q to t. Is $q \to r$ a longest simple path from q to r? No, for the path $q \to s \to t \to r$ is a simple path that is longer. Is $r \to t$ a longest simple path from r to t? No again, for the path $r \to q \to s \to t$ is a simple path that is longer.

This example shows that for longest simple paths, not only does the problem lack optimal substructure, but you cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems. If you combine the longest simple paths $q \to s \to t \to r$ and $r \to q \to s \to t$, you get the path $q \to s \to t \to r \to q \to s \to t$, which is not simple. Indeed, the problem of finding an unweighted longest simple path does not appear to have any sort of optimal substructure. No efficient dynamic-programming algorithm for this problem has ever been found. In fact, this problem is NP-complete, which—as we shall see in Chapter 34—means that we are unlikely to find a way to solve it in polynomial time.

Why is the substructure of a longest simple path so different from that of a shortest path? Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not ***independent***, whereas for shortest paths they are. What do we mean by subproblems being independent? We mean that the solution to one subproblem does not affect the solution to another subproblem of the same problem. For the example of Figure 14.6, we have the problem of finding a longest simple path from q to t with two subproblems: finding longest simple paths from q to r and from r to t. For the first of these subproblems, we chose the path $q \to s \to t \to r$, which used the vertices s and t. These vertices cannot appear in a solution to the second subproblem, since the combination of the two solutions to subproblems yields a path that is not simple. If vertex t cannot be in the solution to the second problem, then there is no way to solve it, since t is required to be on the path that forms the solution, and it is not the vertex where the subproblem solutions are "spliced" together (that vertex being r). Because vertices s and t appear in one subproblem solution, they cannot appear in the other subproblem solution. One of them must be in the solution to the other subproblem, however, and an optimal solution requires both.

Thus, we say that these subproblems are not independent. Looked at another way, using resources in solving one subproblem (those resources being vertices) renders them unavailable for the other subproblem.

Why, then, are the subproblems independent for finding a shortest path? The answer is that by nature, the subproblems do not share resources. We claim that if a vertex w is on a shortest path p from u to v, then we can splice together *any* shortest path $u \overset{p_1}{\rightsquigarrow} w$ and *any* shortest path $w \overset{p_2}{\rightsquigarrow} v$ to produce a shortest path from u to v. We are assured that, other than w, no vertex can appear in both paths p_1 and p_2. Why? Suppose that some vertex $x \neq w$ appears in both p_1 and p_2, so that we can decompose p_1 as $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ and p_2 as $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. By the optimal substructure of this problem, path p has as many edges as p_1 and p_2 together. Let's say that p has e edges. Now let us construct a path $p' = u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ from u to v. Because we have excised the paths from x to w and from w to x, each of which contains at least one edge, path p' contains at most $e - 2$ edges, which contradicts the assumption that p is a shortest path. Thus, we are assured that the subproblems for the shortest-path problem are independent.

The two problems examined in Sections 14.1 and 14.2 have independent subproblems. In matrix-chain multiplication, the subproblems are multiplying subchains $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. These subchains are disjoint, so that no matrix could possibly be included in both of them. In rod cutting, to determine the best way to cut up a rod of length n, we looked at the best ways of cutting up rods of length i for $i = 0, 1, \ldots, n-1$. Because an optimal solution to the length-n problem includes just one of these subproblem solutions (after cutting off the first piece), independence of subproblems is not an issue.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems***.[6] In contrast, a problem for which a divide-and-

[6] It may seem strange that dynamic programming relies on subproblems being both independent and overlapping. Although these requirements may sound contradictory, they describe two different notions, rather than two points on the same axis. Two subproblems of the same problem are independent if they do not share resources. Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.

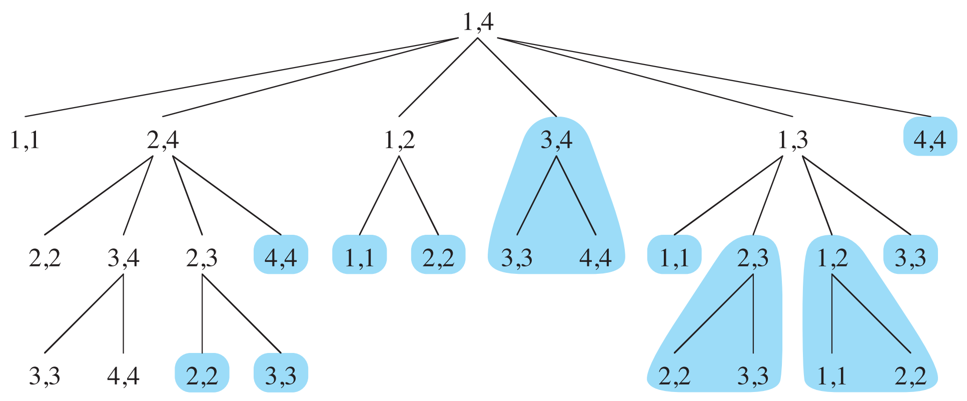


Figure 14.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a subtree shaded blue are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

In Section 14.1, we briefly examined how a recursive solution to rod cutting makes exponentially many calls to find solutions of smaller subproblems. The dynamic-programming solution reduces the running time from the exponential time of the recursive algorithm down to quadratic time.

To illustrate the overlapping-subproblems property in greater detail, let's revisit the matrix-chain multiplication problem. Referring back to Figure 14.5, observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, it references entry $m[3, 4]$ four times: during the computations of $m[2, 4]$, $m[1, 4]$, $m[3, 5]$, and $m[3, 6]$. If the algorithm were to recompute $m[3, 4]$ each time, rather than just looking it up, the running time would increase dramatically. To see how, consider the inefficient recursive procedure RECURSIVE-MATRIX-CHAIN on the facing page, which determines $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A_{i:j} = A_i A_{i+1} \cdots A_j$. The procedure is based directly on the recurrence (14.7). Figure 14.7 shows the recursion tree produced by the call RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node is labeled by the values of the parameters i and j. Observe that some pairs of values occur many times.

In fact, the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. To see why, let $T(n)$ denote the time taken by RECURSIVE-MATRIX-

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  $m[i, j] = \infty$
4  for $k = i$ to $j - 1$
5      $q$ = RECURSIVE-MATRIX-CHAIN(p, i, k)
             + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
             + $p_{i-1} p_k p_j$
6      if $q < m[i, j]$
7          $m[i, j] = q$
8  return $m[i, j]$
```

CHAIN to compute an optimal parenthesization of a chain of n matrices. Because the execution of lines 1–2 and of lines 6–7 each take at least unit time, as does the multiplication in line 5, inspection of the procedure yields the recurrence

$$T(n) \ge \begin{cases} 1 & \text{if } n = 1 \,, \\ 1 + \displaystyle\sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) & \text{if } n > 1 \,. \end{cases}$$

Noting that for $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \ge 2 \sum_{i=1}^{n-1} T(i) + n \,. \tag{14.8}$$

Let's prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we'll show that $T(n) \ge 2^{n-1}$ for all $n \ge 1$. For the base case $n = 1$, the summation is empty, and we get $T(1) \ge 1 = 2^0$. Inductively, for $n \ge 2$ we have

$$\begin{aligned} T(n) &\ge 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{j=0}^{n-2} 2^j + n && \text{(letting } j = i - 1\text{)} \\ &= 2(2^{n-1} - 1) + n && \text{(by equation (A.6) on page 1142)} \\ &= 2^n - 2 + n \\ &\ge 2^{n-1} \,, \end{aligned}$$

which completes the proof. Thus, the total amount of work performed by the call RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ is at least exponential in n.

Compare this top-down, recursive algorithm (without memoization) with the bottom-up dynamic-programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property. Matrix-chain multiplication has only $\Theta(n^2)$ distinct subproblems, and the dynamic-programming algorithm solves each exactly once. The recursive algorithm, on the other hand, must solve each subproblem every time it reappears in the recursion tree. Whenever a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly, and the total number of distinct subproblems is small, dynamic programming can improve efficiency, sometimes dramatically.

Reconstructing an optimal solution

As a practical matter, you'll often want to store in a separate table which choice you made in each subproblem so that you do not have to reconstruct this information from the table of costs.

For matrix-chain multiplication, the table $s[i, j]$ saves a significant amount of work when we need to reconstruct an optimal solution. Suppose that the MATRIX-CHAIN-ORDER procedure on page 378 did not maintain the $s[i, j]$ table, so that it filled in only the table $m[i, j]$ containing optimal subproblem costs. The procedure chooses from among $j - i$ possibilities when determining which subproblems to use in an optimal solution to parenthesizing $A_i A_{i+1} \cdots A_j$, and $j - i$ is not a constant. Therefore, it would take $\Theta(j-i) = \omega(1)$ time to reconstruct which subproblems it chose for a solution to a given problem. Because MATRIX-CHAIN-ORDER stores in $s[i, j]$ the index of the matrix at which it split the product $A_i A_{i+1} \cdots A_j$, the PRINT-OPTIMAL-PARENS procedure on page 381 can look up each choice in $O(1)$ time.

Memoization

As we saw for the rod-cutting problem, there is an alternative approach to dynamic programming that often offers the efficiency of the bottom-up dynamic-programming approach while maintaining a top-down strategy. The idea is to ***memoize*** the natural, but inefficient, recursive algorithm. As in the bottom-up approach, you maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table.

Each subsequent encounter of this subproblem simply looks up the value stored in the table and returns it.[7]

The procedure Memoized-Matrix-Chain is a memoized version of the procedure Recursive-Matrix-Chain on page 389. Note where it resembles the memoized top-down method on page 369 for the rod-cutting problem.

```
Memoized-Matrix-Chain$(p, n)$
1  let $m[1:n, 1:n]$ be a new table
2  for $i = 1$ to $n$
3      for $j = i$ to $n$
4          $m[i, j] = \infty$
5  return Lookup-Chain$(m, p, 1, n)$

Lookup-Chain$(m, p, i, j)$
1  if $m[i, j] < \infty$
2      return $m[i, j]$
3  if $i == j$
4      $m[i, j] = 0$
5  else for $k = i$ to $j - 1$
6          $q =$ Lookup-Chain$(m, p, i, k)$
               $+$ Lookup-Chain$(m, p, k+1, j) + p_{i-1} p_k p_j$
7          if $q < m[i, j]$
8              $m[i, j] = q$
9  return $m[i, j]$
```

The Memoized-Matrix-Chain procedure, like the bottom-up Matrix-Chain-Order procedure on page 378, maintains a table $m[1:n, 1:n]$ of computed values of $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix $A_{i:j}$. Each table entry initially contains the value ∞ to indicate that the entry has yet to be filled in. Upon calling Lookup-Chain(m, p, i, j), if line 1 finds that $m[i, j] < \infty$, then the procedure simply returns the previously computed cost $m[i, j]$ in line 2. Otherwise, the cost is computed as in Recursive-Matrix-Chain, stored in $m[i, j]$, and returned. Thus, Lookup-Chain(m, p, i, j) always returns the value of $m[i, j]$, but it computes it only upon the first call of Lookup-Chain with these specific values of i and j.

[7] This approach presupposes that you know the set of all possible subproblem parameters and that you have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys.

Figure 14.7 illustrates how MEMOIZED-MATRIX-CHAIN saves time compared with RECURSIVE-MATRIX-CHAIN. Subtrees shaded blue represent values that are looked up rather than recomputed.

Like the bottom-up procedure MATRIX-CHAIN-ORDER, the memoized procedure MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time. To begin with, line 4 of MEMOIZED-MATRIX-CHAIN executes $\Theta(n^2)$ times, which dominates the running time outside of the call to LOOKUP-CHAIN in line 5. We can categorize the calls of LOOKUP-CHAIN into two types:

1. calls in which $m[i, j] = \infty$, so that lines 3–9 execute, and
2. calls in which $m[i, j] < \infty$, so that LOOKUP-CHAIN simply returns in line 2.

There are $\Theta(n^2)$ calls of the first type, one per table entry. All calls of the second type are made as recursive calls by calls of the first type. Whenever a given call of LOOKUP-CHAIN makes recursive calls, it makes $O(n)$ of them. Therefore, there are $O(n^3)$ calls of the second type in all. Each call of the second type takes $O(1)$ time, and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls. The total time, therefore, is $O(n^3)$. Memoization thus turns an $\Omega(2^n)$-time algorithm into an $O(n^3)$-time algorithm.

We have seen how to solve the matrix-chain multiplication problem by either a top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming algorithm in $O(n^3)$ time. Both the bottom-up and memoized methods take advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once. Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblems are repeatedly solved.

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table. Moreover, for some problems you can exploit the regular pattern of table accesses in the dynamic-programming algorithm to reduce time or space requirements even further. On the other hand, in certain situations, some of the subproblems in the subproblem space might not need to be solved at all. In that case, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercises

14.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

14.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

14.3-3

Consider the antithetical variant of the matrix-chain multiplication problem where the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

14.3-4

As stated, in dynamic programming, you first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that she does not always need to solve all the subproblems in order to find an optimal solution. She suggests that she can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

14.3-5

Suppose that the rod-cutting problem of Section 14.1 also had a limit l_i on the number of pieces of length i allowed to be produced, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 14.1 no longer holds.

14.4 Longest common subsequence

Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called ***bases***, where the possible bases are adenine, cytosine, guanine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the 4-element set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. (See Section C.1 for the definition of a string.) For example, the DNA of one organism may be $S_1 = \mathtt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$, and the DNA of another organism may be $S_2 = \mathtt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. We can, and do, define similarity in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. (Chapter 32 explores algorithms to solve this problem.) In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 14-5 looks at this notion.) Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2. These bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is `GTCGTCGGAAGCCGGCCGAA`.

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with 0 or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. The sequence $\langle B, C, A \rangle$ is not a *longest* common subsequence (***LCS***) of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both sequences X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the ***longest-common-subsequence problem***, the input is two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and the goal is to find a maximum-length common subsequence of X and Y. This section shows how to efficiently solve the LCS problem using dynamic programming.

Step 1: Characterizing a longest common subsequence

You can solve the LCS problem with a brute-force approach: enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence you find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, however, as the following theorem shows. As we'll see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be precise, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith ***prefix*** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Theorem 14.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k-1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2). ■

The way that Theorem 14.1 characterizes longest common subsequences says that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property, as we'll see in a moment.

Step 2: A recursive solution

Theorem 14.1 implies that you should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$, you need to find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then you have to solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.

Whichever of these two LCSs is longer is an LCS of X and Y. Because these cases exhaust all possibilities, one of the optimal subproblem solutions must appear within an LCS of X and Y.

The LCS problem has the overlapping-subproblems property. Here's how. To find an LCS of X and Y, you might need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, solving the LCS problem recursively involves establishing a recurrence for the value of an optimal solution. Let's define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases} \tag{14.9}$$

In this recursive formulation, a condition in the problem restricts which subproblems to consider. When $x_i = y_j$, you can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}. Otherwise, you instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j. In the previous dynamic-programming algorithms we have examined—for rod cutting and matrix-chain multiplication—we didn't rule out any subproblems due to conditions in the problem. Finding an LCS is not the only dynamic-programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem (see Problem 14-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (14.9), you could write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS problem has only $\Theta(mn)$ distinct subproblems (computing $c[i, j]$ for $0 \le i \le m$ and $0 \le j \le n$), dynamic programming can compute the solutions bottom up.

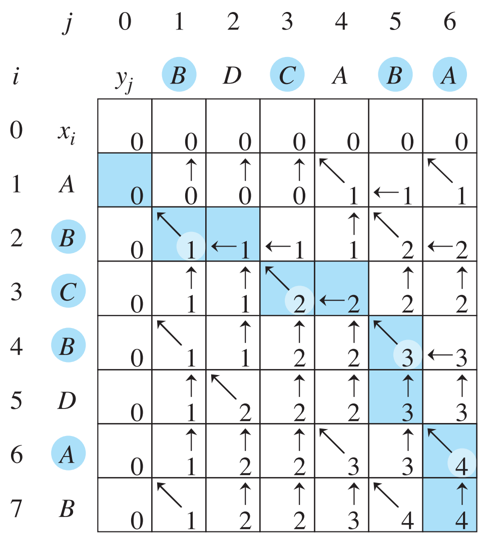
The procedure LCS-LENGTH on the next page takes two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as inputs, along with their lengths. It stores the $c[i, j]$ values in a table $c[0:m, 0:n]$, and it computes the entries in ***row-major*** order. That is, the procedure fills in the first row of c from left to right, then the second row, and so on. The procedure also maintains the table $b[1:m, 1:n]$ to help in constructing an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables, where $c[m, n]$ contains the length of an LCS of X and Y. Figure 14.8 shows the tables produced by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

