The Algorithm Design Manual

Second Edition

Steven S. Skiena

The Algorithm Design Manual

Second Edition

Springer

Steven S. Skiena
Department of Computer Science
State University of New York
at Stony Brook
New York, USA
skiena@cs.sunysb.edu

ISBN: 978-1-84800-069-8 e-ISBN: 978-1-84800-070-4
DOI: 10.1007/978-1-84800-070-4

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2008931136

Printed on acid-free paper

Springer Science+Business Media
springer.com

Preface

Most professional programmers that I've encountered are not well prepared to tackle algorithm design problems. This is a pity, because the techniques of algorithm design form one of the core practical *technologies* of computer science. Designing correct, efficient, and implementable algorithms for real-world problems requires access to two distinct bodies of knowledge:

- *Techniques* – Good algorithm designers understand several fundamental algorithm design techniques, including data structures, dynamic programming, depth-first search, backtracking, and heuristics. Perhaps the single most important design technique is *modeling*, the art of abstracting a messy real-world application into a clean problem suitable for algorithmic attack.

- *Resources* – Good algorithm designers stand on the shoulders of giants. Rather than laboring from scratch to produce a new algorithm for every task, they can figure out what is known about a particular problem. Rather than re-implementing popular algorithms from scratch, they seek existing implementations to serve as a starting point. They are familiar with many classic algorithmic problems, which provide sufficient source material to model most any application.

This book is intended as a manual on algorithm design, providing access to combinatorial algorithm technology for both students and computer professionals. It is divided into two parts: *Techniques* and *Resources*. The former is a general guide to techniques for the design and analysis of computer algorithms. The Resources section is intended for browsing and reference, and comprises the catalog of algorithmic resources, implementations, and an extensive bibliography.

To the Reader

I have been gratified by the warm reception the first edition of *The Algorithm Design Manual* has received since its initial publication in 1997. It has been recognized as a unique guide to using algorithmic techniques to solve problems that often arise in practice. But much has changed in the world since the *The Algorithm Design Manual* was first published over ten years ago. Indeed, if we date the origins of modern algorithm design and analysis to about 1970, then roughly 30% of modern algorithmic history has happened since the first coming of *The Algorithm Design Manual*.

Three aspects of *The Algorithm Design Manual* have been particularly beloved: (1) the catalog of algorithmic problems, (2) the war stories, and (3) the electronic component of the book. These features have been preserved and strengthened in this edition:

- *The Catalog of Algorithmic Problems* – Since finding out what is known about an algorithmic problem can be a difficult task, I provide a catalog of the 75 most important problems arising in practice. By browsing through this catalog, the student or practitioner can quickly identify what their problem is called, what is known about it, and how they should proceed to solve it. To aid in problem identification, we include a pair of "before" and "after" pictures for each problem, illustrating the required input and output specifications. One perceptive reviewer called my book "The Hitchhiker's Guide to Algorithms" on the strength of this catalog.

  The catalog is *the* most important part of this book. To update the catalog for this edition, I have solicited feedback from the world's leading experts on each associated problem. Particular attention has been paid to updating the discussion of available software implementations for each problem.

- *War Stories* – In practice, algorithm problems do not arise at the beginning of a large project. Rather, they typically arise as subproblems when it becomes clear that the programmer does not know how to proceed or that the current solution is inadequate.

  To provide a better perspective on how algorithm problems arise in the real world, we include a collection of "war stories," or tales from our experience with real problems. The moral of these stories is that algorithm design and analysis is not just theory, but an important tool to be pulled out and used as needed.

  This edition retains all the original war stories (with updates as appropriate) plus additional new war stories covering external sorting, graph algorithms, simulated annealing, and other topics.

- *Electronic Component* – Since the practical person is usually looking for a program more than an algorithm, we provide pointers to solid implementations whenever they are available. We have collected these implementations

at one central website site (*http://www.cs.sunysb.edu/∼algorith*) for easy retrieval. We have been the number one "Algorithm" site on Google pretty much since the initial publication of the book.

Further, we provide recommendations to make it easier to identify the correct code for the job. With these implementations available, the critical issue in algorithm design becomes properly modeling your application, more so than becoming intimate with the details of the actual algorithm. This focus permeates the entire book.

Equally important is what we do not do in this book. We do not stress the mathematical analysis of algorithms, leaving most of the analysis as informal arguments. You will not find a single theorem anywhere in this book. When more details are needed, the reader should study the cited programs or references. The goal of this manual is to get you going in the right direction as quickly as possible.

To the Instructor

This book covers enough material for a standard *Introduction to Algorithms* course. We assume the reader has completed the equivalent of a second programming course, typically titled *Data Structures* or *Computer Science II*.

A full set of lecture slides for teaching this course is available online at *http://www.algorist.com* . Further, I make available online audio and video lectures using these slides to teach a full-semester algorithm course. Let me help teach your course, by the magic of the Internet!

This book stresses design over analysis. It is suitable for both traditional lecture courses and the new "active learning" method, where the professor does not lecture but instead guides student groups to solve real problems. The "war stories" provide an appropriate introduction to the active learning method.

I have made several pedagogical improvements throughout the book. Textbook-oriented features include:

- *More Leisurely Discussion* – The tutorial material in the first part of the book has been *doubled* over the previous edition. The pages have been devoted to more thorough and careful exposition of fundamental material, instead of adding more specialized topics.

- *False Starts* – Algorithms textbooks generally present important algorithms as a fait accompli, obscuring the ideas involved in designing them and the subtle reasons why other approaches fail. The war stories illustrate such development on certain applied problems, but I have expanded such coverage into classical algorithm design material as well.

- *Stop and Think* – Here I illustrate my thought process as I solve a topic-specific homework problem—false starts and all. I have interspersed such

problem blocks throughout the text to increase the problem-solving activity of my readers. Answers appear immediately following each problem.

- *More and Improved Homework Problems* – This edition of *The Algorithm Design Manual* has twice as many homework exercises as the previous one. Exercises that proved confusing or ambiguous have been improved or replaced. Degree of difficulty ratings (from 1 to 10) have been assigned to all problems.

- *Self-Motivating Exam Design* – In my algorithms courses, I promise the students that *all* midterm and final exam questions will be taken directly from homework problems in this book. This provides a "student-motivated exam," so students know exactly how to study to do well on the exam. I have carefully picked the quantity, variety, and difficulty of homework exercises to make this work; ensuring there are neither too few or too many candidate problems.

- *Take-Home Lessons* – Highlighted "take-home" lesson boxes scattered throughout the text emphasize the big-picture concepts to be gained from the chapter.

- *Links to Programming Challenge Problems* – Each chapter's exercises will contain links to 3-5 relevant "Programming Challenge" problems from *http://www.programming-challenges.com*. These can be used to add a programming component to paper-and-pencil algorithms courses.

- *More Code, Less Pseudo-code* – More algorithms in this book appear as code (written in C) instead of pseudo-code. I believe the concreteness and reliability of actual tested implementations provides a big win over less formal presentations for simple algorithms. Full implementations are available for study at *http://www.algorist.com* .

- *Chapter Notes* – Each tutorial chapter concludes with a brief notes section, pointing readers to primary sources and additional references.

Acknowledgments

Updating a book dedication after ten years focuses attention on the effects of time. Since the first edition, Renee has become my wife and then the mother of our two children, Bonnie and Abby. My father has left this world, but Mom and my brothers Len and Rob remain a vital presence in my life. I dedicate this book to my family, new and old, here and departed.

I would like to thank several people for their concrete contributions to this new edition. Andrew Gaun and Betson Thomas helped in many ways, particularly in building the infrastructure for the new *http://www.cs.sunysb.edu/∼algorith*and dealing with a variety of manuscript preparation issues. David Gries offered valuable feedback well beyond the call of duty. Himanshu Gupta and Bin Tang bravely

taught courses using a manuscript version of this edition. Thanks also to my Springer-Verlag editors, Wayne Wheeler and Allan Wylde.

A select group of algorithmic sages reviewed sections of the Hitchhiker's guide, sharing their wisdom and alerting me to new developments. Thanks to:

> Ami Amir, Herve Bronnimann, Bernard Chazelle, Chris Chu, Scott Cotton, Yefim Dinitz, Komei Fukuda, Michael Goodrich, Lenny Heath, Cihat Imamoglu, Tao Jiang, David Karger, Giuseppe Liotta, Albert Mao, Silvano Martello, Catherine McGeoch, Kurt Mehlhorn, Scott A. Mitchell, Naceur Meskini, Gene Myers, Gonzalo Navarro, Stephen North, Joe O'Rourke, Mike Paterson, Theo Pavlidis, Seth Pettie, Michel Pocchiola, Bart Preneel, Tomasz Radzik, Edward Reingold, Frank Ruskey, Peter Sanders, Joao Setubal, Jonathan Shewchuk, Robert Skeel, Jens Stoye, Torsten Suel, Bruce Watson, and Uri Zwick.

Several exercises were originated by colleagues or inspired by other texts. Reconstructing the original sources years later can be challenging, but credits for each problem (to the best of my recollection) appear on the website.

It would be rude not to thank important contributors to the original edition. Ricky Bradley and Dario Vlah built up the substantial infrastructure required for the WWW site in a logical and extensible manner. Zhong Li drew most of the catalog figures using xfig. Richard Crandall, Ron Danielson, Takis Metaxas, Dave Miller, Giri Narasimhan, and Joe Zachary all reviewed preliminary versions of the first edition; their thoughtful feedback helped to shape what you see here.

Much of what I know about algorithms I learned along with my graduate students. Several of them (Yaw-Ling Lin, Sundaram Gopalakrishnan, Ting Chen, Francine Evans, Harald Rau, Ricky Bradley, and Dimitris Margaritis) are the real heroes of the war stories related within. My Stony Brook friends and algorithm colleagues Estie Arkin, Michael Bender, Jie Gao, and Joe Mitchell have always been a pleasure to work and be with. Finally, thanks to Michael Brochstein and the rest of the city contingent for revealing a proper life well beyond Stony Brook.

Caveat

It is traditional for the author to magnanimously accept the blame for whatever deficiencies remain. I don't. Any errors, deficiencies, or problems in this book are somebody else's fault, but I would appreciate knowing about them so as to determine who is to blame.

Steven S. Skiena
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400
http://www.cs.sunysb.edu/∼skiena
April 2008

Contents

1

Introduction to Algorithm Design

What is an algorithm? An algorithm is a procedure to accomplish a specific task. An algorithm is the idea behind any reasonable computer program.

To be interesting, an algorithm must solve a general, well-specified *problem.* An algorithmic problem is specified by describing the complete set of *instances* it must work on and of its output after running on one of these instances. This distinction, between a problem and an instance of a problem, is fundamental. For example, the algorithmic *problem* known as *sorting* is defined as follows:

Problem: Sorting
Input: A sequence of n keys $a_1, \ldots, a_n$.
Output: The permutation (reordering) of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_{n-1} \leq a'_n$.

An *instance* of sorting might be an array of names, like {*Mike, Bob, Sally, Jill, Jan*}, or a list of numbers like {*154, 245, 568, 324, 654, 324*}. Determining that you are dealing with a general problem is your first step towards solving it.

An *algorithm* is a procedure that takes any of the possible input instances and transforms it to the desired output. There are many different algorithms for solving the problem of sorting. For example, *insertion sort* is a method for sorting that starts with a single element (thus forming a trivially sorted list) and then incrementally inserts the remaining elements so that the list stays sorted. This algorithm, implemented in C, is described below:

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_1,

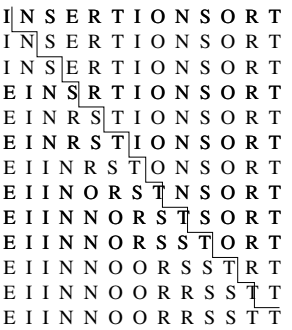

Figure 1.1: Animation of insertion sort in action (time flows down)

```
insertion_sort(item s[], int n)
{
        int i,j;                /* counters */

        for (i=1; i<n; i++) {
                j=i;
                while ((j>0) && (s[j] < s[j-1])) {
                        swap(&s[j],&s[j-1]);
                        j = j-1;
                }
        }
}
```

An animation of the logical flow of this algorithm on a particular instance (the letters in the word "INSERTIONSORT") is given in Figure 1.1

Note the generality of this algorithm. It works just as well on names as it does on numbers, given the appropriate comparison operation ($<$) to test which of the two keys should appear first in sorted order. It can be readily verified that this algorithm correctly orders every possible input instance according to our definition of the sorting problem.

There are three desirable properties for a good algorithm. We seek algorithms that are *correct* and *efficient*, while being *easy to implement*. These goals may not be simultaneously achievable. In industrial settings, any program that seems to give good enough answers without slowing the application down is often acceptable, regardless of whether a better algorithm exists. The issue of finding the best possible answer or achieving maximum efficiency usually arises in industry only after serious performance or legal troubles.