```
LCS-LENGTH(X, Y, m, n)
let $b[1:m, 1:n]$ and $c[0:m, 0:n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$          // compute table entries in row-major order
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "$\nwarrow$"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] =$ "$\uparrow$"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] =$ "$\leftarrow$"
return $c$ and $b$

PRINT-LCS(b, X, i, j)
if $i == 0$ or $j == 0$
    return                  // the LCS has length 0
if $b[i, j] ==$ "$\nwarrow$"
    PRINT-LCS$(b, X, i-1, j-1)$
    print $x_i$             // same as $y_j$
elseif $b[i, j] ==$ "$\uparrow$"
    PRINT-LCS$(b, X, i-1, j)$
else PRINT-LCS$(b, X, i, j-1)$
```

Step 4: Constructing an LCS

With the b table returned by LCS-LENGTH, you can quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Begin at $b[m, n]$ and trace through the table by following the arrows. Each "$\nwarrow$" encountered in an entry $b[i, j]$ implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found. This method gives you the elements of this LCS in reverse order. The recursive procedure PRINT-LCS prints out an LCS of X and Y in the proper, forward order.



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 14.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner, as shown by the sequence shaded blue. Each "$\nwarrow$" on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

The initial call is PRINT-LCS(b, X, m, n). For the b table in Figure 14.8, this procedure prints $BCBA$. The procedure takes $O(m + n)$ time, since it decrements at least one of i and j in each recursive call.

Improving the code

Once you have developed an algorithm, you will often find that you can improve on the time or space it uses. Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance. Others can yield substantial asymptotic savings in time and space.

In the LCS algorithm, for example, you can eliminate the b table altogether. Each $c[i, j]$ entry depends on only three other c table entries: $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given the value of $c[i, j]$, you can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b. Thus, you can reconstruct an LCS in $O(m+n)$ time using a procedure similar to PRINT-LCS. (Exercise 14.4-2 asks you to give the pseudocode.) Although this method saves $\Theta(mn)$ space, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since the c table takes $\Theta(mn)$ space anyway.

You can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table c at a time: the row being computed and the previous row. (In fact, as Exercise 14.4-4 asks you to show, you can use only slightly more than the space for one row of c to compute the length of an LCS.) This improvement works if you need only the length of an LCS. If you need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace the algorithm's steps in $O(m + n)$ time.

Exercises

14.4-1

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

14.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

14.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

14.4-4

Show how to compute the length of an LCS using only $2 \cdot \min\{m, n\}$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min\{m, n\}$ entries plus $O(1)$ additional space.

14.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

★ ***14.4-6***

Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* The last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i-1$. Maintain candidate subsequences by linking them through the input sequence.)

14.5 Optimal binary search trees

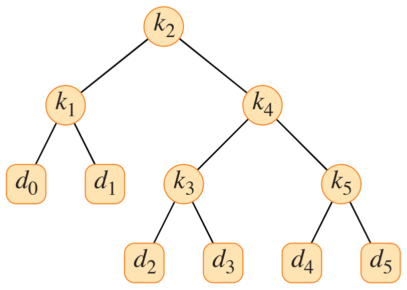
Suppose that you are designing a program to translate text from English to Latvian. For each occurrence of each English word in the text, you need to look up its Latvian equivalent. You can perform these lookup operations by building a binary search tree with n English words as keys and their Latvian equivalents as satellite data. Because you will search the tree for each individual word in the text, you want the total time spent searching to be as low as possible. You can ensure an $O(\lg n)$ search time per occurrence by using a red-black tree or any other balanced binary search tree. Words appear with different frequencies, however, and a frequently used word such as *the* can end up appearing far from the root while a rarely used word such as *naumachia* appears near the root. Such an organization would slow down the translation, since the number of nodes visited when searching for a key in a binary search tree equals 1 plus the depth of the node containing the key. You want words that occur frequently in the text to be placed nearer the root.[8] Moreover, some words in the text might have no Latvian translation,[9] and such words would not appear in the binary search tree at all. How can you organize a binary search tree so as to minimize the number of nodes visited in all searches, given that you know how often each word occurs?

What you need is an ***optimal binary search tree***. Formally, given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys such that $k_1 < k_2 < \cdots < k_n$, build a binary search tree containing them. For each key k_i, you are given the probability p_i that any given search is for key k_i. Since some searches may be for values not in K, you also have $n + 1$ "dummy" keys $d_0, d_1, d_2, \ldots, d_n$ representing those values. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, you have the probability q_i that a search corresponds to d_i. Figure 14.9 shows two binary search trees for a set of $n = 5$ keys. Each key k_i is an internal node, and each dummy key d_i is a leaf. Since every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \tag{14.10}$$

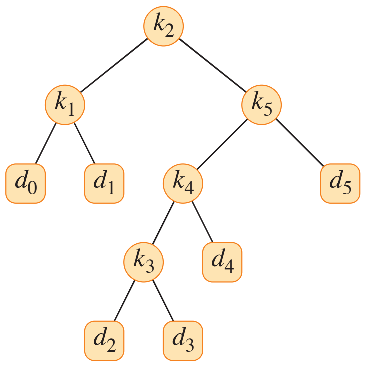
[8] If the subject of the text is ancient Rome, you might want *naumachia* to appear near the root.

[9] Yes, *naumachia* has a Latvian counterpart: *nomačija*.



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

(a)



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

(b)

Figure 14.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.

Knowing the probabilities of searches for each key and each dummy key allows us to determine the expected cost of a search in a given binary search tree T. Let us assume that the actual cost of a search equals the number of nodes examined, which is the depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is

$$\begin{aligned} \text{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \;, \qquad (14.11) \end{aligned}$$

where depth_T denotes a node's depth in the tree T. The last equation follows from equation (14.10). Figure 14.9 shows how to calculate the expected search cost node by node.

For a given set of probabilities, your goal is to construct a binary search tree whose expected search cost is smallest. We call such a tree an ***optimal binary search tree***. Figure 14.9(a) shows one binary search tree, with expected cost 2.80, for the probabilities given in the figure caption. Part (b) of the figure displays an optimal binary search tree, with expected cost 2.75. This example demonstrates that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor does an optimal binary search tree always have the key with the greatest probability at the root. Here, key k_5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k_2. (The lowest expected cost of any binary search tree with k_5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. You can label the nodes of any n-node binary tree with the keys $k_1, k_2, \ldots, k_n$ to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4 on page 329, we saw that the number of binary trees with n nodes is $\Omega(4^n/n^{3/2})$. Thus you would need to examine an exponential number of binary search trees to perform an exhaustive search. We'll see how to solve this problem more efficiently with dynamic programming.

Step 1: The structure of an optimal binary search tree

To characterize the optimal substructure of optimal binary search trees, we start with an observation about subtrees. Consider any subtree of a binary search tree. It must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.

Now we can state the optimal substructure: if an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. The usual cut-and-paste argument applies. If there were a subtree T'' whose expected cost is lower than that of T', then cutting T' out of T and pasting in T'' would result in a binary search tree of lower expected cost than T, thus contradicting the optimality of T.

With the optimal substructure in hand, here is how to construct an optimal solution to the problem from optimal solutions to subproblems. Given keys $k_i, \ldots, k_j$, one of these keys, say k_r ($i \le r \le j$), is the root of an optimal subtree containing these keys. The left subtree of the root k_r contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$), and the right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$). As long as you examine all candidate roots k_r, where $i \le r \le j$, and you determine all optimal binary search trees containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$, you are guaranteed to find an optimal binary search tree.

There is one technical detail worth understanding about "empty" subtrees. Suppose that in a subtree with keys $k_i, \ldots, k_j$, you select k_i as the root. By the above argument, k_i's left subtree contains the keys $k_i, \ldots, k_{i-1}$: no keys at all. Bear in mind, however, that subtrees also contain dummy keys. We adopt the convention that a subtree containing keys $k_i, \ldots, k_{i-1}$ has no actual keys but does contain the single dummy key d_{i-1}. Symmetrically, if you select k_j as the root, then k_j's right subtree contains the keys $k_{j+1}, \ldots, k_j$. This right subtree contains no actual keys, but it does contain the dummy key d_j.

Step 2: A recursive solution

To define the value of an optimal solution recursively, the subproblem domain is finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \ge 1$, $j \le n$, and $j \ge i - 1$. (When $j = i - 1$, there is just the dummy key d_{i-1}, but no actual keys.) Let $e[i, j]$ denote the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Your goal is to compute $e[1, n]$, the expected cost of searching an optimal binary search tree for all the actual and dummy keys.

The easy case occurs when $j = i - 1$. Then the subproblem consists of just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.

When $j \ge i$, you need to select a root k_r from among $k_i, \ldots, k_j$ and then make an optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree and an optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (14.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys $k_i, \ldots, k_j$, denote this sum of probabilities as

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \; . \tag{14.12}$$

Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) \; .$$

Noting that

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j) \; ,$$

we rewrite $e[i, j]$ as

$$e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j) . \tag{14.13}$$

The recursive equation (14.13) assumes that you know which node k_r to use as the root. Of course, you choose the root that gives the lowest expected search cost, giving the final recursive formulation:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 , \\ \min\{e[i, r - 1] + e[r + 1, j] + w(i, j) : i \le r \le j\} & \text{if } i \le j . \end{cases} \tag{14.14}$$

The $e[i, j]$ values give the expected search costs in optimal binary search trees. To help keep track of the structure of optimal binary search trees, define *root*$[i, j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. Although we'll see how to compute the values of *root*$[i, j]$, the construction of an optimal binary search tree from these values is left as Exercise 14.5-1.

Step 3: Computing the expected search cost of an optimal binary search tree

At this point, you may have noticed some similarities between our characterizations of optimal binary search trees and matrix-chain multiplication. For both problem domains, the subproblems consist of contiguous index subranges. A direct, recursive implementation of equation (14.14) would be just as inefficient as a direct, recursive matrix-chain multiplication algorithm. Instead, you can store the $e[i, j]$ values in a table $e[1:n+1, 0:n]$. The first index needs to run to $n+1$ rather than n because in order to have a subtree containing only the dummy key d_n, you need to compute and store $e[n + 1, n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0, you need to compute and store $e[1, 0]$. Only the entries $e[i, j]$ for which $j \ge i - 1$ are filled in. The table *root*$[i, j]$ records the root of the subtree containing keys $k_i, \ldots, k_j$ and uses only the entries for which $1 \le i \le j \le n$.

One other table makes the dynamic-programming algorithm a little faster. Instead of computing the value of $w(i, j)$ from scratch every time you compute $e[i, j]$, which would take $\Theta(j - i)$ additions, store these values in a table $w[1:n + 1, 0:n]$. For the base case, compute $w[i, i-1] = q_{i-1}$ for $1 \le i \le n+1$. For $j \ge i$, compute

$$w[i, j] = w[i, j - 1] + p_j + q_j . \tag{14.15}$$

Thus, you can compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.

The OPTIMAL-BST procedure on the next page takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and it returns the tables e and *root*. From the description above and the similarity to the MATRIX-CHAIN-ORDER procedure in Section 14.2, you should find the operation of this procedure to be fairly straightforward. The **for** loop of lines 2–4 initializes the values of $e[i, i-1]$ and $w[i, i-1]$. Then the **for** loop of lines 5–14 uses the recurrences (14.14) and (14.15) to compute $e[i, j]$ and $w[i, j]$ for all $1 \le i \le j \le n$. In the first iteration, when $l = 1$, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1, 2, \ldots, n$. The second iteration, with $l = 2$, computes $e[i, i+1]$ and $w[i, i+1]$ for $i = 1, 2, \ldots, n-1$, and so on. The innermost **for** loop, in lines 10–14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. This **for** loop saves the current value of the index r in *root*$[i, j]$ whenever it finds a better key to use as the root.