In this chapter, we will focus on the issues of algorithm correctness, and defer a discussion of efficiency concerns to Chapter 2. It is seldom obvious whether a given

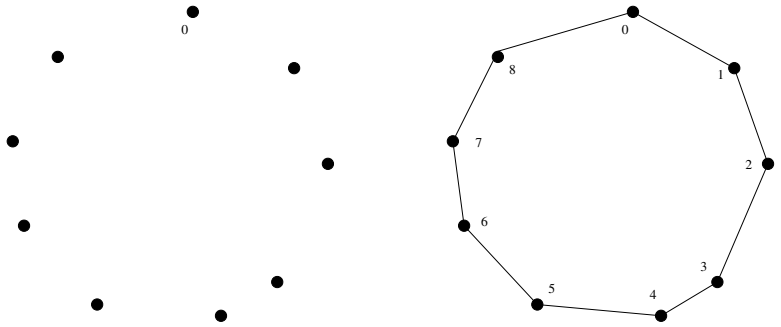

Figure 1.2: A good instance for the nearest-neighbor heuristic

algorithm correctly solves a given problem. Correct algorithms usually come with a proof of correctness, which is an explanation of *why* we know that the algorithm must take every instance of the problem to the desired result. However, before we go further we demonstrate why *"it's obvious"* never suffices as a proof of correctness, and is usually flat-out wrong.

1.1 Robot Tour Optimization

Let's consider a problem that arises often in manufacturing, transportation, and testing applications. Suppose we are given a robot arm equipped with a tool, say a soldering iron. In manufacturing circuit boards, all the chips and other components must be fastened onto the substrate. More specifically, each chip has a set of contact points (or wires) that must be soldered to the board. To program the robot arm for this job, we must first construct an ordering of the contact points so the robot visits (and solders) the first contact point, then the second point, third, and so forth until the job is done. The robot arm then proceeds back to the first contact point to prepare for the next board, thus turning the tool-path into a closed tour, or cycle.

Robots are expensive devices, so we want the tour that minimizes the time it takes to assemble the circuit board. A reasonable assumption is that the robot arm moves with fixed speed, so the time to travel between two points is proportional to their distance. In short, we must solve the following algorithm problem:

Problem: Robot Tour Optimization
Input: A set S of n points in the plane.
Output: What is the shortest cycle tour that visits each point in the set S?

You are given the job of programming the robot arm. Stop right now and think up an algorithm to solve this problem. I'll be happy to wait until you find one...

Several algorithms might come to mind to solve this problem. Perhaps the most popular idea is the *nearest-neighbor* heuristic. Starting from some point p_0, we walk first to its nearest neighbor p_1. From p_1, we walk to its nearest unvisited neighbor, thus excluding only p_0 as a candidate. We now repeat this process until we run out of unvisited points, after which we return to p_0 to close off the tour. Written in pseudo-code, the nearest-neighbor heuristic looks like this:

```
NearestNeighbor(P)
    Pick and visit an initial point p_0 from P
    p = p_0
    i = 0
    While there are still unvisited points
        i = i + 1
        Select p_i to be the closest unvisited point to p_{i-1}
        Visit p_i
    Return to p_0 from p_{n-1}
```

This algorithm has a lot to recommend it. It is simple to understand and implement. It makes sense to visit nearby points before we visit faraway points to reduce the total travel time. The algorithm works perfectly on the example in Figure 1.2. The nearest-neighbor rule is reasonably efficient, for it looks at each pair of points (p_i, p_j) at most twice: once when adding p_i to the tour, the other when adding p_j. Against all these positives there is only one problem. This algorithm is completely wrong.

Wrong? How can it be wrong? The algorithm always finds a tour, but it doesn't necessarily find the shortest possible tour. It doesn't necessarily even come close. Consider the set of points in Figure 1.3, all of which lie spaced along a line. The numbers describe the distance that each point lies to the left or right of the point labeled '0'. When we start from the point '0' and repeatedly walk to the nearest unvisited neighbor, we might keep jumping left-right-left-right over '0' as the algorithm offers no advice on how to break ties. A much better (indeed optimal) tour for these points starts from the leftmost point and visits each point as we walk right before returning at the rightmost point.

Try now to imagine your boss's delight as she watches a demo of your robot arm hopscotching left-right-left-right during the assembly of such a simple board.

"But wait," you might be saying. "The problem was in starting at point '0'. Instead, why don't we start the nearest-neighbor rule using the leftmost point as the initial point p_0? By doing this, we will find the optimal solution on this instance."

That is 100% true, at least until we rotate our example 90 degrees. Now all points are equally leftmost. If the point '0' were moved just slightly to the left, it would be picked as the starting point. Now the robot arm will hopscotch up-down-up-down instead of left-right-left-right, but the travel time will be just as bad as before. No matter what you do to pick the first point, the nearest-neighbor rule is doomed to work incorrectly on certain point sets.

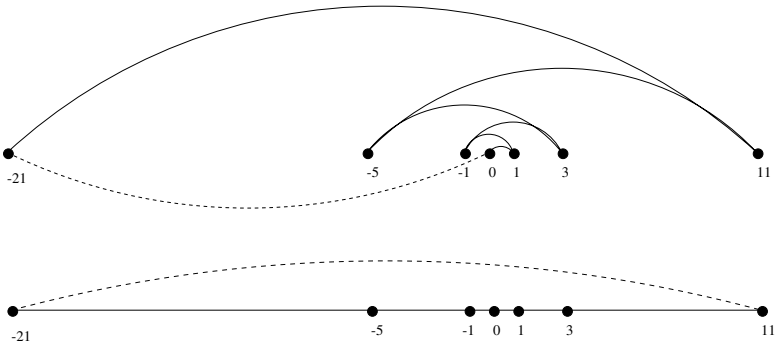

Figure 1.3: A bad instance for the nearest-neighbor heuristic, with the optimal solution

Maybe what we need is a different approach. Always walking to the closest point is too restrictive, since it seems to trap us into making moves we didn't want. A different idea might be to repeatedly connect the closest pair of endpoints whose connection will not create a problem, such as premature termination of the cycle. Each vertex begins as its own single vertex chain. After merging everything together, we will end up with a single chain containing all the points in it. Connecting the final two endpoints gives us a cycle. At any step during the execution of this *closest-pair heuristic*, we will have a set of single vertices and vertex-disjoint chains available to merge. In pseudocode:

ClosestPair(P)
    Let n be the number of points in set P.
    For $i = 1$ to $n - 1$ do
        $d = \infty$
        For each pair of endpoints (s, t) from distinct vertex chains
            if $dist(s, t) \leq d$ then $s_m = s$, $t_m = t$, and $d = dist(s, t)$
        Connect (s_m, t_m) by an edge
    Connect the two endpoints by an edge

This closest-pair rule does the right thing in the example in Figure 1.3. It starts by connecting '0' to its immediate neighbors, the points 1 and -1. Subsequently, the next closest pair will alternate left-right, growing the central path by one link at a time. The closest-pair heuristic is somewhat more complicated and less efficient than the previous one, but at least it gives the right answer in this example.