```
OPTIMAL-BST($p, q, n$)
 1  let $e[1:n+1, 0:n]$, $w[1:n+1, 0:n]$,
            and $root[1:n, 1:n]$ be new tables
 2  for $i = 1$ to $n+1$                  // base cases
 3      $e[i, i-1] = q_{i-1}$             // equation (14.14)
 4      $w[i, i-1] = q_{i-1}$
 5  for $l = 1$ to $n$
 6      for $i = 1$ to $n - l + 1$
 7          $j = i + l - 1$
 8          $e[i, j] = \infty$
 9          $w[i, j] = w[i, j-1] + p_j + q_j$          // equation (14.15)
10          for $r = i$ to $j$                          // try all possible roots $r$
11              $t = e[i, r-1] + e[r+1, j] + w[i, j]$   // equation (14.14)
12              if $t < e[i, j]$                        // new minimum?
13                  $e[i, j] = t$
14                  $root[i, j] = r$
15  return $e$ and $root$
```

Figure 14.10 shows the tables $e[i, j]$, $w[i, j]$, and *root*$[i, j]$ computed by the procedure OPTIMAL-BST on the key distribution shown in Figure 14.9. As in the matrix-chain multiplication example of Figure 14.5, the tables are rotated to make the diagonals run horizontally. OPTIMAL-BST computes the rows from bottom to top and from left to right within each row.

The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER. Its running time is $O(n^3)$, since its **for** loops are nested three deep and each loop index takes on at most n values. The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within at most 1 in all directions. Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.

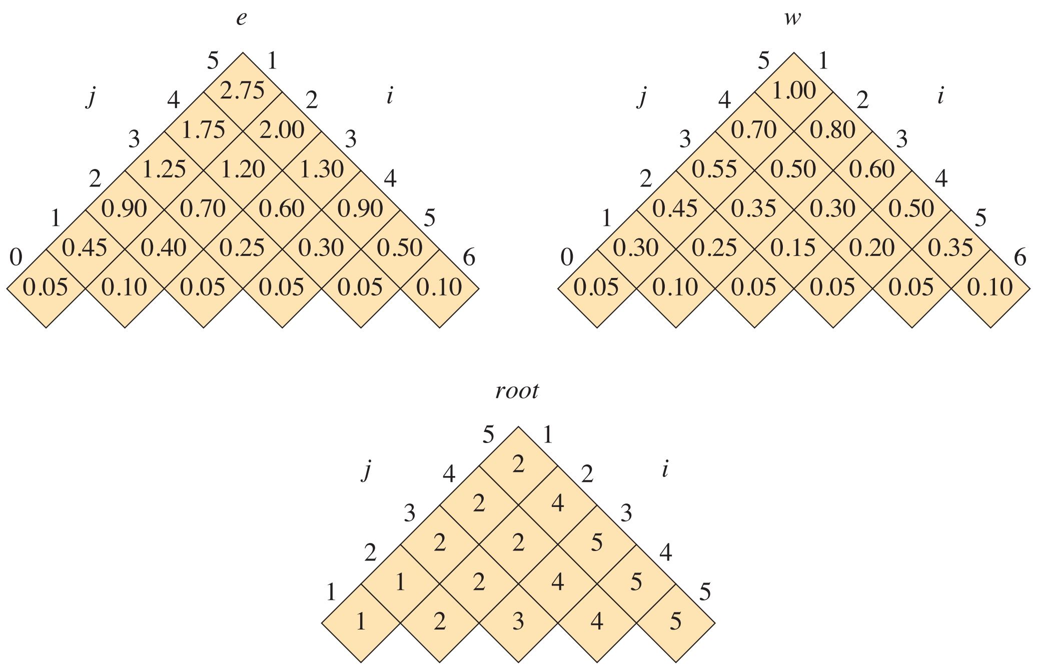


Figure 14.10 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 14.9. The tables are rotated so that the diagonals run horizontally.

Exercises

14.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST$(root, n)$ which, given the table $root[1:n, 1:n]$, outputs the structure of an optimal binary search tree. For the example in Figure 14.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 14.9(b).

14.5-2
Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

14.5-3
Suppose that instead of maintaining the table $w[i, j]$, you computed the value of $w(i, j)$ directly from equation (14.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

★ ***14.5-4***
Knuth [264] has shown that there are always roots of optimal subtrees such that $root[i, j-1] \le root[i, j] \le root[i+1, j]$ for all $1 \le i < j \le n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Problems

14-1 Longest simple path in a directed acyclic graph
You are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. The ***weight*** of a path is the sum of the weights of the edges in the path. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What is the running time of your algorithm?

14-2 Longest palindrome subsequence
A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

14-3 Bitonic euclidean traveling-salesperson problem
In the ***euclidean traveling-salesperson problem***, you are given a set of n points in the plane, and your goal is to find the shortest closed tour that connects all n points.

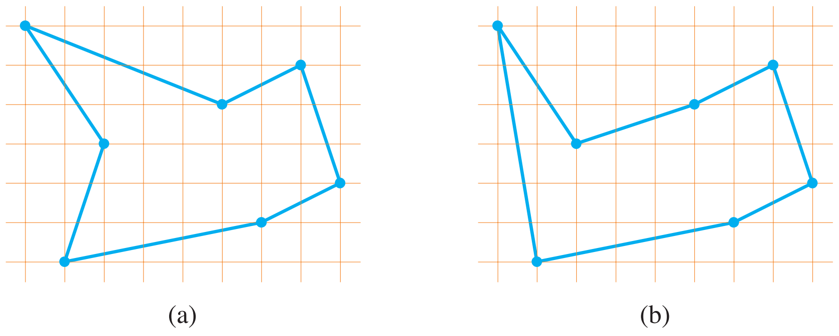


Figure 14.11 Seven points in the plane, shown on a unit grid. **(a)** The shortest closed tour, with length approximately 24.89. This tour is not bitonic. **(b)** The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

Figure 14.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).

J. L. Bentley has suggested simplifying the problem by considering only ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 14.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

14-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width). The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters, which are to be printed neatly on a number of lines that hold a maximum of M characters each. No word exceeds the line length, so that $l_i \leq M$ for $i = 1, 2, \ldots, n$. The criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and exactly one space appears between words, then the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. The goal is to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly. Analyze the running time and space requirements of your algorithm.

14-5 Edit distance

In order to transform a source string of text $x[1:m]$ to a target string $y[1:n]$, you can perform various transformation operations. The goal is, given x and y, to produce a series of transformations that changes x to y. An array z—assumed to be large enough to hold all the characters it needs—holds the intermediate results. Initially, z is empty, and at termination, you should have $z[j] = y[j]$ for $j = 1, 2, \ldots, n$. The procedure for solving this problem maintains current indices i into x and j into z, and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. Every character in x must be examined during the transformation, which means that at the end of the sequence of transformation operations, $i = m + 1$.

You may choose from among six transformation operations, each of which has a constant cost that depends on the operation:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$ and has cost Q_C.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$ and has cost Q_R.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$ and has cost Q_D.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x and has cost Q_I.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order: setting $z[j] = x[i+1]$ and $z[j+1] = x[i]$, and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i+1]$ and has cost Q_T.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation. It has cost Q_K.

Figure 14.12 gives one way to transform the source string `algorithm` to the target string `altruistic`. Several other sequences of transformation operations can transform `algorithm` to `altruistic`.

Assume that $Q_C < Q_D + Q_I$ and $Q_R < Q_D + Q_I$, since otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming `algorithm` to `altruistic` is $3Q_C + Q_R + Q_D + 4Q_I + Q_T + Q_K$.

a. Given two sequences $x[1:m]$ and $y[1:n]$ and the costs of the transformation operations, the ***edit distance*** from x to y is the cost of the least expensive op-

Operation	x	z
initial strings	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

Figure 14.12 A sequence of operations that transforms the source algorithm to the target string altruistic. The underlined characters are $x[i]$ and $z[j]$ after the operation.

eration sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1:m]$ to $y[1:n]$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [405, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x =$ GATCGGCAT and $y =$ CAATGTGAATC, one alignment is

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

A + under a position indicates a score of $+1$ for that position, a – indicates a score of -1, and a $*$ indicates a score of -2, so that this alignment has a total score of $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explain how to cast the problem of finding an optimal alignment as an edit-distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

14-6 Planning a company party

Professor Blutarsky is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure, that is, the supervisor relation forms a tree rooted at the president. The human resources department has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Blutarsky is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.3. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

14-7 Viterbi algorithm

Dynamic programming on a directed graph can play a part in speech recognition. A directed graph $G = (V, E)$ with labeled edges forms a formal model of a person speaking a restricted language. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. Each directed path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model, with the label of a path being the concatenation of the labels of the edges on that path.

a. Describe an efficient algorithm that, given an edge-labeled directed graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (*Hint:* You may find concepts from Chapter 20 useful.)

Now suppose that every edge $(u, v) \in E$ has an associated nonnegative probability $p(u, v)$ of being traversed, so that the corresponding sound is produced. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. Think of the probability of a path beginning at vertex v_0 as the probability that a "random walk" beginning at v_0 follows the specified path, where the edge leaving a vertex u is taken randomly, according to the probabilities of the available edges leaving u.

b. Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at vertex v_0 and having label s. Analyze the running time of your algorithm.

14-8 Image compression by seam carving

Suppose that you are given a color picture consisting of an $m \times n$ array $A[1:m, 1:n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. You want to compress this picture slightly, by removing one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid incongruous visual effects, however, the pixels removed in two adjacent rows must lie in either the same column or adjacent columns. In this way, the pixels removed form a "seam" from the top row to the bottom row, where successive pixels in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b. Suppose now that along with each pixel $A[i, j]$, you are given a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Define the disruption measure of a seam as the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

14-9 Breaking a string

A certain string-processing programming language allows you to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose that you want to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that you want to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If you program the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If you program the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, you could break first at 8 (costing 20), then break the left piece at 2 (costing another 8), and finally the right piece at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given an array $L[1:m]$ containing the break points for a string of n characters, compute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

14-10 Planning an investment strategy

Your knowledge of algorithms helps you obtain an exciting job with a hot startup, along with a \$10,000 signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use your investment manager, G. I. Luvcache, to manage your signing bonus. The company that Luvcache works with requires you to observe the following rules. It offers n different investments, numbered 1 through n. In each year j, investment i provides a return rate of r_{ij}. In other words, if you invest d dollars in investment i in year j, then at the end of year j, you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investment. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$. You pay the fee once per year at the end of the year, and it is the same amount, f_2, whether you move money in and out of only one investment, or in and out of many investments.

a. The problem, as stated, allows you to invest your money in multiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)

b. Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.

c. Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?

d. Suppose that Luvcache's company imposes the additional restriction that, at any point, you can have no more than \$15,000 in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

14-11 Inventory planning

The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i, the company knows the demand d_i, that is, the number of machines that it will sell. Let $D = \sum_{i=1}^{n} d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if the company is holding any unsold machines at the end of a month, it must pay inventory costs. The company can hold up to D machines, with the cost for holding j machines given as a function $h(j)$ for $j = 1, 2, \ldots, D$ that monotonically increases with j.

Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polynomial in n and D.

14-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of \$$X$ to spend on free agents. You are allowed to spend less than \$$X$, but the owner will fire you if you spend any more than \$$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available.[10] Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

[10] Although there are nine positions on a baseball team, N is not necessarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate "positions," as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

To determine how valuable a player is going to be, you decide to use a sabermetric statistic[11] known as "WAR," or "wins above replacement." A player with a higher WAR is more valuable than a player with a lower WAR. It is not necessarily more expensive to sign a player with a higher WAR than a player with a lower WAR, because factors other than a player's value determine how much it costs to sign them.

For each available free-agent player p, you have three pieces of information:

- the player's position,
- $p.cost$, the amount of money it costs to sign the player, and
- $p.war$, the player's WAR.

Devise an algorithm that maximizes the total WAR of the players you sign while spending no more than \$$X$. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total WAR of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

Chapter notes

Bellman [44] began the systematic study of dynamic programming in 1955, publishing a book about it in 1957. The word "programming," both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis.

Galil and Park [172] classify dynamic-programming algorithms according to the size of the table and the number of other table entries each entry depends on. They call a dynamic-programming algorithm tD/eD if its table size is $O(n^t)$ and each entry depends on $O(n^e)$ other entries. For example, the matrix-chain multiplication algorithm in Section 14.2 is $2D/1D$, and the longest-common-subsequence algorithm in Section 14.4 is $2D/0D$.

The MATRIX-CHAIN-ORDER algorithm on page 378 is by Muraoka and Kuck [339]. Hu and Shing [230, 231] give an $O(n \lg n)$-time algorithm for the matrix-chain multiplication problem.

The $O(mn)$-time algorithm for the longest-common-subsequence problem appears to be a folk algorithm. Knuth [95] posed the question of whether subquadratic

[11] ***Sabermetrics*** is the application of statistical analysis to baseball records. It provides several ways to compare the relative values of individual players.

algorithms for the LCS problem exist. Masek and Paterson [316] answered this question in the affirmative by giving an algorithm that runs in $O(mn/\lg n)$ time, where $n \le m$ and the sequences are drawn from a set of bounded size. For the special case in which no element appears more than once in an input sequence, Szymanski [425] shows how to solve the problem in $O((n+m)\lg(n+m))$ time. Many of these results extend to the problem of computing string edit distances (Problem 14-5).

An early paper on variable-length binary encodings by Gilbert and Moore [181], which had applications to constructing optimal binary search trees for the case in which all probabilities p_i are 0, contains an $O(n^3)$-time algorithm. Aho, Hopcroft, and Ullman [5] present the algorithm from Section 14.5. Splay trees [418], which modify the tree in response to the search queries, come within a constant factor of the optimal bounds without being initialized with the frequencies. Exercise 14.5-4 is due to Knuth [264]. Hu and Tucker [232] devised an algorithm for the case in which all probabilities p_i are 0 that uses $O(n^2)$ time and $O(n)$ space. Subsequently, Knuth [261] reduced the time to $O(n \lg n)$.

Problem 14-8 is due to Avidan and Shamir [30], who have posted on the web a wonderful video illustrating this image-compression technique.

15 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill, and simpler, more efficient algorithms will do. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice leads to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 14, particularly Section 14.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We first examine, in Section 15.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We'll arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that an optimal solution can result from always making greedy choices. Section 15.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 15.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. Finally, Section 15.4 shows that in order to decide which blocks to replace when a miss occurs in a cache, the "furthest-in-future" strategy is optimal if the sequence of block accesses is known in advance.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that you can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 21), Dijkstra's algorithm for shortest paths from a single source (Section 22.3), and a greedy set-covering heuristic (Section 35.3). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read this chapter and Chapter 21 independently of each other, you might find it useful to read them together.

15.1 An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Imagine that you are in charge of scheduling a conference room. You are presented with a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to reserve the conference room, and the room can serve only one activity at a time. Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \le s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$. (Assume that if your staff needs time to change over the room from one activity to the next, the changeover time is built into the intervals.) In the ***activity-selection problem***, your goal is to select a maximum-size subset of mutually compatible activities. Assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n . \tag{15.1}$$

(We'll see later the advantage that this assumption provides.) For example, consider the set of activities in Figure 15.1. The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities, and another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We'll see how to solve this problem, proceeding in several steps. First we'll explore a dynamic-programming solution, in which you consider several choices when determining which subproblems to use in an optimal solution. We'll then observe that you need to consider only one choice—the greedy choice—and that when you make the greedy choice, only one subproblem remains. Based on these observations, we'll develop a recursive greedy algorithm to solve the activity-selection problem. Finally, we'll complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we go through in this section are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 15.1 A set $\{a_1, a_2, \ldots, a_{11}\}$ of activities. Activity a_i has start time s_i and finish time f_i.

The optimal substructure of the activity-selection problem

Let's verify that the activity-selection problem exhibits optimal substructure. Denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that you want to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k. By including a_k in an optimal solution, you are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts). Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that an optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If you could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then you could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. You would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

This way of characterizing optimal substructure suggests that you can solve the activity-selection problem by dynamic programming. Let's denote the size of an optimal solution for the set S_{ij} by $c[i, j]$. Then, the dynamic-programming approach gives the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

Of course, if you do not know that an optimal solution for the set S_{ij} includes activity a_k, you must examine all activities in S_{ij} to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \tag{15.2}$$

You can then develop a recursive algorithm and memoize it, or you can work bottom-up and fill in table entries as you go along. But you would be overlooking another important characteristic of the activity-selection problem that you can use to great advantage.

Making the greedy choice

What if you could choose an activity to add to an optimal solution without having to first solve all the subproblems? That could save you from having to consider all the choices inherent in recurrence (15.2). In fact, for the activity-selection problem, you need to consider only one choice: the greedy choice.

What is the greedy choice for the activity-selection problem? Intuition suggests that you should choose an activity that leaves the resource available for as many other activities as possible. Of the activities you end up choosing, one of them must be the first one to finish. Intuition says, therefore, choose the activity in S with the earliest finish time, since that leaves the resource available for as many of the activities that follow it as possible. (If more than one activity in S has the earliest finish time, then choose any such activity.) In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1. Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem. Exercise 15.1-3 asks you to explore other possibilities.

Once you make the greedy choice, you have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't you have to consider activities that finish before a_1 starts? Because $s_1 < f_1$, and because f_1 is the earliest finish time of any activity, no activity can have a finish time less than or equal to s_1. Thus, all activities that are compatible with activity a_1 must start after a_1 finishes.

Furthermore, we have already established that the activity-selection problem exhibits optimal substructure. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If you make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.[1] Optimal substructure says that if a_1 belongs to an optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1.

One big question remains: Is this intuition correct? Is the greedy choice—in which you choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

[1] We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. The context will make it clear whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 15.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m belongs to some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are compatible, which follows because the activities in A_k are compatible, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

Although you might be able to solve the activity-selection problem with dynamic programming, Theorem 15.1 says that you don't need to. Instead, you can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because you always choose the activity with the earliest finish time, the finish times of the activities that you choose must strictly increase. You can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm. Instead, it can work top-down, choosing an activity to put into the optimal solution that it constructs and then solving the subproblem of choosing activities from those that are compatible with those already chosen. Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that you know you can bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, let's see a straightforward, recursive procedure to solve the activity-selection problem. The procedure RECURSIVE-ACTIVITY-SELECTOR on the following page takes the start and finish times of the activities, represented as arrays s and f,[2] the index k that defines the subproblem S_k it is to solve, and the size n of the original problem. It returns a maximum-

[2] Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than with subscripts.

size set of mutually compatible activities in S_k. The procedure assumes that the n input activities are already ordered by monotonically increasing finish time, according to equation (15.1). If not, you can first sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S. The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  $m = k + 1$
2  while $m \leq n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
3      $m = m + 1$
4  if $m \leq n$
5      return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
6  else return $\emptyset$
```

Figure 15.2 shows how the algorithm operates on the activities in Figure 15.1. In a given recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), the **while** loop of lines 2–3 looks for the first activity in S_k to finish. The loop examines $a_{k+1}, a_{k+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_k, which means that $s_m \geq f_k$. If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). Alternatively, the loop may terminate because $m > n$, in which case the procedure has examined all activities in S_k without finding one that is compatible with a_k. In this case, $S_k = \emptyset$, and so line 6 returns $\emptyset$.

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ is $\Theta(n)$. To see why, observe that over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

The recursive procedure can be converted to an iterative one because the procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive" (see Problem 7-5): it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form. In fact, some compilers for certain programming languages perform this task automatically.