But this is not true in all examples. Consider what this algorithm does on the point set in Figure 1.4(l). It consists of two rows of equally spaced points, with the rows slightly closer together (distance $1 - e$) than the neighboring points are spaced within each row (distance $1 + e$). Thus the closest pairs of points stretch across the gap, not around the boundary. After we pair off these points, the closest

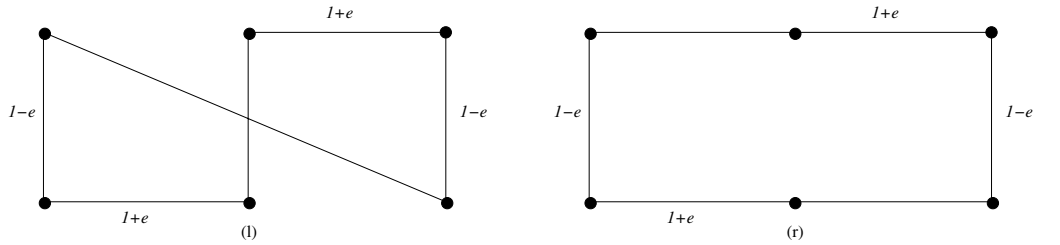

Figure 1.4: A bad instance for the closest-pair heuristic, with the optimal solution

remaining pairs will connect these pairs alternately around the boundary. The total path length of the closest-pair tour is $3(1-e)+2(1+e)+\sqrt{(1-e)^2+(2+2e)^2}$. Compared to the tour shown in Figure 1.4(r), we travel over 20% farther than necessary when $e \approx 0$. Examples exist where the penalty is considerably worse than this.

Thus this second algorithm is also wrong. Which one of these algorithms performs better? You can't tell just by looking at them. Clearly, both heuristics can end up with very bad tours on very innocent-looking input.

At this point, you might wonder what a correct algorithm for our problem looks like. Well, we could try enumerating *all* possible orderings of the set of points, and then select the ordering that minimizes the total length:

```
OptimalTSP(P)
    d = ∞
    For each of the n! permutations P_i of point set P
        If (cost(P_i) ≤ d) then d = cost(P_i) and P_min = P_i
    Return P_min
```

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour. This algorithm is correct, since we pick the best of all the possibilities. But it is also extremely slow. The fastest computer in the world couldn't hope to enumerate all the 20! =2,432,902,008,176,640,000 orderings of 20 points within a day. For real circuit boards, where $n \approx 1,000$, forget about it. All of the world's computers working full time wouldn't come close to finishing the problem before the end of the universe, at which point it presumably becomes moot.

The quest for an efficient algorithm to solve this problem, called the *traveling salesman problem* (TSP), will take us through much of this book. If you need to know how the story ends, check out the catalog entry for the traveling salesman problem in Section 16.4 (page 533).

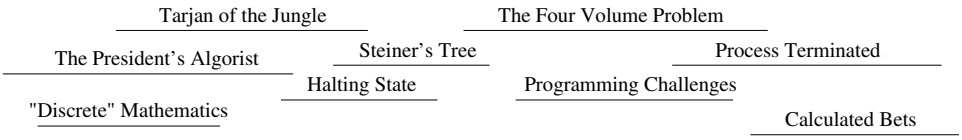

Figure 1.5: An instance of the non-overlapping movie scheduling problem

Take-Home Lesson: There is a fundamental difference between *algorithms*, which always produce a correct result, and *heuristics*, which may usually do a good job but without providing any guarantee.

1.2 Selecting the Right Jobs

Now consider the following scheduling problem. Imagine you are a highly-in-demand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming. To take the job, you must commit to being available throughout this entire period. Thus you cannot simultaneously accept two jobs whose intervals overlap.

For an artist such as yourself, the criteria for job acceptance is clear: you want to make as much money as possible. Because each of these films pays the same fee per film, this implies you seek the largest possible set of jobs (intervals) such that no two of them conflict with each other.

For example, consider the available projects in Figure 1.5. We can star in at most four films, namely *"Discrete" Mathematics*, *Programming Challenges*, *Calculated Bets*, and one of either *Halting State* or *Steiner's Tree*.

You (or your agent) must solve the following algorithmic scheduling problem:

Problem: Movie Scheduling Problem
Input: A set I of n intervals on the line.
Output: What is the largest subset of mutually non-overlapping intervals which can be selected from I?

You are given the job of developing a scheduling algorithm for this task. Stop right now and try to find one. Again, I'll be happy to wait.

There are several ideas that may come to mind. One is based on the notion that it is best to work whenever work is available. This implies that you should start with the job with the earliest start date – after all, there is no other job you can work on, then at least during the begining of this period.

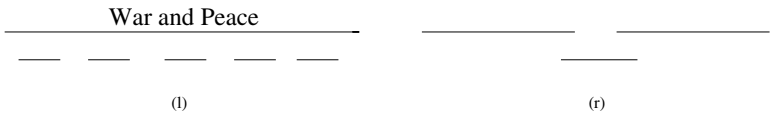

Figure 1.6: Bad instances for the (l) earliest job first and (r) shortest job first heuristics.

EarliestJobFirst(I)

Accept the earlest starting job j from I which does not overlap any previously accepted job, and repeat until no more such jobs remain.

This idea makes sense, at least until we realize that accepting the earliest job might block us from taking many other jobs if that first job is long. Check out Figure 1.6(l), where the epic "War and Peace" is both the first job available and long enough to kill off all other prospects.

This bad example naturally suggests another idea. The problem with "War and Peace" is that it is too long. Perhaps we should start by taking the shortest job, and keep seeking the shortest available job at every turn. Maximizing the number of jobs we do in a given period is clearly connected to banging them out as quickly as possible. This yields the heuristic:

ShortestJobFirst(I)

While ($I \neq \emptyset$) do

Accept the shortest possible job j from I.

Delete j, and any interval which intersects j from I.

Again this idea makes sense, at least until we realize that accepting the shortest job might block us from taking two other jobs, as shown in Figure 1.6(r). While the potential loss here seems smaller than with the previous heuristic, it can readily limit us to half the optimal payoff.

At this point, an algorithm where we try all possibilities may start to look good, because we can be certain it is correct. If we ignore the details of testing whether a set of intervals are in fact disjoint, it looks something like this:

ExhaustiveScheduling(I)

$j = 0$

$S_{max} = \emptyset$

For each of the 2^n subsets S_i of intervals I

If (S_i is mutally non-overlapping) and ($size(S_i) > j$)

then $j = size(S_i)$ and $S_{max} = S_i$.

Return S_{max}

But how slow is it? The key limitation is enumerating the 2^n subsets of n things. The good news is that this is *much* better than enumerating all $n!$ orders

of n things, as proposed for the robot tour optimization problem. There are only about one million subsets when $n = 20$, which could be exhaustively counted within seconds on a decent computer. However, when fed $n = 100$ movies, 2^{100} is much much greater than the 20! which made our robot cry "uncle" in the previous problem.

The difference between our scheduling and robotics problems are that there *is* an algorithm which solves movie scheduling both correctly and efficiently. Think about the first job to terminate—i.e. the interval x which contains the rightmost point which is leftmost among all intervals. This role is played by *"Discrete" Mathematics* in Figure 1.5. Other jobs may well have started before x, but all of these must at least partially overlap each other, so we can select at most one from the group. The first of these jobs to terminate is x, so any of the overlapping jobs potentially block out other opportunities to the right of it. Clearly we can never lose by picking x. This suggests the following correct, efficient algorithm:

```
OptimalScheduling(I)
    While (I ≠ ∅) do
        Accept the job j from I with the earliest completion date.
        Delete j, and any interval which intersects j from I.
```

Ensuring the optimal answer over all possible inputs is a difficult but often achievable goal. Seeking counterexamples that break pretender algorithms is an important part of the algorithm design process. Efficient algorithms are often lurking out there; this book seeks to develop your skills to help you find them.

> *Take-Home Lesson:* Reasonable-looking algorithms can easily be incorrect. Algorithm correctness is a property that must be carefully demonstrated.

1.3 Reasoning about Correctness

Hopefully, the previous examples have opened your eyes to the subtleties of algorithm correctness. We need tools to distinguish correct algorithms from incorrect ones, the primary one of which is called a *proof*.

A proper mathematical proof consists of several parts. First, there is a clear, precise statement of what you are trying to prove. Second, there is a set of assumptions of things which are taken to be true and hence used as part of the proof. Third, there is a chain of reasoning which takes you from these assumptions to the statement you are trying to prove. Finally, there is a little square (∎) or *QED* at the bottom to denote that you have finished, representing the Latin phrase for "thus it is demonstrated."

This book is not going to emphasize formal proofs of correctness, because they are very difficult to do right and quite misleading when you do them wrong. A proof is indeed a *demonstration*. Proofs are useful only when they are honest; crisp arguments explaining why an algorithm satisfies a nontrivial correctness property.

Correct algorithms require careful exposition, and efforts to show both correctness and *not incorrectness*. We develop tools for doing so in the subsections below.

1.3.1 Expressing Algorithms

Reasoning about an algorithm is impossible without a careful description of the sequence of steps to be performed. The three most common forms of algorithmic notation are (1) English, (2) pseudocode, or (3) a real programming language. We will use all three in this book. Pseudocode is perhaps the most mysterious of the bunch, but it is best defined as a programming language that never complains about syntax errors. All three methods are useful because there is a natural tradeoff between greater ease of expression and precision. English is the most natural but least precise programming language, while Java and C/C++ are precise but difficult to write and understand. Pseudocode is generally useful because it represents a happy medium.

The choice of which notation is best depends upon which method you are most comfortable with. I usually prefer to describe the *ideas* of an algorithm in English, moving to a more formal, programming-language-like pseudocode or even real code to clarify sufficiently tricky details.

A common mistake my students make is to use pseudocode to dress up an ill-defined idea so that it looks more formal. Clarity should be the goal. For example, the `ExhaustiveScheduling` algorithm on page 10 could have been better written in English as:

ExhaustiveScheduling(I)
Test all 2^n subsets of intervals from I, and return the largest subset consisting of mutually non-overlapping intervals.

Take-Home Lesson: The heart of any algorithm is an *idea*. If your idea is not clearly revealed when you express an algorithm, then you are using too low-level a notation to describe it.

1.3.2 Problems and Properties

We need more than just an algorithm description in order to demonstrate correctness. We also need a careful description of the problem that it is intended to solve.

Problem specifications have two parts: (1) the set of allowed input instances, and (2) the required properties of the algorithm's output. It is impossible to prove the correctness of an algorithm for a fuzzily-stated problem. Put another way, ask the wrong problem and you will get the wrong answer.

Some problem specifications allow too broad a class of input instances. Suppose we had allowed film projects in our movie scheduling problem to have gaps in

production (i.e. , filming in September and November but a hiatus in October). Then the schedule associated with any particular film would consist of a given *set* of intervals. Our star would be free to take on two interleaving but not overlapping projects (such as the film above nested with one filming in August and October). The earliest completion algorithm would not work for such a generalized scheduling problem. Indeed, *no* efficient algorithm exists for this generalized problem.

> *Take-Home Lesson:* An important and honorable technique in algorithm design is to narrow the set of allowable instances until there *is* a correct and efficient algorithm. For example, we can restrict a graph problem from general graphs down to trees, or a geometric problem from two dimensions down to one.

There are two common traps in specifying the output requirements of a problem. One is asking an ill-defined question. Asking for the *best* route between two places on a map is a silly question unless you define what *best* means. Do you mean the shortest route in total distance, or the fastest route, or the one minimizing the number of turns?

The second trap is creating compound goals. The three path-planning criteria mentioned above are all well-defined goals that lead to correct, efficient optimization algorithms. However, you must pick a single criteria. A goal like *Find the shortest path from a to b that doesn't use more than twice as many turns as necessary* is perfectly well defined, but complicated to reason and solve.

I encourage you to check out the problem statements for each of the 75 catalog problems in the second part of this book. Finding the right formulation for your problem is an important part of solving it. And studying the definition of all these classic algorithm problems will help you recognize when someone else has thought about similar problems before you.