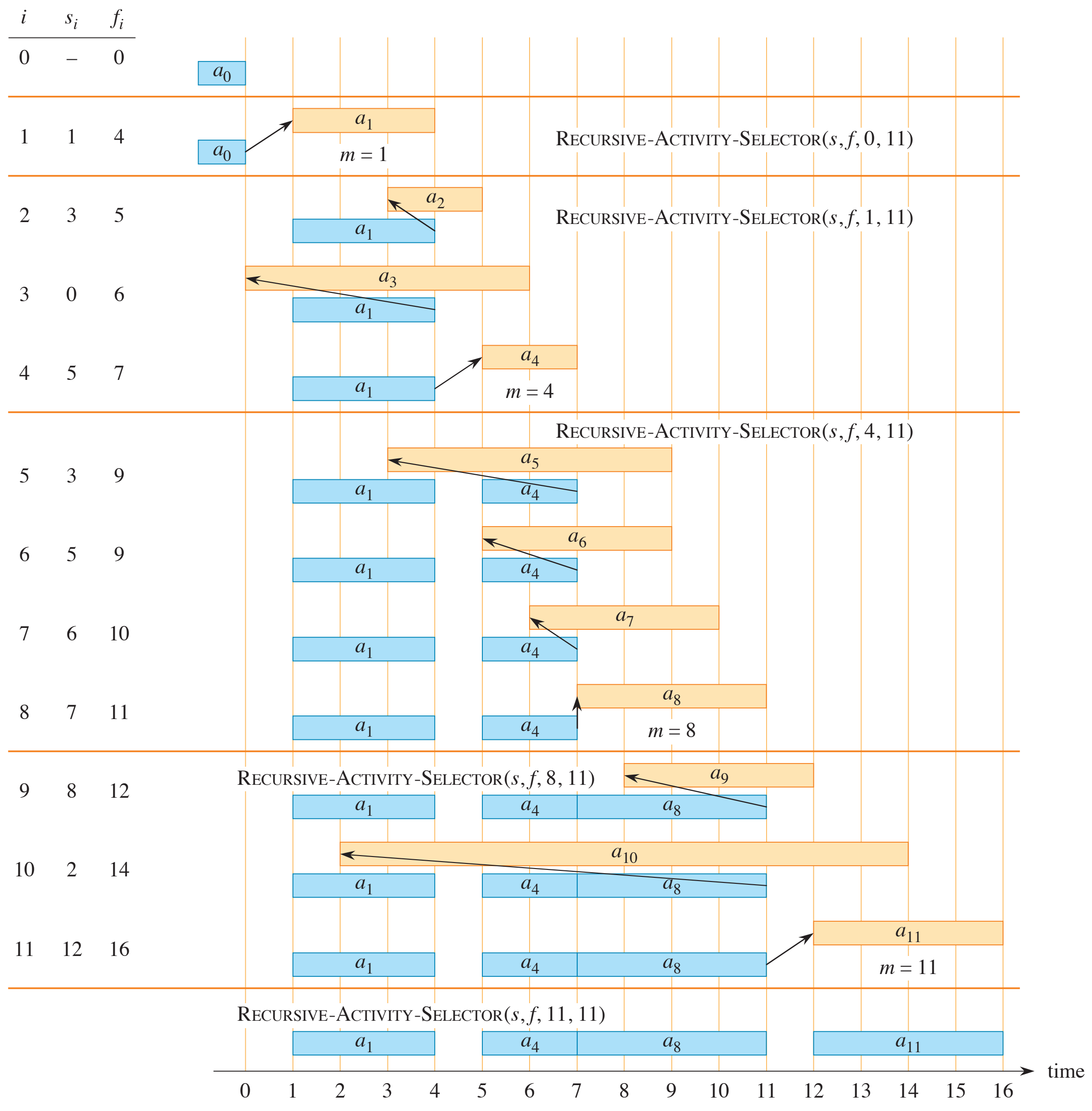


Figure 15.2 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities from Figure 15.1. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, 11)$, selects activity a_1. In each recursive call, the activities that have already been selected are blue, and the activity shown in tan is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR$(s, f, 11, 11)$, returns $\emptyset$. The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It, too, assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  return A
```

The procedure works as follows. The variable k indexes the most recent addition to A, corresponding to the activity a_k in the recursive version. Since the procedure considers the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A. That is,

$$f_k = \max\{f_i : a_i \in A\} . \tag{15.3}$$

Lines 1–2 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 3–6 finds the earliest activity in S_k to finish. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities. Such an activity is the earliest in S_k to finish. To see whether activity a_m is compatible with every activity currently in A, it suffices by equation (15.3) to check (in line 4) that its start time s_m is not earlier than the finish time f_k of the activity most recently added to A. If activity a_m is compatible, then lines 5–6 add activity a_m to A and set k to m. The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Exercises

15.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (15.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (15.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

15.1-2
Suppose that instead of always selecting the first activity to finish, you instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

15.1-3
Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

15.1-4
You are given a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. You wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the ***interval-graph coloring problem***. It is modeled by an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

15.1-5
Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, the goal is to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

15.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes the choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 15.1 to develop a greedy algorithm was a bit more involved than is typical. It consisted of the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (15.2), but bypassed developing a recursive algorithm based solely on this recurrence.)
3. Show that if you make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

These steps highlighted in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, the first cut at the activity-selection problem defined the subproblems S_{ij}, where both i and j varied. We then found that if you always make the greedy choice, you can restrict the subproblems to be of the form S_k.

An alternative approach is to fashion optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, start by dropping the second subscript and defining subproblems of the form S_k. Then prove that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k. More generally, you can design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which you make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if you combine an

optimal solution to the subproblem with the greedy choice you have made, you arrive at an optimal solution to the original problem.

Later sections of this chapter will use this more direct process. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can you tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients. If you can demonstrate that the problem has these properties, then you are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: you can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when you are considering which choice to make, you make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, you make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, you typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, you can solve them top down, but memoizing. Of course, even though the code works top down, you still must solve the subproblems before making a choice.) In a greedy algorithm, you make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses top down, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, you need to prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 15.1, the proof examines a globally optimal solution to some subproblem. It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

You can usually make the greedy choice more efficiently than when you have to consider a wider set of choices. For example, in the activity-selection problem, assuming that the activities were already sorted in monotonically increasing order by finish times, each activity needed to be examined just once. By preprocessing the input or by using an appropriate data structure (often a priority queue), you often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

As we saw in Chapter 14, a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing whether dynamic programming applies, and it's also essential for greedy algorithms. As an example of optimal substructure, recall how Section 15.1 demonstrated that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}. Given this optimal substructure, we argued that if you know which activity to use as a_k, you can construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj}. This observation of optimal substructure gave rise to the recurrence (15.2) that describes the value of an optimal solution.

You will usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, you have the luxury of assuming that you arrived at a subproblem by having made the greedy choice in the original problem. All you really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtle differences between the two techniques, let's investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is the following. A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. The thief cannot take a fractional amount of an item or take an item more than once.)

In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, if the most valuable load weighing at most W pounds includes item j, then the remaining load must be the most valuable load weighing at most $W - w_j$ pounds that the thief can take from the $n - 1$ original items excluding item j. For the comparable fractional problem, if if the most valuable load weighing at most W pounds includes weight w of item j, then the remaining load must be the most valuable load weighing at most $W - w$ pounds that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, a greedy strategy works to solve the fractional knapsack problem, but not the 0-1 problem. To solve the fractional problem, first compute the value per pound v_i / w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and the thief can still carry more, then the thief takes as much as possible of the item with the next greatest value per pound, and so forth, until reaching the weight limit W. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. You are asked to prove that the fractional knapsack problem has the greedy-choice property in Exercise 15.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 15.3(a). This example has three items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth \$60. Item 2 weighs 20 pounds and is worth \$100. Item 3 weighs 30 pounds and is worth \$120. Thus, the value per pound of item 1 is \$6 per pound, which is greater than the value per pound of either item 2 (\$5 per pound) or item 3 (\$4 per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 15.3(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 15.3(c). Taking item 1 doesn't work in the 0-1 problem, because the thief is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load. In the 0-1 problem, when you consider whether to include an item in the knapsack, you must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before you can make the choice. The problem formulated in this way gives rise to many overlapping sub-

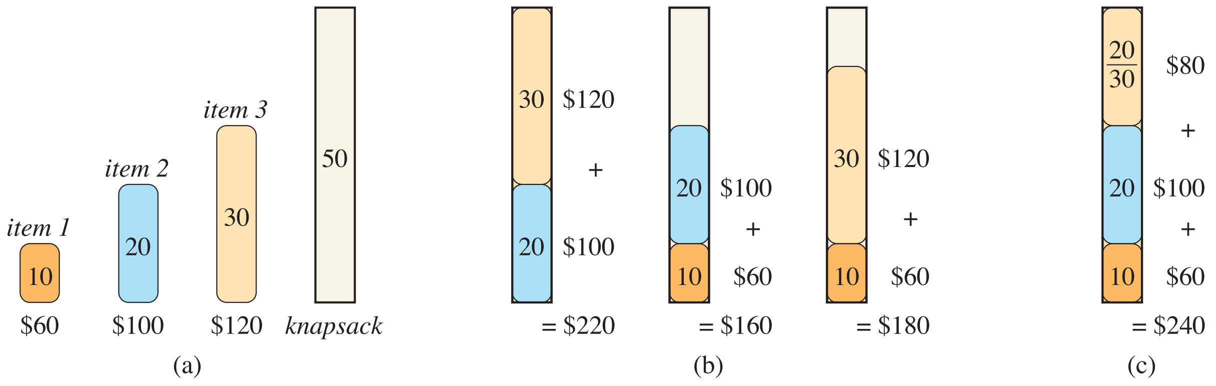


Figure 15.3 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

problems—a hallmark of dynamic programming, and indeed, as Exercise 15.2-2 asks you to show, you can use dynamic programming to solve the 0-1 problem.

Exercises

15.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

15.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in the knapsack.

15.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

15.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. The professor plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water and can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. The professor has an official North Dakota state map, which shows all the places along U.S. 2 to refill water and the distances between these locations.

The professor's goal is to minimize the number of water stops along the route across the state. Give an efficient method by which the professor can determine which water stops to make. Prove that your strategy yields an optimal solution, and give its running time.

15.2-5
Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

★ ***15.2-6***
Show how to solve the fractional knapsack problem in $O(n)$ time.

15.2-7
You are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} {a_i}^{b_i}$. Give an algorithm that maximizes your payoff. Prove that your algorithm maximizes the payoff, and state its running time, omitting the time for reordering the sets.

15.3 Huffman codes

Huffman codes compress data well: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. The data arrive as a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.

Suppose that you have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the frequencies given by Figure 15.4. The character `a` occurs 45,000 times, the character `b` occurs 13,000 times, and so on.

You have many options for how to represent such a file of information. Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short)

	`a`	`b`	`c`	`d`	`e`	`f`
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 15.4 A character-coding problem. A data file of 100,000 characters contains only the characters `a`–`f`, with the frequencies indicated. With each character represented by a 3-bit codeword, encoding the file requires 300,000 bits. With the variable-length code shown, the encoding requires only 224,000 bits.

in which each character is represented by a unique binary string, which we call a ***codeword***. If you use a ***fixed-length code***, you need $\lceil \lg n \rceil$ bits to represent $n \geq 2$ characters. For 6 characters, therefore, you need 3 bits: `a` = 000, `b` = 001, `c` = 010, `d` = 011, `e` = 100, and `f` = 101. This method requires 300,000 bits to encode the entire file. Can you do better?

A ***variable-length code*** can do considerably better than a fixed-length code. The idea is simple: give frequent characters short codewords and infrequent characters long codewords. Figure 15.4 shows such a code. Here, the 1-bit string 0 represents `a`, and the 4-bit string 1100 represents `f`. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

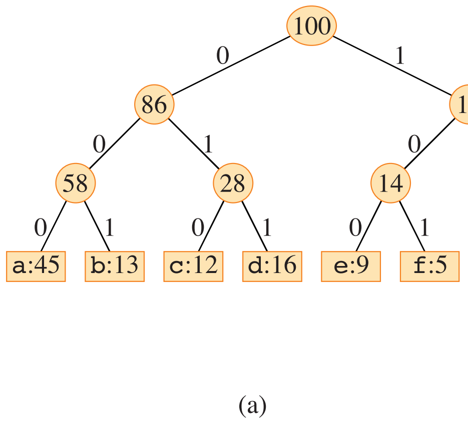
to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix-free codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix-free codes***. Although we won't prove it here, a prefix-free code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix-free codes.

Encoding is always simple for any binary character code: just concatenate the codewords representing each character of the file. For example, with the variable-length prefix-free code of Figure 15.4, the 4-character file `face` has the encoding $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where "·" denotes concatenation.

Prefix-free codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. You can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 100011001101 parses uniquely as $100 \cdot 0 \cdot 1100 \cdot 1101$, which decodes to `cafe`.



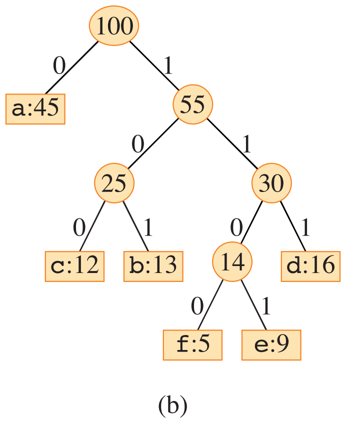


Figure 15.5 Trees corresponding to the coding schemes in Figure 15.4. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. All frequencies are in thousands. **(a)** The tree corresponding to the fixed-length code `a` = 000, `b` = 001, `c` = 010, `d` = 011, `e` = 100, `f` = 101. **(b)** The tree corresponding to the optimal prefix-free code `a` = 0, `b` = 101, `c` = 100, `d` = 111, `e` = 1101, `f` = 1100.

The decoding process needs a convenient representation for the prefix-free code so that you can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. Interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 15.5 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 15.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 15.5(a), is not a full binary tree: it contains codewords beginning with 10, but none beginning with 11. Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix-free code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3 on page 1175).

Given a tree T corresponding to a prefix-free code, we can compute the number of bits required to encode a file. For each character c in the alphabet C, let the attribute $c.\mathit{freq}$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) , \tag{15.4}$$

which we define as the ***cost*** of the tree T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix-free code, called a ***Huffman code*** in his honor. In line with our observations in Section 15.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

The procedure HUFFMAN assumes that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.\mathit{freq}$ giving its frequency. The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. The result of merging two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN($C$)
1  $n = |C|$
2  $Q = C$
3  for $i = 1$ to $n - 1$
4      allocate a new node $z$
5      $x$ = EXTRACT-MIN($Q$)
6      $y$ = EXTRACT-MIN($Q$)
7      $z.left = x$
8      $z.right = y$
9      $z.freq = x.freq + y.freq$
10     INSERT($Q, z$)
11 return EXTRACT-MIN($Q$)     // the root of the tree is the only node left
```

For our example, Huffman's algorithm proceeds as shown in Figure 15.6. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix-free code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

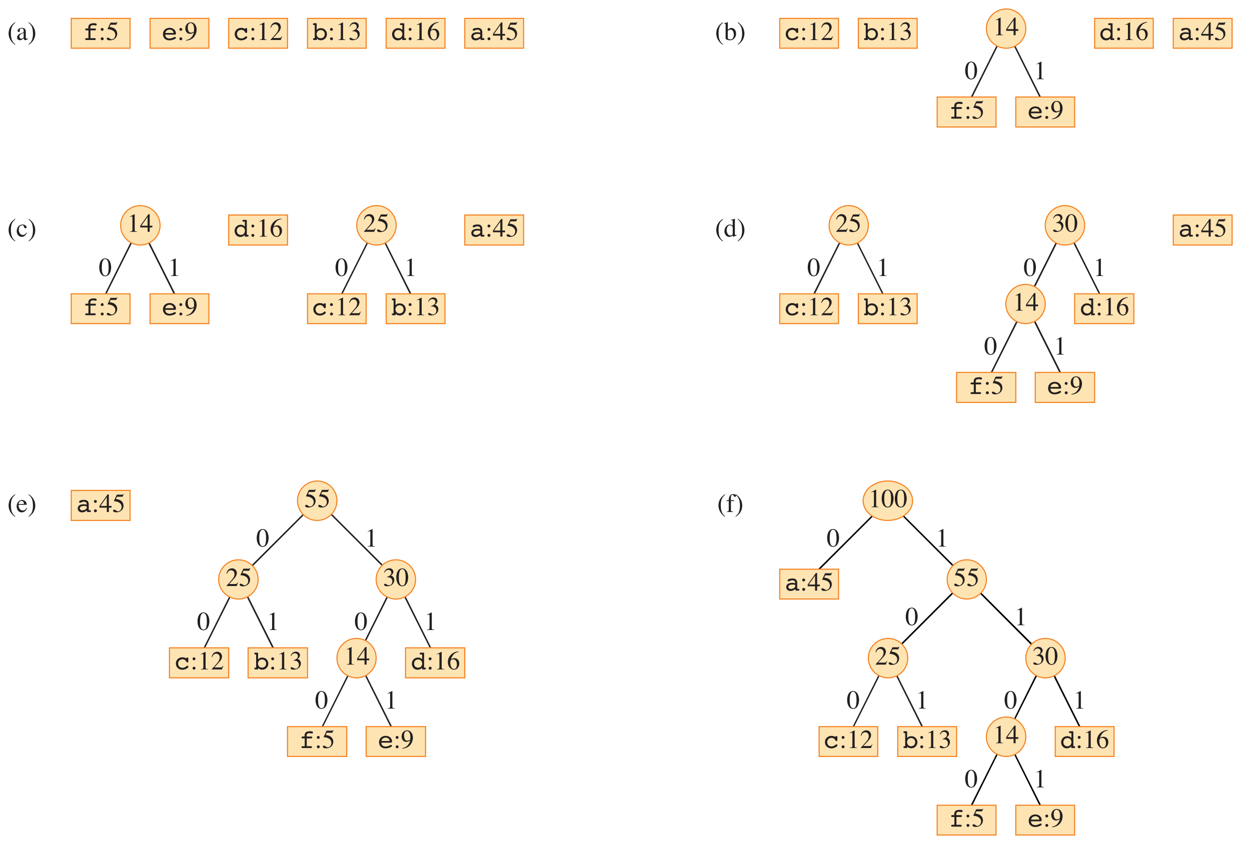


Figure 15.6 The steps of Huffman's algorithm for the frequencies given in Figure 15.4. Each part shows the contents of the queue sorted into increasing order by frequency. Each step merges the two trees with the lowest frequencies. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

The HUFFMAN procedure works as follows. Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3–10 repeatedly extracts the two nodes x and y of lowest frequency from the queue and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 9. The node z has x as its left child and y as its right child. (This order is arbitrary. Switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 11 returns the one node left in the queue, which is the root of the code tree.

The algorithm produces the same result without the variables x and y, assigning the values returned by the EXTRACT-MIN calls directly to $z.left$ and $z.right$ in lines 7 and 8, and changing line 9 to $z.freq = z.left.freq + z.right.freq$. We'll use the node names x and y in the proof of correctness, however, so we leave them in.

The running time of Huffman's algorithm depends on how the min-priority queue Q is implemented. Let's assume that it's implemented as a binary min-heap (see Chapter 6). For a set C of n characters, the BUILD-MIN-HEAP procedure discussed in Section 6.3 can initialize Q in line 2 in $O(n)$ time. The **for** loop in lines 3–10 executes exactly $n - 1$ times, and since each heap operation runs in $O(\lg n)$ time, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we'll show that the problem of determining an optimal prefix-free code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 15.2 (Optimal prefix-free codes have the greedy-choice property)
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Let a and b be any two characters that are sibling leaves of maximum depth in T. Without loss of generality, assume that $a.freq \le b.freq$ and $x.freq \le y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \le a.freq$ and $y.freq \le b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$, but $x.freq = b.freq$ implies that $a.freq = b.freq = x.freq = y.freq$ (see Exercise 15.3-1), and the lemma would be trivially true. Therefore, assume that $x.freq \ne b.freq$, which means that $x \ne b$.

As Figure 15.7 shows, imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a

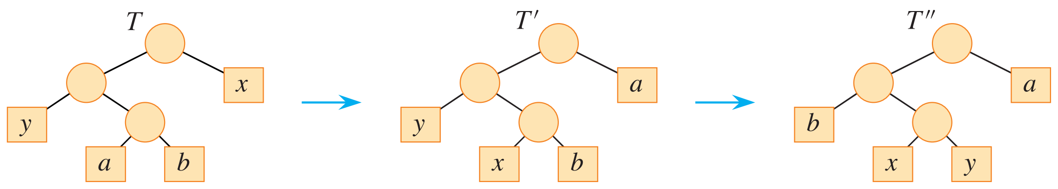


Figure 15.7 An illustration of the key step in the proof of Lemma 15.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies. They appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

tree T'' in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (15.4), the difference in cost between T and T' is

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&\quad = x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&\quad = x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&\quad = (a.\mathit{freq} - x.\mathit{freq})(d_T(a) - d_T(x)) \\
&\quad \geq 0 ,
\end{aligned}$$

because both $a.\mathit{freq} - x.\mathit{freq}$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.\mathit{freq} - x.\mathit{freq}$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ∎

Lemma 15.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 15.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix-free codes has the optimal-substructure property.

Lemma 15.3 (Optimal prefix-free codes have the optimal-substructure property)
Let C be a given alphabet with frequency $c.\mathit{freq}$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = (C - \{x, y\}) \cup \{z\}$. Define *freq* for all characters in C' with the same values as in C, along with $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Let T' be any tree representing an optimal prefix-free code for alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix-free code for the alphabet C.

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T', by considering the component costs in equation (15.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.\mathit{freq} \cdot d_T(c) = c.\mathit{freq} \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.\mathit{freq} \cdot d_T(x) + y.\mathit{freq} \cdot d_T(y) &= (x.\mathit{freq} + y.\mathit{freq})(d_{T'}(z) + 1) \\ &= z.\mathit{freq} \cdot d_{T'}(z) + (x.\mathit{freq} + y.\mathit{freq}) , \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.\mathit{freq} + y.\mathit{freq}$$

or, equivalently,

$$B(T') = B(T) - x.\mathit{freq} - y.\mathit{freq} .$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix-free code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 15.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.\mathit{freq} - y.\mathit{freq} \\ &< B(T) - x.\mathit{freq} - y.\mathit{freq} \\ &= B(T') , \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix-free code for C'. Thus, T must represent an optimal prefix-free code for the alphabet C. ■

Theorem 15.4
Procedure HUFFMAN produces an optimal prefix-free code.

Proof Immediate from Lemmas 15.2 and 15.3. ■

Exercises

15.3-1
Explain why, in the proof of Lemma 15.2, if $x.\mathit{freq} = b.\mathit{freq}$, then we must have $a.\mathit{freq} = b.\mathit{freq} = x.\mathit{freq} = y.\mathit{freq}$.

15.3-2
Prove that a non-full binary tree cannot correspond to an optimal prefix-free code.

15.3-3
What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

`a`:1 `b`:1 `c`:2 `d`:3 `e`:5 `f`:8 `g`:13 `h`:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

15.3-4
Prove that the total cost $B(T)$ of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

15.3-5
Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

15.3-6
Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

15.3-7
A data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

15.3-8
Show that no lossless (invertible) compression scheme can guarantee that for every input file, the corresponding output file is shorter. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

15.4 Offline caching

Computer systems can decrease the time to access data by storing a subset of the main memory in the ***cache***: a small but faster memory. A cache organizes data into ***cache blocks*** typically comprising 32, 64, or 128 bytes. You can also think of main memory as a cache for disk-resident data in a virtual-memory system. Here, the blocks are called ***pages***, and 4096 bytes is a typical size.

As a computer program executes, it makes a sequence of memory requests. Say that there are n memory requests, to data in blocks $b_1, b_2, \ldots, b_n$, in that order. The blocks in the access sequence might not be distinct, and indeed, any given block is usually accessed multiple times. For example, a program that accesses four distinct blocks p, q, r, s might make a sequence of requests to blocks $s, q, s, q, q, s, p, p, r, s, s, q, p, r, q$. The cache can hold up to some fixed number k of cache blocks. It starts out empty before the first request. Each request causes at most one block to enter the cache and at most one block to be evicted from the cache. Upon a request for block b_i, any one of three scenarios may occur:

1. Block b_i is already in the cache, due to a previous request for the same block. The cache remains unchanged. This situation is known as a ***cache hit***.
2. Block b_i is not in the cache at that time, but the cache contains fewer than k blocks. In this case, block b_i is placed into the cache, so that the cache contains one more block than it did before the request.
3. Block b_i is not in the cache at that time and the cache is full: it contains k blocks. Block b_i is placed into the cache, but before that happens, some other block in the cache must be evicted from the cache in order to make room.

The latter two situations, in which the requested block is not already in the cache, are called ***cache misses***. The goal is to minimize the number of cache misses or, equivalently, to maximize the number of cache hits, over the entire sequence of n requests. A cache miss that occurs while the cache holds fewer than k blocks—that is, as the cache is first being filled up—is known as a ***compulsory miss***, since no prior decision could have kept the requested block in the cache. When a cache miss occurs and the cache is full, ideally the choice of which block to evict should allow for the smallest possible number of cache misses over the entire sequence of future requests.

Typically, caching is an online problem. That is, the computer has to decide which blocks to keep in the cache without knowing the future requests. Here, however, let's consider the offline version of this problem, in which the computer knows in advance the entire sequence of n requests and the cache size k, with a goal of minimizing the total number of cache misses.

To solve this offline problem, you can use a greedy strategy called ***furthest-in-future***, which chooses to evict the block in the cache whose next access in the request sequence comes furthest in the future. Intuitively, this strategy makes sense: if you're not going to need something for a while, why keep it around? We'll show that the furthest-in-future strategy is indeed optimal by showing that the offline caching problem exhibits optimal substructure and that furthest-in-future has the greedy-choice property.

Now, you might be thinking that since the computer usually doesn't know the sequence of requests in advance, there is no point in studying the offline problem. Actually, there is. In some situations, you do know the sequence of requests in advance. For example, if you view the main memory as the cache and the full set of data as residing on disk (or a solid-state drive), there are algorithms that plan out the entire set of reads and writes in advance. Furthermore, we can use the number of cache misses produced by an optimal algorithm as a baseline for comparing how well online algorithms perform. We'll do just that in Section 27.3.

Offline caching can even model real-world problems. For example, consider a scenario where you know in advance a fixed schedule of n events at known locations. Events may occur at a location multiple times, not necessarily consecutively. You are managing a group of k agents, you need to ensure that you have one agent at each location when an event occurs, and you want to minimize the number of times that agents have to move. Here, the agents are like the blocks, the events are like the requests, and moving an agent is akin to a cache miss.

Optimal substructure of offline caching

To show that the offline problem exhibits optimal substructure, let's define the subproblem (C, i) as processing requests for blocks $b_i, b_{i+1}, \ldots, b_n$ with cache configuration C at the time that the request for block b_i occurs, that is, C is a subset of the set of blocks such that $|C| \leq k$. A solution to subproblem (C, i) is a sequence of decisions that specifies which block to evict (if any) upon each request for blocks $b_i, b_{i+1}, \ldots, b_n$. An optimal solution to subproblem (C, i) minimizes the number of cache misses.

Consider an optimal solution S to subproblem (C, i), and let C' be the contents of the cache after processing the request for block b_i in solution S. Let S' be the subsolution of S for the resulting subproblem $(C', i + 1)$. If the request for b_i results in a cache hit, then the cache remains unchanged, so that $C' = C$. If the request for block b_i results in a cache miss, then the contents of the cache change, so that $C' \neq C$. We claim that in either case, S' is an optimal solution to subproblem $(C', i + 1)$. Why? If S' is not an optimal solution to subproblem $(C', i + 1)$, then there exists another solution S'' to subproblem $(C', i + 1)$ that makes fewer cache misses than S'. Combining S'' with the decision of S at the request for block b_i yields another solution that makes fewer cache misses than S, which contradicts the assumption that S is an optimal solution to subproblem (C, i).

To quantify a recursive solution, we need a little more notation. Let $R_{C,i}$ be the set of all cache configurations that can immediately follow configuration C after processing a request for block b_i. If the request results in a cache hit, then the cache remains unchanged, so that $R_{C,i} = \{C\}$. If the request for b_i results in a cache miss, then there are two possibilities. If the cache is not full ($|C| < k$), then the cache is filling up and the only choice is to insert b_i into the cache, so that $R_{C,i} = \{C \cup \{b_i\}\}$. If the cache is full ($|C| = k$) upon a cache miss, then $R_{C,i}$ contains k potential configurations: one for each candidate block in C that could be evicted and replaced by block b_i. In this case, $R_{C,i} = \{(C - \{x\}) \cup \{b_i\} : x \in C\}$. For example, if $C = \{p, q, r\}$, $k = 3$, and block s is requested, then $R_{C,i} = \{\{p, q, s\}, \{p, r, s\}, \{q, r, s\}\}$.

Let $miss(C, i)$ denote the minimum number of cache misses in a solution for subproblem (C, i). Here is a recurrence for $miss(C, i)$:

$$miss(C, i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C \,, \\ 1 & \text{if } i = n \text{ and } b_n \notin C \,, \\ miss(C, i + 1) & \text{if } i < n \text{ and } b_i \in C \,, \\ 1 + \min\{miss(C', i + 1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C \,. \end{cases}$$

Greedy-choice property

To prove that the furthest-in-future strategy yields an optimal solution, we need to show that optimal offline caching exhibits the greedy-choice property. Combined with the optimal-substructure property, the greedy-choice property will prove that furthest-in-future produces the minimum possible number of cache misses.

Theorem 15.5 (Optimal offline caching has the greedy-choice property)
Consider a subproblem (C, i) when the cache C contains k blocks, so that it is full, and a cache miss occurs. When block b_i is requested, let $z = b_m$ be the block in C whose next access is furthest in the future. (If some block in the cache will never again be referenced, then consider any such block to be block z, and add a dummy request for block $z = b_m = b_{n+1}$.) Then evicting block z upon a request for block b_i is included in some optimal solution for the subproblem (C, i).

Proof Let S be an optimal solution to subproblem (C, i). If S evicts block z upon the request for block b_i, then we are done, since we have shown that some optimal solution includes evicting z.

So now suppose that optimal solution S evicts some other block x when block b_i is requested. We'll construct another solution S' to subproblem (C, i) which, upon the request for b_i, evicts block z instead of x and induces no more cache misses than S does, so that S' is also optimal. Because different solutions may yield different cache configurations, denote by $C_{S,j}$ the configuration of the cache under solution S just before the request for some block b_j, and likewise for solution S' and $C_{S',j}$. We'll show how to construct S' with the following properties:

1. For $j = i + 1, \ldots, m$, let $D_j = C_{S,j} \cap C_{S',j}$. Then, $|D_j| \geq k - 1$, so that the cache configurations $C_{S,j}$ and $C_{S',j}$ differ by at most one block. If they differ, then $C_{S,j} = D_j \cup \{z\}$ and $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$.
2. For each request of blocks $b_i, \ldots, b_{m-1}$, if solution S has a cache hit, then solution S' also has a cache hit.
3. For all $j > m$, the cache configurations $C_{S,j}$ and $C_{S',j}$ are identical.
4. Over the sequence of requests for blocks $b_i, \ldots, b_m$, the number of cache misses produced by solution S' is at most the number of cache misses produced by solution S.

We'll prove inductively that these properties hold for each request.

1. We proceed by induction on j, for $j = i + 1, \ldots, m$. For the base case, the initial caches $C_{S,i}$ and $C_{S',i}$ are identical. Upon the request for block b_i, solution S evicts x and solution S' evicts z. Thus, cache configurations $C_{S,i+1}$ and $C_{S',i+1}$ differ by just one block, $C_{S,i+1} = D_{i+1} \cup \{z\}$, $C_{S',i+1} = D_{i+1} \cup \{x\}$, and $x \neq z$.

   The inductive step defines how solution S' behaves upon a request for block b_j for $i + 1 \leq j \leq m - 1$. The inductive hypothesis is that property 1 holds when b_j is requested. Because $z = b_m$ is the block in $C_{S,i}$ whose next reference is furthest in the future, we know that $b_j \neq z$. We consider several scenarios:

   - If $C_{S,j} = C_{S',j}$ (so that $|D_j| = k$), then solution S' makes the same decision upon the request for b_j as S makes, so that $C_{S,j+1} = C_{S',j+1}$.
   - If $|D_j| = k - 1$ and $b_j \in D_j$, then both caches already contain block b_j, and both solutions S and S' have cache hits. Therefore, $C_{S,j+1} = C_{S,j}$ and $C_{S',j+1} = C_{S',j}$.
   - If $|D_j| = k - 1$ and $b_j \notin D_j$, then because $C_{S,j} = D_j \cup \{z\}$ and $b_j \neq z$, solution S has a cache miss. It evicts either block z or some block $w \in D_j$.
     - If solution S evicts block z, then $C_{S,j+1} = D_j \cup \{b_j\}$. There are two cases, depending on whether $b_j = y$:
       - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Thus, $C_{S,j+1} = C_{S',j+1}$.
       - If $b_j \neq y$, then solution S' has a cache miss. It evicts block y, so that $C_{S',j+1} = D_j \cup \{b_j\}$, and again $C_{S,j+1} = C_{S',j+1}$.

  - If solution S evicts some block $w \in D_j$, then $C_{S,j+1} = (D_j - \{w\}) \cup \{b_j, z\}$. Once again, there are two cases, depending on whether $b_j = y$:
    - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Since $w \in D_j$ and w was not evicted by solution S', we have $w \in C_{S',j+1}$. Therefore, $w \notin D_{j+1}$ and $b_j \in D_{j+1}$, so that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$. Thus, $C_{S,j+1} = D_{j+1} \cup \{z\}$, $C_{S',j+1} = D_{j+1} \cup \{w\}$, and because $w \neq z$, property 1 holds when block b_{j+1} is requested. (In other words, block w replaces block y in property 1.)
    - If $b_j \neq y$, then solution S' has a cache miss. It evicts block w, so that $C_{S',j+1} = (D_j - \{w\}) \cup \{b_j, y\}$. Therefore, we have that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$ and so $C_{S,j+1} = D_{j+1} \cup \{z\}$ and $C_{S',j+1} = D_{j+1} \cup \{y\}$.

2. In the above discussion about maintaining property 1, solution S may have a cache hit in only the first two cases, and solution S' has a cache hit in these cases if and only if S does.

3. If $C_{S,m} = C_{S',m}$, then solution S' makes the same decision upon the request for block $z = b_m$ as S makes, so that $C_{S,m+1} = C_{S',m+1}$. If $C_{S,m} \neq C_{S',m}$, then by property 1, $C_{S,m} = D_m \cup \{z\}$ and $C_{S',m} = D_m \cup \{y\}$, where $y \neq z$. In this case, solution S has a cache hit, so that $C_{S,m+1} = C_{S,m} = D_m \cup \{z\}$. Solution S' evicts block y and brings in block z, so that $C_{S',m+1} = D_m \cup \{z\} = C_{S,m+1}$. Thus, regardless of whether or not $C_{S,m} = C_{S',m}$, we have $C_{S,m+1} = C_{S',m+1}$, and starting with the request for block b_{m+1}, solution S' simply makes the same decisions as S.

4. By property 2, upon the requests for blocks $b_i, \ldots, b_{m-1}$, whenever solution S has a cache hit, so does S'. Only the request for block $b_m = z$ remains to be considered. If S has a cache miss upon the request for b_m, then regardless of whether S' has a cache hit or a cache miss, we are done: S' has at most the same number of cache misses as S.