1.3.3 Demonstrating Incorrectness

The best way to prove that an algorithm is *incorrect* is to produce an instance in which it yields an incorrect answer. Such instances are called *counter-examples*. No rational person will ever leap to the defense of an algorithm after a counter-example has been identified. Very simple instances can instantly kill reasonable-looking heuristics with a quick *touché*. Good counter-examples have two important properties:

- *Verifiability* – To demonstrate that a particular instance is a counter-example to a particular algorithm, you must be able to (1) calculate what answer your algorithm will give in this instance, and (2) display a better answer so as to prove the algorithm didn't find it.

  Since you must hold the given instance in your head to reason about it, an important part of verifiability is...

- *Simplicity* – Good counter-examples have all unnecessary details boiled away. They make clear exactly *why* the proposed algorithm fails. Once a counter-example has been found, it is worth simplifying it down to its essence. For example, the counter-example of Figure 1.6(l) could be made simpler and better by reducing the number of overlapped segments from four to two.

Hunting for counter-examples is a skill worth developing. It bears some similarity to the task of developing test sets for computer programs, but relies more on inspiration than exhaustion. Here are some techniques to aid your quest:

- *Think small* – Note that the robot tour counter-examples I presented boiled down to six points or less, and the scheduling counter-examples to only three intervals. This is indicative of the fact that when algorithms fail, there is usually a very simple example on which they fail. Amateur algorists tend to draw a big messy instance and then stare at it helplessly. The pros look carefully at several small examples, because they are easier to verify and reason about.

- *Think exhaustively* – There are only a small number of possibilities for the smallest nontrivial value of n. For example, there are only three interesting ways two intervals on the line can occur: (1) as disjoint intervals, (2) as overlapping intervals, and (3) as properly nesting intervals, one within the other. All cases of three intervals (including counter-examples to both movie heuristics) can be systematically constructed by adding a third segment in each possible way to these three instances.

- *Hunt for the weakness* – If a proposed algorithm is of the form "always take the biggest" (better known as the *greedy algorithm*), think about why that might prove to be the wrong thing to do. In particular, ...

- *Go for a tie* – A devious way to break a greedy heuristic is to provide instances where everything is the same size. Suddenly the heuristic has nothing to base its decision on, and perhaps has the freedom to return something suboptimal as the answer.

- *Seek extremes* – Many counter-examples are mixtures of huge and tiny, left and right, few and many, near and far. It is usually easier to verify or reason about extreme examples than more muddled ones. Consider two tightly bunched clouds of points separated by a much larger distance d. The optimal TSP tour will be essentially $2d$ regardless of the number of points, because what happens within each cloud doesn't really matter.

Take-Home Lesson: Searching for counterexamples is the best way to disprove the correctness of a heuristic.

1.3.4 Induction and Recursion

Failure to find a counterexample to a given algorithm does not mean "it is obvious" that the algorithm is correct. A proof or demonstration of correctness is needed. Often mathematical induction is the method of choice.

When I first learned about mathematical induction it seemed like complete magic. You proved a formula like $\sum_{i=1}^{n} i = n(n+1)/2$ for some basis case like 1 or 2, then *assumed* it was true all the way to $n-1$ before proving it was true for general n using the assumption. That was a proof? Ridiculous!

When I first learned the programming technique of recursion it also seemed like complete magic. The program tested whether the input argument was some basis case like 1 or 2. If not, you solved the bigger case by breaking it into pieces and *calling the subprogram itself* to solve these pieces. That was a program? Ridiculous!

The reason both seemed like magic is because recursion *is* mathematical induction. In both, we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces. The *initial* or boundary condition terminates the recursion. Once you understand either recursion or induction, you should be able to see why the other one also works.

I've heard it said that a computer scientist is a mathematician who only knows how to prove things by induction. This is partially true because computer scientists are lousy at proving things, but primarily because so many of the algorithms we study are either recursive or incremental.

Consider the correctness of *insertion sort*, which we introduced at the beginning of this chapter. The *reason* it is correct can be shown inductively:

- The basis case consists of a single element, and by definition a one-element array is completely sorted.
- In general, we can assume that the first $n-1$ elements of array A are completely sorted after $n-1$ iterations of insertion sort.
- To insert one last element x to A, we find where it goes, namely the unique spot between the biggest element less than or equal to x and the smallest element greater than x. This is done by moving all the greater elements back by one position, creating room for x in the desired location. ∎

One must be suspicious of inductive proofs, however, because very subtle reasoning errors can creep in. The first are *boundary errors*. For example, our insertion sort correctness proof above boldly stated that there was a unique place to insert x between two elements, when our basis case was a single-element array. Greater care is needed to properly deal with the special cases of inserting the minimum or maximum elements.

The second and more common class of inductive proof errors concerns cavallier extension claims. Adding one extra item to a given problem instance might cause the entire optimal solution to change. This was the case in our scheduling problem (see Figure 1.7). The optimal schedule after inserting a new segment may contain

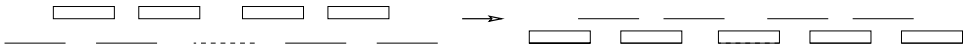

Figure 1.7: Large-scale changes in the optimal solution (boxes) after inserting a single interval (dashed) into the instance

none of the segments of any particular optimal solution prior to insertion. Boldly ignoring such difficulties can lead to very convincing inductive proofs of incorrect algorithms.

Take-Home Lesson: Mathematical induction is usually the right way to verify the correctness of a recursive or incremental insertion algorithm.

Stop and Think: Incremental Correctness

Problem: Prove the correctness of the following recursive algorithm for incrementing natural numbers, i.e. $y \rightarrow y + 1$:

```
Increment(y)
    if y = 0 then return(1) else
        if (y mod 2) = 1 then
            return(2 · Increment(⌊y/2⌋))
        else return(y + 1)
```

Solution: The correctness of this algorithm is certainly *not* obvious to me. But as it is recursive and I am a computer scientist, my natural instinct is to try to prove it by induction.

The basis case of $y = 0$ is obviously correctly handled. Clearly the value 1 is returned, and $0 + 1 = 1$.

Now assume the function works correctly for the general case of $y = n-1$. Given this, we must demonstrate the truth for the case of $y = n$. Half of the cases are easy, namely the even numbers (For which $(y \bmod 2) = 0$), since $y + 1$ is explicitly returned.

For the odd numbers, the answer depends upon what is returned by $Increment(\lfloor y/2 \rfloor)$. Here we want to use our inductive assumption, but it isn't quite right. We have assumed that `increment` worked correctly for $y = n - 1$, but not for a value which is about half of it. We can fix this problem by strengthening our assumption to declare that the general case holds for all $y \leq n-1$. This costs us nothing in principle, but is necessary to establish the correctness of the algorithm.

Now, the case of odd y (i.e. $y = 2m + 1$ for some integer m) can be dealt with as:

$$\begin{aligned} 2 \cdot Increment(\lfloor (2m+1)/2 \rfloor) &= 2 \cdot Increment(\lfloor m + 1/2 \rfloor) \\ &= 2 \cdot Increment(m) \\ &= 2(m+1) \\ &= 2m + 2 = y + 1 \end{aligned}$$

and the general case is resolved. ∎

1.3.5 Summations

Mathematical summation formulae arise often in algorithm analysis, which we will study in Chapter 2. Further, proving the correctness of summation formulae is a classic application of induction. Several exercises on inductive proofs of summations appear as exercises at the end this chapter. To make these more accessible, I review the basics of summations here.

Summation formula are concise expressions describing the addition of an arbitrarily large set of numbers, in particular the formula

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n)$$

There are simple closed forms for summations of many algebraic functions. For example, since n ones is n,

$$\sum_{i=1}^{n} 1 = n$$

The sum of the first n integers can be seen by pairing up the ith and $(n - i + 1)$th integers:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n/2} (i + (n - i + 1)) = n(n+1)/2$$

Recognizing two basic classes of summation formulae will get you a long way in algorithm analysis:

- *Arithmetic progressions* – We already encountered arithmetic progressions when we saw $S(n) = \sum_{i}^{n} i = n(n+1)/2$ in the analysis of selection sort. From the big picture perspective, the important thing is that the sum is quadratic, not that the constant is 1/2. In general,

$$S(n, p) = \sum_{i}^{n} i^p = \Theta(n^{p+1})$$

for $p \geq 1$. Thus the sum of squares is cubic, and the sum of cubes is quartic (if you use such a word). The "big Theta" notation ($\Theta(x)$) will be properly explained in Section 2.2.

For $p < -1$, this sum always converges to a constant, even as $n \to \infty$. The interesting case is between results in ...

- *Geometric series* – In geometric progressions, the index of the loop effects the exponent, i.e.

$$G(n, a) = \sum_{i=0}^{n} a^i = a(a^{n+1} - 1)/(a - 1)$$

How we interpret this sum depends upon the *base* of the progression, i.e. a. When $a < 1$, this converges to a constant even as $n \to \infty$.

This series convergence proves to be the great "free lunch" of algorithm analysis. It means that the sum of a linear number of things can be constant, not linear. For example, $1 + 1/2 + 1/4 + 1/8 + \ldots \leq 2$ no matter how many terms we add up.

When $a > 1$, the sum grows rapidly with each new term, as in $1 + 2 + 4 + 8 + 16 + 32 = 63$. Indeed, $G(n, a) = \Theta(a^{n+1})$ for $a > 1$.

Stop and Think: Factorial Formulae

Problem: Prove that $\sum_{i=1}^{n} i \times i! = (n + 1)! - 1$ by induction.

Solution: The inductive paradigm is straightforward. First verify the basis case (here we do $n = 1$, although $n = 0$ would be even more general):

$$\sum_{i=1}^{1} i \times i! = 1 = (1 + 1)! - 1 = 2 - 1 = 1$$

Now assume the statement is true up to n. To prove the general case of $n + 1$, observe that rolling out the largest term

$$\sum_{i=1}^{n+1} i \times i! = (n + 1) \times (n + 1)! + \sum_{i=1}^{n} i \times i!$$

reveals the left side of our inductive assumption. Substituting the right side gives us

$$\sum_{i=1}^{n+1} i \times i! \quad = \quad (n + 1) \times (n + 1)! + (n + 1)! - 1$$

$$
\begin{aligned}
&= (n+1)! \times ((n+1)+1) - 1 \\
&= (n+2)! - 1
\end{aligned}
$$

This general trick of separating out the largest term from the summation to reveal an instance of the inductive assumption lies at the heart of all such proofs. ∎

1.4 Modeling the Problem

Modeling is the art of formulating your application in terms of precisely described, well-understood problems. Proper modeling is the key to applying algorithmic design techniques to real-world problems. Indeed, proper modeling can eliminate the need to design or even implement algorithms, by relating your application to what has been done before. Proper modeling is the key to effectively using the "Hitchhiker's Guide" in Part II of this book.

Real-world applications involve real-world objects. You might be working on a system to route traffic in a network, to find the best way to schedule classrooms in a university, or to search for patterns in a corporate database. Most algorithms, however, are designed to work on rigorously defined *abstract* structures such as permutations, graphs, and sets. To exploit the algorithms literature, you must learn to describe your problem abstractly, in terms of procedures on fundamental structures.

1.4.1 Combinatorial Objects

Odds are very good that others have stumbled upon your algorithmic problem before you, perhaps in substantially different contexts. But to find out what is known about your particular "widget optimization problem," you can't hope to look in a book under *widget*. You must formulate widget optimization in terms of computing properties of common structures such as:

- *Permutations* – which are arrangements, or orderings, of items. For example, $\{1, 4, 3, 2\}$ and $\{4, 3, 2, 1\}$ are two distinct permutations of the same set of four integers. We have already seen permutations in the robot optimization problem, and in sorting. Permutations are likely the object in question whenever your problem seeks an "arrangement," "tour," "ordering," or "sequence."

- *Subsets* – which represent selections from a set of items. For example, $\{1, 3, 4\}$ and $\{2\}$ are two distinct subsets of the first four integers. Order does not matter in subsets the way it does with permutations, so the subsets $\{1, 3, 4\}$ and $\{4, 3, 1\}$ would be considered identical. We saw subsets arise in the movie scheduling problem. Subsets are likely the object in question whenever your problem seeks a "cluster," "collection," "committee," "group," "packaging," or "selection."