   So now suppose that S has a cache hit and S' has a cache miss upon the request for b_m. We'll show that there exists a request for at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ in which the request results in a cache miss for S and a cache hit for S', thereby compensating for what happens upon the request for block b_m. The proof is by contradiction. Assume that no request for blocks $b_{i+1}, \ldots, b_{m-1}$ results in a cache miss for S and a cache hit for S'.

   We start by observing that once the caches $C_{S,j}$ and $C_{S'j}$ are equal for some $j > i$, they remain equal thereafter. Observe also that if $b_m \in C_{S,m}$ and $b_m \notin C_{S',m}$, then $C_{S,m} \neq C_{S',m}$. Therefore, solution S cannot have evicted block z upon the requests for blocks $b_i, \ldots, b_{m-1}$, for if it had, then these two cache configurations would be equal. The remaining possibility is that upon each of these requests, we had $C_{S,j} = D_j \cup \{z\}$, $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$, and solution S evicted some block $w \in D_j$. Moreover, since none of these requests resulted in a cache miss for S and a cache hit for S', the case of $b_j = y$ never occurred. That is, for every request of blocks $b_{i+1}, \ldots, b_{m-1}$, the requested block b_j was never the block $y \in C_{S',j} - C_{S,j}$. In these cases, after processing the request, we had $C_{S',j+1} = D_{j+1} \cup \{y\}$: the difference between the two caches did not change. Now, let's go back to the request for block b_i, where afterward, we had $C_{S',i+1} = D_{i+1} \cup \{x\}$. Because every succeeding request until requesting block b_m did not change the difference between the caches, we had $C_{S',j} = D_j \cup \{x\}$ for $j = i+1, \ldots, m$.

By definition, block $z = b_m$ is requested after block x. That means at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ is block x. But for $j = i+1, \ldots, m$, we have $x \in C_{S',j}$ and $x \notin C_{S,j}$, so that at least one of these requests had a cache hit for S' and a cache miss for S, a contradiction. We conclude that if solution S has a cache hit and solution S' has a cache miss upon the request for block b_m, then some earlier request had the opposite result, and so solution S' produces no more cache misses than solution S. Since S is assumed to be optimal, S' is optimal as well. ■

Along with the optimal-substructure property, Theorem 15.5 tells us that the furthest-in-future strategy yields the minimum number of cache misses.

Exercises

15.4-1

Write pseudocode for a cache manager that uses the furthest-in-future strategy. It should take as input a set C of blocks in the cache, the number of blocks k that the cache can hold, a sequence $b_1, b_2, \ldots, b_n$ of requested blocks, and the index i into the sequence for the block b_i being requested. For each request, it should print out whether a cache hit or cache miss occurs, and for each cache miss, it should also print out which block, if any, is evicted.

15.4-2

Real cache managers do not know the future requests, and so they often use the past to decide which block to evict. The ***least-recently-used***, or ***LRU***, strategy evicts the block that, of all blocks currently in the cache, was the least recently requested. (You can think of LRU as "furthest-in-past.") Give an example of a request sequence in which the LRU strategy is not optimal, by showing that it induces more cache misses than the furthest-in-future strategy does on the same request sequence.

15.4-3

Professor Croesus suggests that in the proof of Theorem 15.5, the last clause in property 1 can change to $C_{S',j} = D_j \cup \{x\}$ or, equivalently, require the block y given in property 1 to always be the block x evicted by solution S upon the request for block b_i. Show where the proof breaks down with this requirement.

15.4-4

This section has assumed that at most one block is placed into the cache whenever a block is requested. You can imagine, however, a strategy in which multiple blocks may enter the cache upon a single request. Show that for every solution that allows multiple blocks to enter the cache upon each request, there is another solution that brings in only one block upon each request and is at least as good.

Problems

15-1 Coin changing

Consider the problem of making change for n cents using the smallest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

15-2 Scheduling to minimize average completion time

You are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete. Let C_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} C_i$. For example, suppose that there are two tasks a_1 and a_2 with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then we have $C_2 = 5$, $C_1 = 8$, and the average completion time is $(5+8)/2 = 6.5$. If task a_1 runs first, however, then we have $C_1 = 3$, $C_2 = 8$, and the average completion time is $(3+8)/2 = 5.5$.

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run nonpreemptively, that is, once task a_i starts, it must run continuously for p_i units of time until it is done. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its ***release time*** b_i. Suppose also that tasks may be ***preempted***, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $b_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

Chapter notes

Much more material on greedy algorithms can be found in Lawler [276] and Papadimitriou and Steiglitz [353]. The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [131].

The proof of correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [179].

Huffman codes were invented in 1952 [233]. Lelewer and Hirschberg [294] surveys data-compression techniques known as of 1987.

The furthest-in-future strategy was proposed by Belady [41], who suggested it for virtual-memory systems. Alternative proofs that furthest-in-future is optimal appear in articles by Lee et al. [284] and Van Roy [443].

16 Amortized Analysis

Imagine that you join Buff's Gym. Buff charges a membership fee of \$60 per month, plus \$3 for every time you use the gym. Because you are disciplined, you visit Buff's Gym every day during the month of November. On top of the \$60 monthly charge for November, you pay another $3 \times \$30 = \$90$ that month. Although you can think of your fees as a flat fee of \$60 and another \$90 in daily fees, you can think about it in another way. All together, you pay \$150 over 30 days, or an average of \$5 per day. When you look at your fees in this way, you are ***amortizing*** the monthly fee over the 30 days of the month, spreading it out at \$2 per day.

You can do the same thing when you analyze running times. In an ***amortized analysis***, you average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, you show that if you average over a sequence of operations, then the average cost of an operation is small, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved. An amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 16.1 starts with aggregate analysis, in which you determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. You take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 16.2 covers the accounting method, in which you determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 16.3 discusses the potential method, which is like the accounting method in that you determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We'll use use two examples in this chapter to examine each of these three methods. One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not —appear in the code. If, for example, you assign a credit to an object x when using the accounting method, you have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When you perform an amortized analysis, you often gain insight into a particular data structure, and this insight can help you optimize the design. For example, Section 16.4 will use the potential method to analyze a dynamically expanding and contracting table.

16.1 Aggregate analysis

In ***aggregate analysis***, you show that for all n, a sequence of n operations takes $T(n)$ *worst-case* time in total. In the worst case, the average cost, or ***amortized cost***, per operation is therefore $T(n)/n$. This amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

As the first example of aggregate analysis, let's analyze stacks that have been augmented with a new operation. Section 10.1.3 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S.

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

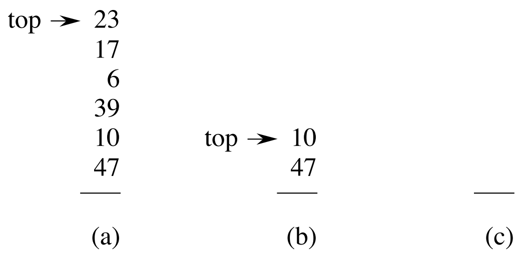


Figure 16.1 The action of MULTIPOP on a stack S, shown initially in **(a)**. The top 4 objects are popped by MULTIPOP$(S, 4)$, whose result is shown in **(b)**. The next operation is MULTIPOP$(S, 7)$, which empties the stack—shown in **(c)**—since fewer than 7 objects remained.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$.

Now let's add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. Of course, the procedure assumes that k is positive, and otherwise, the MULTIPOP operation leaves the stack unchanged. In the pseudocode for MULTIPOP, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise. Figure 16.1 shows an example of MULTIPOP.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and $k > 0$
2      POP(S)
3      $k = k - 1$
```

What is the running time of MULTIPOP(S, k) on a stack of s objects? The actual running time is linear in the number of POP operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number $\min\{s, k\}$ of objects popped off the stack. Each iteration of the loop makes one call to POP in line 2. Thus, the total cost of MULTIPOP is $\min\{s, k\}$, and the actual running time is a linear function of this cost.

Now let's analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since the sequence contains at most n MULTIPOP operations costing $O(n)$ each.

Although this analysis is correct, the $O(n^2)$ result, which came from considering the worst-case cost of each operation individually, is not tight.

Yes, a single MULTIPOP might be expensive, but an aggregate analysis shows that any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack has an upper bound on its cost of $O(n)$. Why? An object cannot be popped from the stack unless it was first pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n. For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. Averaging over the n operations gives an average cost per operation of $O(n)/n = O(1)$. Aggregate analysis assigns the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

To recap: although the average cost, and hence the running time, of a stack operation is $O(1)$, the analysis did not rely on probabilistic reasoning. Instead, the analysis yielded a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded that the average cost per operation—that is, the amortized cost—is $O(1)$.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing a k-bit binary counter that counts upward from 0. An array $A[0:k-1]$ of bits represents the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$. To add 1 (modulo 2^k) to the value in the counter, call the INCREMENT procedure.

```
INCREMENT(A, k)
1  $i = 0$
2  while $i < k$ and $A[i] == 1$
3      $A[i] = 0$
4      $i = i + 1$
5  if $i < k$
6      $A[i] = 1$
```

Figure 16.2 shows what happens to a binary counter when INCREMENT is called 16 times, starting with the initial value 0 and ending with the value 16. Each iteration of the **while** loop in lines 2–4 adds a 1 into position i. If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 16.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded in blue. The running cost for flipping bits is shown at the right. The total cost is always less than twice the total number of INCREMENT operations.

position $i + 1$ during the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, $A[i]$ must be 0, so that line 6 adds a 1 into position i, flipping the 0 to a 1. If the loop ends with $i = k$, then the call of INCREMENT flipped all k bits from 1 to 0. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes $\Theta(k)$ time in the worst case, in which all the bits in array A are 1. Thus, a sequence of n INCREMENT operations on an initially zero counter takes $O(nk)$ time in the worst case.

Although a single call of INCREMENT might flip all k bits, not all bits flip upon each call. (Note the similarity to MULTIPOP, where a single call might pop many objects, but not every call pops many objects.) As Figure 16.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of n INCREMENT operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT operations. In general, for $i = 0, 1, \ldots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number of flips in the sequence is thus

$$\begin{aligned}\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n ,\end{aligned}$$

by equation (A.7) on page 1142. Thus, a sequence of n INCREMENT operations on an initially zero counter takes $O(n)$ time in the worst case. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Exercises

16.1-1
If the set of stack operations includes a MULTIPUSH operation, which pushes k items onto the stack, does the $O(1)$ bound on the amortized cost of stack operations continue to hold?

16.1-2
Show that if a DECREMENT operation is included in the k-bit counter example, n operations can cost as much as $\Theta(nk)$ time.

16.1-3
Use aggregate analysis to determine the amortized cost per operation for a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise.

16.2 The accounting method

In the ***accounting method*** of amortized analysis, you assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount that you charge an operation is its ***amortized cost***. When an operation's amortized cost exceeds its actual cost, you assign the difference to specific objects in the data structure as ***credit***. Credit can help pay for later operations whose amortized cost is less than their actual cost. Thus, you can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

You must choose the amortized costs of operations carefully. If you want to use amortized costs to show that in the worst case the average cost per operation is small, you must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, the upper bound must apply to all sequences of operations. Let's denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\widehat{c}_i$. Then you need to have

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i \tag{16.1}$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or $\sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i$. By inequality (16.1), the total credit associated with the data structure must be nonnegative at all times. If you ever allowed the total credit to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred. In that case, for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, you must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, we return to the stack example. Recall that the actual costs of the operations were

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min\{s,k\}$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

The amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable, and thus all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

Now let's see how to pay for any sequence of stack operations by charging the amortized costs. Let \$1 represent each unit of cost. At first, the stack is empty. Recall the analogy of Section 10.1.3 between the stack data structure and a stack of plates in a cafeteria. Upon pushing a plate onto the stack, use \$1 to pay the actual cost of the push, leaving a credit of \$1 (out of the \$2 charged). Place that \$1 of credit on top of the plate. At any point in time, every plate on the stack has \$1 of credit on it.

The \$1 stored on the plate serves to prepay the cost of popping the plate from the stack. A POP operation incurs no charge: pay the actual cost of popping a plate by taking the \$1 of credit off the plate. Thus, by charging the PUSH operation a little bit more, we can view the POP operation as free.

Moreover, the MULTIPOP operation also incurs no charge, since it's just repeated POP operations, each of which is free. If a MULTIPOP operation pops k plates, then the actual cost is paid by the k dollars stored on the k plates. Because each plate on the stack has \$1 of credit on it, and the stack always has a nonnegative number of plates, the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, let's analyze the INCREMENT operation on a binary counter that starts at 0. Recall that the running time of this operation is proportional to the number of bits flipped, which serves as the cost for this example. Again, we'll use \$1 to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, the amortized cost to set a 0-bit to 1 is \$2. When a bit is set to 1, \$1 of the \$2 pays to actually set the bit. The second \$1 resides on the bit as credit to be used later if and when the bit is reset to 0. At any point in time, every 1-bit in the counter has \$1 of credit on it, and thus resetting a bit to 0 can be viewed as costing nothing, and the \$1 on the bit prepays for the reset.

Here is how to determine the amortized cost of INCREMENT. The cost of resetting the bits to 0 within the **while** loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit to 1, in line 6, and therefore the amortized cost of an INCREMENT operation is at most \$2. The number of 1-bits in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Exercises

16.2-1

You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

16.2-2

Redo Exercise 16.1-3 using an accounting method of analysis.

16.2-3

You wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(n)$ time on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

16.3 The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. The potential applies to the data structure as a whole rather than to specific objects within the data structure.

The potential method works as follows. Starting with an initial data structure D_0, a sequence of n operations occurs. For each $i = 1, 2, \ldots, n$, let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with D_i. The ***amortized cost*** $\widehat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \tag{16.2}$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. By equation (16.2), the total amortized cost of the n operations is

$$\begin{aligned} \sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \tag{16.3}$$

The second equation follows from equation (A.12) on page 1143 because the $\Phi(D_i)$ terms telescope.

If you can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \widehat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$. In practice, you don't always know how many operations might be performed. Therefore, if you require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then you guarantee, as in the accounting method, that you've paid in advance. It's usually simplest to just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i. (See Exercise 16.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is positive, then the amortized cost $\widehat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the ith operation, and the decrease in the potential pays for the actual cost of the operation.

The amortized costs defined by equations (16.2) and (16.3) depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs, yet still be upper bounds on the actual costs. You will often find tradeoffs that you can make in choosing a potential function. The best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. The potential of the empty initial stack D_0 is $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) . \end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Now let's compute the amortized costs of the various stack operations. If the ith operation on a stack containing s objects is a PUSH operation, then the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 . \end{aligned}$$

By equation (16.2), the amortized cost of this PUSH operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2\,.\end{aligned}$$

Suppose that the ith operation on the stack of s objects is MULTIPOP(S, k), which causes $k' = \min\{s, k\}$ objects to be popped off the stack. The actual cost of the operation is k', and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'\,.$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0\,.\end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing a binary counter

As another example of the potential method, we revisit incrementing a k-bit binary counter. This time, the potential of the counter after the ith INCREMENT operation is defined to be the number of 1-bits in the counter after the ith operation, which we'll denote by b_i.

Here is how to compute the amortized cost of an INCREMENT operation. Suppose that the ith INCREMENT operation resets t_i bits to 0. The actual cost c_i of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1. If $b_i = 0$, then the ith operation had reset all k bits to 0, and so $b_{i-1} = t_i = k$. If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. In either case, $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i\,.\end{aligned}$$

The amortized cost is therefore

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2\,.\end{aligned}$$

If the counter starts at 0, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method provides a simple and clever way to analyze the counter even when it does not start at 0. The counter starts with b_0 1-bits, and after n INCREMENT operations it has b_n 1-bits, where $0 \leq b_0, b_n \leq k$. Rewrite equation (16.3) as

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

Since $\Phi(D_0) = b_0$, $\Phi(D_n) = b_n$, and $\widehat{c}_i \leq 2$ for all $1 \leq i \leq n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 .\end{aligned}$$

In particular, $b_0 \leq k$ means that as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if at least $n = \Omega(k)$ INCREMENT operations occur, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Exercises

16.3-1

Suppose you have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

16.3-2

Redo Exercise 16.1-3 using a potential method of analysis.

16.3-3

Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

16.3-4

What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

16.3-5

Show how to implement a queue with two ordinary stacks (Exercise 10.1-7) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

16.3-6

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT(S, x) inserts x into S.

DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

16.4 Dynamic tables

When you design an application that uses a table, you do not always know in advance how many items the table will hold. You might allocate space for the table, only to find out later that it is not enough. The program must then reallocate the table with a larger size and copy all items stored in the original table over into the new, larger table. Similarly, if many items have been deleted from the table, it might be worthwhile to reallocate the table with a smaller size. This section studies this problem of dynamically expanding and contracting a table. Amortized analyses will show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, you'll see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Let's assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single ***slot***, that is, a space for one item. Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant: it could be a stack (Section 10.1.3), a heap (Chapter 6), a hash table (Chapter 11), or something else.

It is convenient to use a concept introduced in Section 11.2, where we analyzed hashing. The ***load factor*** $\alpha(T)$ of a nonempty table T is defined as the number of items stored in the table divided by the size (number of slots) of the table. An empty table (one with no slots) has size 0, and its load factor is defined to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

We start by analyzing a dynamic table that allows only insertion and then move on to the more general case that supports both insertion and deletion.

16.4.1 Table expansion

Let's assume that storage for a table is allocated as an array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1.[1] In some software environments, upon an attempt to insert an item into a full table, the only alternative is to abort with an error. The scenario in this section assumes, however, that the software environment, like many modern ones, provides a memory-management system that can allocate and free blocks of storage on request. Thus, upon inserting an item into a full table, the system can ***expand*** the table by allocating a new table with more slots than the old table had. Because the table must always reside in contiguous memory, the system must allocate a new array for the larger table and then copy items from the old table into the new table.

A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least $1/2$, and thus the amount of wasted space never exceeds half the total space in the table.

The TABLE-INSERT procedure on the following page assumes that T is an object representing the table. The attribute $T.table$ contains a pointer to the block of storage representing the table, $T.num$ contains the number of items in the table, and $T.size$ gives the total number of slots in the table. Initially, the table is empty: $T.num = T.size = 0$.

There are two types of insertion here: the TABLE-INSERT procedure itself and the ***elementary insertion*** into a table in lines 6 and 10. We can analyze the running time of TABLE-INSERT in terms of the number of elementary insertions by assigning a cost of 1 to each elementary insertion. In most computing environments, the overhead for allocating an initial table in line 2 is constant and the overhead for allocating and freeing storage in lines 5 and 7 is dominated by the cost of transfer-

[1] In some situations, such as an open-address hash table, it's better to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 16.4-2.)

```
TABLE-INSERT(T, x)
1  if T.size == 0
2      allocate T.table with 1 slot
3      T.size = 1
4  if T.num == T.size
5      allocate new-table with 2 · T.size slots
6      insert all items in T.table into new-table
7      free T.table
8      T.table = new-table
9      T.size = 2 · T.size
10 insert x into T.table
11 T.num = T.num + 1
```

ring items in line 6. Thus, the actual running time of TABLE-INSERT is linear in the number of elementary insertions. An ***expansion*** occurs when lines 5–9 execute.

Now, we'll use all three amortized analysis techniques to analyze a sequence of n TABLE-INSERT operations on an initially empty table. First, we need to determine the actual cost c_i of the ith operation. If the current table has room for the new item (or if this is the first operation), then $c_i = 1$, since the only elementary insertion performed is the one in line 10. If the current table is full, however, and an expansion occurs, then $c_i = i$: the cost is 1 for the elementary insertion in line 10 plus $i - 1$ for the items copied from the old table to the new table in line 6. For n operations, the worst-case cost of an operation is $O(n)$, which leads to an upper bound of $O(n^2)$ on the total running time for n operations.

This bound is not tight, because the table rarely expands in the course of n TABLE-INSERT operations. Specifically, the ith operation causes an expansion only when $i - 1$ is an exact power of 2. The amortized cost of an operation is in fact $O(1)$, as an aggregate analysis shows. The cost of the ith operation is

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2} , \\ 1 & \text{otherwise} . \end{cases}$$

The total cost of n TABLE-INSERT operations is therefore

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \qquad \text{(by equation (A.6) on page 1142)} \\ &= 3n , \end{aligned}$$

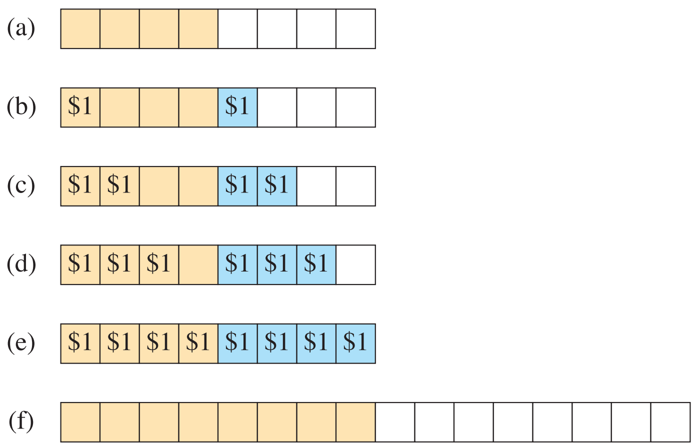


Figure 16.3 Analysis of table expansion by the accounting method. Each call of TABLE-INSERT charges \$3 as follows: \$1 to pay for the elementary insertion, \$1 on the item inserted as prepayment for it to be reinserted later, and \$1 on an item that was already in the table, also as prepayment for reinsertion. **(a)** The table immediately after an expansion, with 8 slots, 4 items (tan slots), and no stored credit. **(b)–(e)** After each of 4 calls to TABLE-INSERT, the table has one more item, with \$1 stored on the new item and \$1 stored on one of the 4 items that were present immediately after the expansion. Slots with these new items are blue. **(f)** Upon the next call to TABLE-INSERT, the table is full, and so it expands again. Each item had \$1 to pay for it to be reinserted. Now the table looks as it did in part (a), with no stored credit but 16 slots and 8 items.

because at most n operations cost 1 each and the costs of the remaining operations form a geometric series. Since the total cost of n TABLE-INSERT operations is bounded by $3n$, the amortized cost of a single operation is at most 3.

The accounting method can provide some intuition for why the amortized cost of a TABLE-INSERT operation should be 3. You can think of each item paying for three elementary insertions: inserting itself into the current table, moving itself the next time that the table expands, and moving some other item that was already in the table the next time that the table expands. For example, suppose that the size of the table is m immediately after an expansion, as shown in Figure 16.3 for $m = 8$. Then the table holds $m/2$ items, and it contains no credit. Each call of TABLE-INSERT charges \$3. The elementary insertion that occurs immediately costs \$1. Another \$1 resides on the item inserted as credit. The third \$1 resides as credit on one of the $m/2$ items already in the table. The table will not fill again until another $m/2 - 1$ items have been inserted, and thus, by the time the table contains m items and is full, each item has \$1 on it to pay for it to be reinserted it during the expansion.

Now, let's see how to use the potential method. We'll use it again in Section 16.4.2 to design a TABLE-DELETE operation that has an $O(1)$ amortized cost as well. Just as the accounting method had no stored credit immediately after an expansion—that is, when $T.num = T.size/2$—let's define the potential to be 0 when $T.num = T.size/2$. As elementary insertions occur, the potential needs to increase enough to pay for all the reinsertions that will happen when the table next expands. The table fills after another $T.size/2$ calls of TABLE-INSERT, when $T.num = T.size$. The next call of TABLE-INSERT after these $T.size/2$ calls triggers an expansion with a cost of $T.size$ to reinsert all the items. Therefore, over the course of $T.size/2$ calls of TABLE-INSERT, the potential must increase from 0 to $T.size$. To achieve this increase, let's design the potential so that each call of TABLE-INSERT increases it by

$$\frac{T.size}{T.size/2} = 2 ,$$

until the table expands. You can see that the potential function

$$\Phi(T) = 2(T.num - T.size/2) \tag{16.4}$$

equals 0 immediately after the table expands, when $T.num = T.size/2$, and it increases by 2 upon each insertion until the table fills. Once the table fills, that is, when $T.num = T.size$, the potential $\Phi(T)$ equals $T.size$. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

To analyze the amortized costs of table operations, it is convenient to think in terms of the change in potential due to each operation. Letting Φ_i denote the potential after the ith operation, we can rewrite equation (16.2) as

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= c_i + \Delta\Phi_i , \end{aligned}$$

where $\Delta\Phi_i$ is the change in potential due to the ith operation. First, consider the case when the ith insertion does not cause the table to expand. In this case, $\Delta\Phi_i$ is 2. Since the actual cost c_i is 1, the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + \Delta\Phi_i \\ &= 1 + 2 \\ &= 3 . \end{aligned}$$

Now, consider the change in potential when the table does expand during the ith insertion because it was full immediately before the insertion. Let num_i denote the number of items stored in the table after the ith operation and $size_i$ denote the total size of the table after the ith operation, so that $size_{i-1} = num_{i-1} = i - 1$

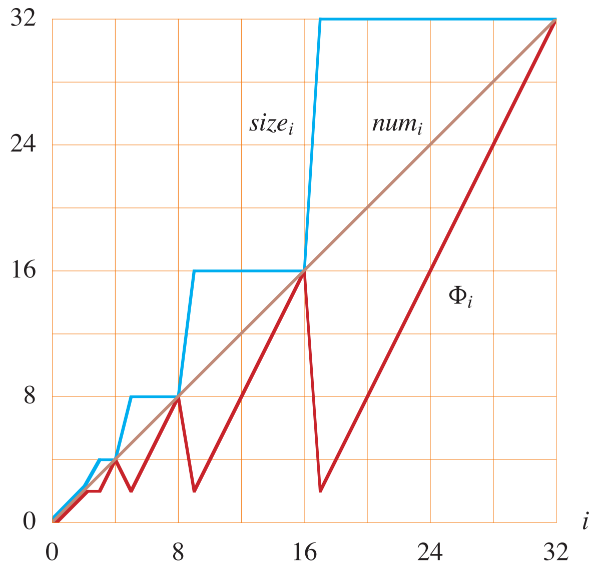


Figure 16.4 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table (the brown line), the number $size_i$ of slots in the table (the blue line), and the potential $\Phi_i = 2(num_i - size_i/2)$ (the red line), each being measured after the ith operation. Immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterward, the potential drops to 0, but it immediately increases by 2 upon insertion of the item that caused the expansion.

and therefore $\Phi_{i-1} = 2(size_{i-1} - size_{i-1}/2) = size_{i-1} = i - 1$. Immediately after the expansion, the potential goes down to 0, and then the new item is inserted, causing the potential to increase to $\Phi_i = 2$. Thus, when the ith insertion triggers an expansion, $\Delta\Phi_i = 2 - (i - 1) = 3 - i$. When the table expands in the ith TABLE-INSERT operation, the actual cost c_i equals i (to reinsert $i - 1$ items and insert the ith item), giving an amortized cost of

$$\begin{aligned} \hat{c}_i &= c_i + \Delta\Phi_i \\ &= i + (3 - i) \\ &= 3 . \end{aligned}$$

Figure 16.4 plots the values of num_i, $size_i$, and Φ_i against i. Notice how the potential builds to pay for expanding the table.

16.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. In order to limit the amount of wasted space, however, you might want to ***contract*** the table when the load factor becomes too small. Table contraction is analogous to table expansion: when the number of items in the table drops too low, allocate a new, smaller table and then copy the items from the old table into the new one. You can then free the storage for the old table by returning it to the memory-management system. In order to not waste space, yet keep the amortized costs low, the insertion and deletion procedures should preserve two properties:

- the load factor of the dynamic table is bounded below by a positive constant, as well as above by 1, and
- the amortized cost of a table operation is bounded above by a constant.

The actual cost of each operation equals the number of elementary insertions or deletions.

You might think that if you double the table size upon inserting an item into a full table, then you should halve the size when deleting an item that would cause the table to become less than half full. This strategy does indeed guarantee that the load factor of the table never drops below $1/2$. Unfortunately, it can also cause the amortized cost of an operation to be quite large. Consider the following scenario. Perform n operations on a table T of size $n/2$, where n is an exact power of 2. The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertions, $T.num = T.size = n/2$. For the second $n/2$ operations, perform the following sequence:

insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

The first insertion causes the table to expand to size n. The two deletions that follow cause the table to contract back to size $n/2$. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$, and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of an operation $\Theta(n)$.

The problem with this strategy is that after the table expands, not enough deletions occur to pay for a contraction. Likewise, after the table contracts, not enough insertions take place to pay for an expansion.

How can we solve this problem? Allow the load factor of the table to drop below $1/2$. Specifically, continue to double the table size upon inserting an item into a full table, but halve the table size when deleting an item causes the table to become less than $1/4$ full, rather than $1/2$ full as before. The load factor of the table is therefore bounded below by the constant $1/4$, and the load factor is $1/2$ immediately after a contraction.

An expansion or contraction should exhaust all the built-up potential, so that immediately after expansion or contraction, when the load factor is $1/2$, the table's potential is 0. Figure 16.5 shows the idea. As the load factor deviates from $1/2$, the

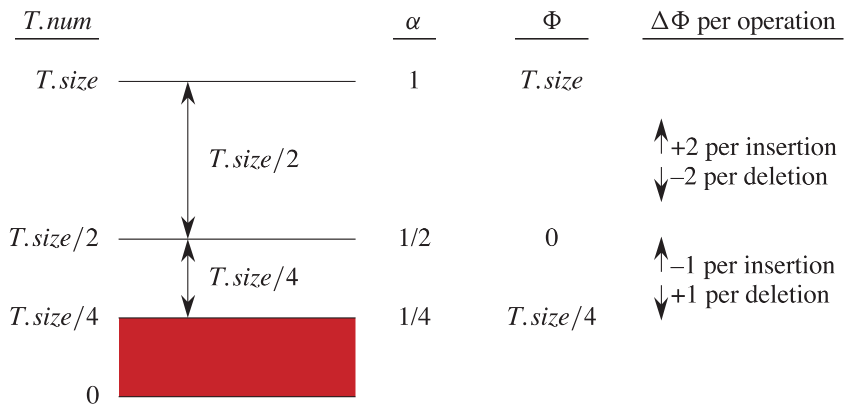


Figure 16.5 How to think about the potential function Φ for table insertion and deletion. When the load factor α is $1/2$, the potential is 0. In order to accumulate sufficient potential to pay for reinserting all $T.size$ items when the table fills, the potential needs to increase by 2 upon each insertion when $\alpha \geq 1/2$. Correspondingly, the potential decreases by 2 upon each deletion that leaves $\alpha \geq 1/2$. In order to accrue enough potential to cover the cost of reinserting all $T.size/4$ items when the table contracts, the potential needs to increase by 1 upon each deletion when $\alpha < 1/2$, and correspondingly the potential decreases by 1 upon each insertion that leaves $\alpha < 1/2$. The red area represents load factors less than $1/4$, which are not allowed.

potential increases so that by the time an expansion or contraction occurs, the table has garnered sufficient potential to pay for copying all the items into the newly allocated table. Thus, the potential function should grow to $T.num$ by the time that the load factor has either increased to 1 or decreased to $1/4$. Immediately after either expanding or contracting the table, the load factor goes back to $1/2$ and the table's potential reduces back to 0.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT. We assume that if a contraction occurs during TABLE-DELETE, it occurs after the item is deleted from the table. The analysis assumes that whenever the number of items in the table drops to 0, the table occupies no storage. That is, if $T.num = 0$, then $T.size = 0$.

How do we design a potential function that gives constant amortized time for both insertion and deletion? When the load factor is at least $1/2$, the same potential function, $\Phi(T) = 2(T.num - T.size/2)$, that we used for insertion still works. When the table is at least half full, each insertion increases the potential by 2 if the table does not expand, and each deletion reduces the potential by 2 if it does not cause the load factor to drop below $1/2$.

What about when the load factor is less than $1/2$, that is, when $1/4 \leq \alpha(T) < 1/2$? As before, when $\alpha(T) = 1/2$, so that $T.num = T.size/2$, the potential $\Phi(T)$ should be 0. To get the load factor from $1/2$ down to $1/4$, $T.size/4$ deletions need to occur, at which time $T.num = T.size/4$. To pay for all the reinsertions, the potential must increase from 0 to $T.size/4$ over these $T.size/4$ deletions. Therefore, for each call of TABLE-DELETE until the table contracts, the potential should increase by

$$\frac{T.size/4}{T.size/4} = 1 .$$

Likewise, when $\alpha < 1/2$, each call of TABLE-INSERT should decrease the potential by 1. When $1/4 \le \alpha(T) < 1/2$, the potential function

$$\Phi(T) = T.size/2 - T.num$$

produces this desired behavior.

Putting the two cases together, we get the potential function

$$\Phi(T) = \begin{cases} 2(T.num - T.size/2) & \text{if } \alpha(T) \ge 1/2 , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 . \end{cases} \tag{16.5}$$

The potential of an empty table is 0 and the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence. Figure 16.6 illustrates how the potential function behaves over a sequence of insertions and deletions.

Now, let's determine the amortized costs of each operation. As before, let num_i denote the number of items stored in the table after the ith operation, $size_i$ denote the total size of the table after the ith operation, $\alpha_i = num_i/size_i$ denote the load factor after the ith operation, Φ_i denote the potential after the ith operation, and $\Delta\Phi_i$ denote the change in potential due to the ith operation. Initially, $num_0 = 0$, $size_0 = 0$, and $\Phi_0 = 0$.

The cases in which the table does not expand or contract and the load factor does not cross $\alpha = 1/2$ are straightforward. As we have seen, if $\alpha_{i-1} \ge 1/2$ and the ith operation is an insertion that does not cause the table to expand, then $\Delta\Phi_i = 2$. Likewise, if the ith operation is a deletion and $\alpha_i \ge 1/2$, then $\Delta\Phi_i = -2$. Furthermore, if $\alpha_{i-1} < 1/2$ and the ith operation is a deletion that does not trigger a contraction, then $\Delta\Phi_i = 1$, and if the ith operation is an insertion and $\alpha_i < 1/2$, then $\Delta\Phi_i = -1$. In other words, if no expansion or contraction occurs and the load factor does not cross $\alpha = 1/2$, then

- if the load factor stays at or above $1/2$, then the potential increases by 2 for an insertion and decreases by 2 for a deletion, and
- if the load factor stays below $1/2$, then the potential increases by 1 for a deletion and decreases by 1 for an insertion.

In each of these cases, the actual cost c_i of the ith operation is just 1, and so

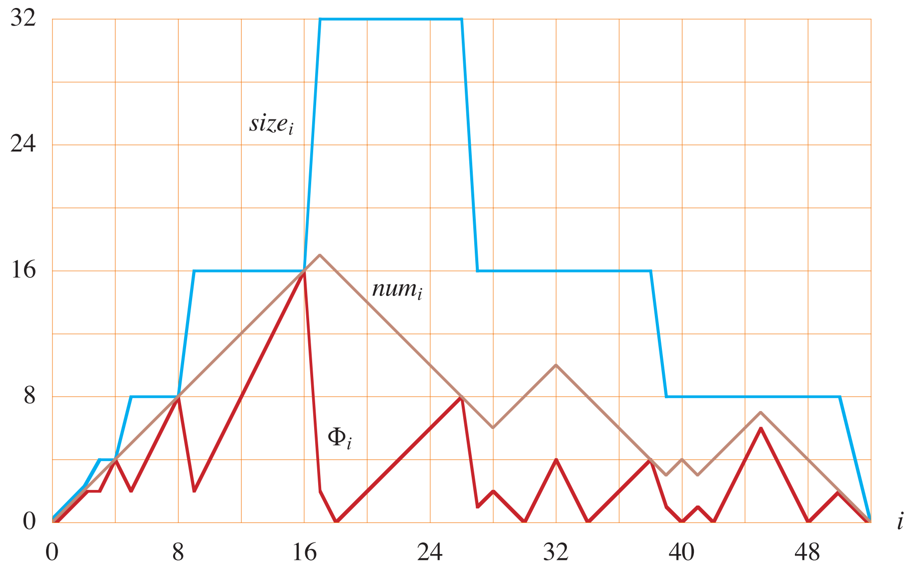


Figure 16.6 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table (the brown line), the number $size_i$ of slots in the table (the blue line), and the potential (the red line)

$$\Phi_i = \begin{cases} 2(num_i - size_i/2) & \text{if } \alpha_i \geq 1/2 , \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2 , \end{cases}$$

where $\alpha_i = num_i/size_i$, each measured after the ith operation. Immediately before an expansion or contraction, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table.

- if the ith operation is an insertion, its amortized cost $\hat{c}_i$ is $c_i + \Delta\Phi_i$, which is $1 + 2 = 3$ if the load factor stays at or above $1/2$, and $1 + (-1) = 0$ if the load factor stays below $1/2$, and
- if the ith operation is a deletion, its amortized cost $\hat{c}_i$ is $c_i + \Delta\Phi_i$, which is $1 + (-2) = -1$ if the load factor stays at or above $1/2$, and $1 + 1 = 2$ if the load factor stays below $1/2$.

Four cases remain: an insertion that takes the load factor from below $1/2$ to $1/2$, a deletion that takes the load factor from $1/2$ to below $1/2$, a deletion that causes the table to contract, and an insertion that causes the table to expand. We analyzed that last case at the end of Section 16.4.1 to show that its amortized cost is 3.

When the ith operation is a deletion that causes the table to contract, we have $num_{i-1} = size_{i-1}/4$ before the contraction, then the item is deleted, and finally $num_i = size_i/2 - 1$ after the contraction. Thus, by equation (16.5) we have

$$\begin{aligned}\Phi_{i-1} &= \mathit{size}_{i-1}/2 - \mathit{num}_{i-1} \\ &= \mathit{size}_{i-1}/2 - \mathit{size}_{i-1}/4 \\ &= \mathit{size}_{i-1}/4 ,\end{aligned}$$

which also equals the actual cost c_i of deleting one item and copying $\mathit{size}_{i-1}/4 - 1$ items into the new, smaller table. Since $\mathit{num}_i = \mathit{size}_i/2 - 1$ after the operation has completed, $\alpha_i < 1/2$, and so

$$\begin{aligned}\Phi_i &= \mathit{size}_i/2 - \mathit{num}_i \\ &= 1 ,\end{aligned}$$

giving $\Delta\Phi_i = 1 - \mathit{size}_{i-1}/4$. Therefore, when the ith operation is a deletion that triggers a contraction, its amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Delta\Phi_i \\ &= \mathit{size}_{i-1}/4 + (1 - \mathit{size}_{i-1}/4) \\ &= 1 .\end{aligned}$$

Finally, we handle the cases where the load factor fits one case of equation (16.5) before the operation and the other case afterward. We start with deletion, where we have $\mathit{num}_{i-1} = \mathit{size}_{i-1}/2$, so that $\alpha_{i-1} = 1/2$, beforehand, and $\mathit{num}_i = \mathit{size}_i/2 - 1$, so that $\alpha_i < 1/2$ afterward. Because $\alpha_{i-1} = 1/2$, we have $\Phi_{i-1} = 0$, and because $\alpha_i < 1/2$, we have $\Phi_i = \mathit{size}_i/2 - \mathit{num}_i = 1$. Thus we get that $\Delta\Phi_i = 1 - 0 = 1$. Since the ith operation is a deletion that does not cause a contraction, the actual cost c_i equals 1, and the amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i = 1 + 1 = 2$.

Conversely, if the ith operation is an insertion that takes the load factor from below $1/2$ to equaling $1/2$, the change in potential $\Delta\Phi_i$ equals -1. Again, the actual cost c_i is 1, and now the amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i = 1 + (-1) = 0$.

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is $O(n)$.

Exercises

16.4-1
Using the potential method, analyze the amortized cost of the first table insertion.

16.4-2
You wish to implement a dynamic, open-address hash table. Why might you consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per

insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

16.4-3

Discuss how to use the accounting method to analyze both the insertion and deletion operations, assuming that the table doubles in size when its load factor exceeds 1 and the table halves in size when its load factor goes below $1/4$.

16.4-4

Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, you contract the table by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function

$$\Phi(T) = |2(T.num - T.size/2)| \ ,$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Problems

16-1 Binary reflected Gray code

A ***binary Gray code*** represents a sequence of nonnegative integers in binary such that to go from one integer to the next, exactly one bit flips every time. The ***binary reflected Gray code*** represents a sequence of the integers 0 to $2^k - 1$ for some positive integer k according to the following recursive method:

- For $k = 1$, the binary reflected Gray code is $\langle 0, 1 \rangle$.
- For $k \geq 2$, first form the binary reflected Gray code for $k - 1$, giving the 2^{k-1} integers 0 to $2^{k-1} - 1$. Then form the reflection of this sequence, which is just the sequence in reverse. (That is, the jth integer in the sequence becomes the $(2^{k-1} - j - 1)$st integer in the reflection). Next, add 2^{k-1} to each of the 2^{k-1} integers in the reflected sequence. Finally, concatenate the two sequences.

For example, for $k = 2$, first form the binary reflected Gray code $\langle 0, 1 \rangle$ for $k = 1$. Its reflection is the sequence $\langle 1, 0 \rangle$. Adding $2^{k-1} = 2$ to each integer in the reflection gives the sequence $\langle 3, 2 \rangle$. Concatenating the two sequences gives $\langle 0, 1, 3, 2 \rangle$ or, in binary, $\langle 00, 01, 11, 10 \rangle$, so that each integer differs from its predecessor by exactly one bit. For $k = 3$, the reflection of the binary reflected Gray code for $k = 2$ is $\langle 2, 3, 1, 0 \rangle$ and adding $2^{k-1} = 4$ gives $\langle 6, 7, 5, 4 \rangle$. Concatenating produces the sequence $\langle 0, 1, 3, 2, 6, 7, 5, 4 \rangle$, which in binary is $\langle 000, 001, 011, 010, 110, 111, 101, 100 \rangle$. In the binary reflected Gray code, only one bit flips even when wrapping around from the last integer to the first.

a. Index the integers in a binary reflected Gray code from 0 to $2^k - 1$, and consider the ith integer in the binary reflected Gray code. To go from the $(i-1)$st integer to the ith integer in the binary reflected Gray code, exactly one bit flips. Show how to determine which bit flips, given the index i.

b. Assuming that given a bit number j, you can flip bit j of an integer in constant time, show how to compute the entire binary reflected Gray code sequence of 2^k numbers in $\Theta(2^k)$ time.

16-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. You can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. Maintain k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i \, 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times, assuming that the only operations are INSERT and SEARCH.

c. Describe how to implement DELETE. Analyze its worst-case and amortized running times, assuming that there can be DELETE, INSERT, and SEARCH operations.

16-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$, which gives the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \le \alpha < 1$. We say that a given node x is ***α-balanced*** if $x.left.size \le \alpha \cdot x.size$ and $x.right.size \le \alpha \cdot x.size$. The tree as a whole is ***α-balanced*** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A $1/2$-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$-balanced. Your algorithm should run in $\Theta(x.size)$ time, and it can use $O(x.size)$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that you implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then you "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$-balanced.

We'll analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, define

$$\Delta(x) = |x.left.size - x.right.size| \ .$$

Define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and also that a $1/2$-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

16-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n + 1)$st node causes $\Omega(\lg n)$ color changes. Then describe a legal red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Although the worst-case number of color changes per operation can be logarithmic, you will prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (*Hint:* Look at Figures 13.5, 13.6, and 13.7 in Sections 13.3 and 13.4.)

You will first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T. Assume that one unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T. Argue that $\Phi(T') = \Phi(T) - 1$.

d. We can break the operation of the RB-INSERT procedure into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

Next you will prove that there are $O(m)$ structural modifications when both insertions and deletions occur. Define, for each node x,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red} , \\ 1 & \text{if } x \text{ is black and has no red children} , \\ 0 & \text{if } x \text{ is black and has one red child} , \\ 2 & \text{if } x \text{ is black and has two red children} . \end{cases}$$

Now redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x) ,$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T.

f. Show that $\Phi(T') \le \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

g. Show that $\Phi(T') \le \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

Chapter notes

Aho, Hopcroft, and Ullman [5] used aggregate analysis to determine the running time of operations on a disjoint-set forest. We'll analyze this data structure using the potential method in Chapter 19. Tarjan [430] surveys the accounting and potential methods of amortized analysis and presents several applications. He attributes the accounting method to several authors, including M. R. Brown, R. E. Tarjan, S. Huddleston, and K. Mehlhorn. He attributes the potential method to D. D. Sleator. The term "amortized" is due to D. D. Sleator and R. E. Tarjan.

Potential functions are also useful for proving lower bounds for certain types of problems. For each configuration of the problem, define a potential function that maps the configuration to a real number. Then determine the potential Φ_{init} of the initial configuration, the potential Φ_{final} of the final configuration, and the maximum change in potential $\Delta\Phi_{\text{max}}$ due to any step. The number of steps must therefore be at least $|\Phi_{\text{final}} - \Phi_{\text{init}}| / |\Delta\Phi_{\text{max}}|$. Examples of potential functions to prove lower bounds in I/O complexity appear in works by Cormen, Sundquist, and Wisniewski [105], Floyd [146], and Aggarwal and Vitter [3]. Krumme, Cybenko, and Venkataraman [271] applied potential functions to prove lower bounds on ***gossiping***: communicating a unique item from each vertex in a graph to every other vertex.

Part V Advanced Data Structures

Introduction

This part returns to studying data structures that support operations on dynamic sets, but at a more advanced level than Part III. One of the chapters, for example, makes extensive use of the amortized analysis techniques from Chapter 16.

Chapter 17 shows how to augment red-black trees—adding additional information in each node—to support dynamic-set operations in addition to those covered in Chapters 12 and 13. The first example augments red-black trees to dynamically maintain order statistics for a set of keys. Another example augments them in a different way to maintain intervals of real numbers. Chapter 17 includes a theorem giving sufficient conditions for when a red-black tree can be augmented while maintaining the $O(\lg n)$ running times for insertion and deletion.

Chapter 18 presents B-trees, which are balanced search trees specifically designed to be stored on disks. Since disks operate much more slowly than random-access memory, B-tree performance depends not only on how much computing time the dynamic-set operations consume but also on how many disk accesses they perform. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, but B-tree operations keep the height low.

Chapter 19 examines data structures for disjoint sets. Starting with a universe of n elements, each initially in its own singleton set, the operation UNION unites two sets. At all times, the n elements are partitioned into disjoint sets, even as calls to the UNION operation change the members of a set dynamically. The query FIND-SET identifies the unique set that contains a given element at the moment. Representing each set as a simple rooted tree yields surprisingly fast operations: a sequence of m operations runs in $O(m\,\alpha(n))$ time, where $\alpha(n)$ is an incredibly slowly growing function—$\alpha(n)$ is at most 4 in any conceivable application. The amortized analysis that proves this time bound is as complex as the data structure is simple.

The topics covered in this part are by no means the only examples of "advanced" data structures. Other advanced data structures include the following:

- ***Fibonacci heaps*** [156] implement mergeable heaps (see Problem 10-2 on page 268) with the operations INSERT, MINIMUM, and UNION taking only $O(1)$ actual and amortized time, and the operations EXTRACT-MIN and DELETE taking $O(\lg n)$ amortized time. The most significant advantage of these data structures, however, is that DECREASE-KEY takes only $O(1)$ amortized time. ***Strict Fibonacci heaps*** [73], developed later, made all of these time bounds actual. Because the DECREASE-KEY operation takes constant amortized time, (strict) Fibonacci heaps constitute key components of some of the asymptotically fastest algorithms to date for graph problems.
- ***Dynamic trees*** [415, 429] maintain a forest of disjoint rooted trees. Each edge in each tree has a real-valued cost. Dynamic trees support queries to find parents, roots, edge costs, and the minimum edge cost on a simple path from a node up to a root. Trees may be manipulated by cutting edges, updating all edge costs on a simple path from a node up to a root, linking a root into another tree, and making a node the root of the tree it appears in. One implementation of dynamic trees gives an $O(\lg n)$ amortized time bound for each operation, while a more complicated implementation yields $O(\lg n)$ worst-case time bounds. Dynamic trees are used in some of the asymptotically fastest network-flow algorithms.
- ***Splay trees*** [418, 429] are a form of binary search tree on which the standard search-tree operations run in $O(\lg n)$ amortized time. One application of splay trees simplifies dynamic trees.
- ***Persistent*** data structures allow queries, and sometimes updates as well, on past versions of a data structure. For example, linked data structures can be made persistent with only a small time and space cost [126]. Problem 13-1 gives a simple example of a persistent dynamic set.
- Several data structures allow a faster implementation of dictionary operations (INSERT, DELETE, and SEARCH) for a restricted universe of keys. By taking advantage of these restrictions, they are able to achieve better worst-case asymptotic running times than comparison-based data structures. If the keys are unique integers drawn from the set $\{0, 1, 2, \ldots, u - 1\}$, where u is an exact power of 2, then a recursive data structure known as a ***van Emde Boas tree*** [440, 441] supports each of the operations SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg \lg u)$ time. ***Fusion trees*** [157] were the first data structure to allow faster dictionary operations when the universe is restricted to integers, implementing these operations in $O(\lg n / \lg \lg n)$ time. Several subsequent data structures, including ***exponential search trees*** [17], have also given improved bounds on some or all of

the dictionary operations and are mentioned in the chapter notes throughout this book.

- ***Dynamic graph data structures*** support various queries while allowing the structure of a graph to change through operations that insert or delete vertices or edges. Examples of the queries that they support include vertex connectivity [214], edge connectivity, minimum spanning trees [213], biconnectivity, and transitive closure [212].

Chapter notes throughout this book mention additional data structures.