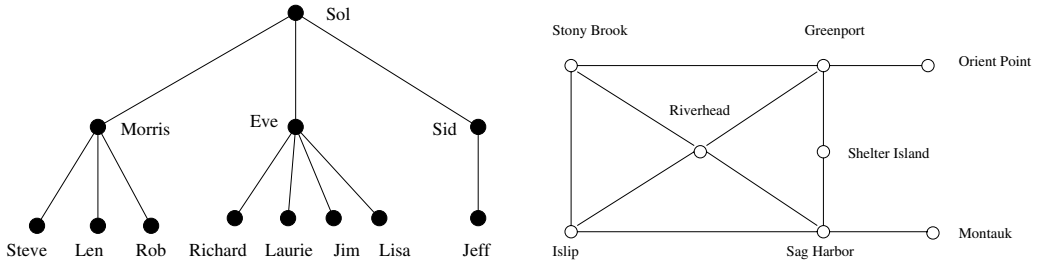

Figure 1.8: Modeling real-world structures with trees and graphs

- *Trees* – which represent hierarchical relationships between items. Figure 1.8(a) shows part of the family tree of the Skiena clan. Trees are likely the object in question whenever your problem seeks a "hierarchy," "dominance relationship," "ancestor/descendant relationship," or "taxonomy."

- *Graphs* – which represent relationships between arbitrary pairs of objects. Figure 1.8(b) models a network of roads as a graph, where the vertices are cities and the edges are roads connecting pairs of cities. Graphs are likely the object in question whenever you seek a "network," "circuit," "web," or "relationship."

- *Points* – which represent locations in some geometric space. For example, the locations of McDonald's restaurants can be described by points on a map/plane. Points are likely the object in question whenever your problems work on "sites," "positions," "data records," or "locations."

- *Polygons* – which represent regions in some geometric spaces. For example, the borders of a country can be described by a polygon on a map/plane. Polygons and polyhedra are likely the object in question whenever you are working on "shapes," "regions," "configurations," or "boundaries."

- *Strings* – which represent sequences of characters or patterns. For example, the names of students in a class can be represented by strings. Strings are likely the object in question whenever you are dealing with "text," "characters," "patterns," or "labels."

These fundamental structures all have associated algorithm problems, which are presented in the catalog of Part II. Familiarity with these problems is important, because they provide the language we use to model applications. To become fluent in this vocabulary, browse through the catalog and study the *input* and *output* pictures for each problem. Understanding these problems, even at a cartoon/definition level, will enable you to know where to look later when the problem arises in your application.

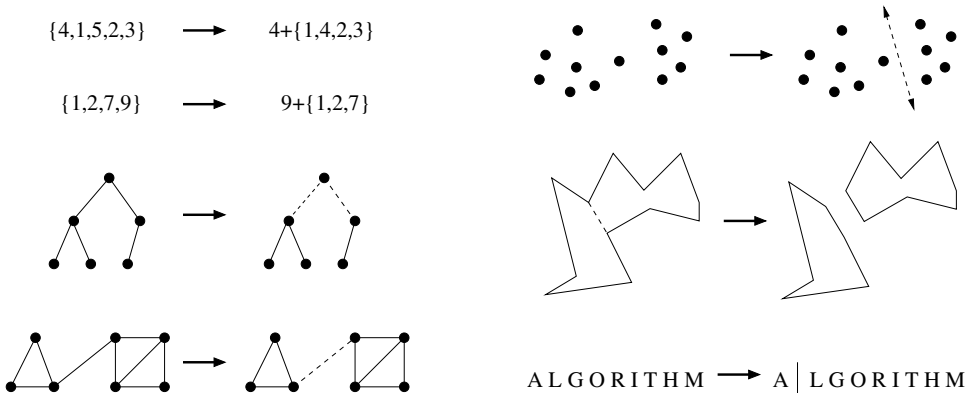

Figure 1.9: Recursive decompositions of combinatorial objects. (left column) Permutations, subsets, trees, and graphs. (right column) Point sets, polygons, and strings

Examples of successful application modeling will be presented in the war stories spaced throughout this book. However, some words of caution are in order. The act of modeling reduces your application to one of a small number of existing problems and structures. Such a process is inherently constraining, and certain details might not fit easily into the given target problem. Also, certain problems can be modeled in several different ways, some much better than others.

Modeling is only the first step in designing an algorithm for a problem. Be alert for how the details of your applications differ from a candidate model, but don't be too quick to say that your problem is unique and special. Temporarily ignoring details that don't fit can free the mind to ask whether they really were fundamental in the first place.

Take-Home Lesson: Modeling your application in terms of well-defined structures and algorithms is the most important single step towards a solution.

1.4.2 Recursive Objects

Learning to think recursively is learning to look for big things that are made from smaller things of *exactly the same type as the big thing*. If you think of houses as sets of rooms, then adding or deleting a room still leaves a house behind.

Recursive structures occur everywhere in the algorithmic world. Indeed, each of the abstract structures described above can be thought about recursively. You just have to see how you can break them down, as shown in Figure 1.9:

- *Permutations* – Delete the first element of a permutation of $\{1, \ldots, n\}$ things and you get a permutation of the remaining $n - 1$ things. Permutations are recursive objects.

- *Subsets* – Every subset of the elements $\{1, \ldots, n\}$ contains a subset of $\{1, \ldots, n-1\}$ made visible by deleting element n if it is present. Subsets are recursive objects.
- *Trees* – Delete the root of a tree and what do you get? A collection of smaller trees. Delete any leaf of a tree and what do you get? A slightly smaller tree. Trees are recursive objects.
- *Graphs* – Delete any vertex from a graph, and you get a smaller graph. Now divide the vertices of a graph into two groups, left and right. Cut through all edges which span from left to right, and what do you get? Two smaller graphs, and a bunch of broken edges. Graphs are recursive objects.
- *Points* – Take a cloud of points, and separate them into two groups by drawing a line. Now you have two smaller clouds of points. Point sets are recursive objects.
- *Polygons* – Inserting any internal chord between two nonadjacent vertices of a simple polygon on n vertices cuts it into two smaller polygons. Polygons are recursive objects.
- *Strings* – Delete the first character from a string, and what do you get? A shorter string. Strings are recursive objects.

Recursive descriptions of objects require both decomposition rules and *basis cases*, namely the specification of the smallest and simplest objects where the decomposition stops. These basis cases are usually easily defined. Permutations and subsets of zero things presumably look like {}. The smallest interesting tree or graph consists of a single vertex, while the smallest interesting point cloud consists of a single point. Polygons are a little trickier; the smallest genuine simple polygon is a triangle. Finally, the empty string has zero characters in it. The decision of whether the basis case contains zero or one element is more a question of taste and convenience than any fundamental principle.

Such recursive decompositions will come to define many of the algorithms we will see in this book. Keep your eyes open for them.

1.5 About the War Stories

The best way to learn how careful algorithm design can have a huge impact on performance is to look at real-world case studies. By carefully studying other people's experiences, we learn how they might apply to our work.

Scattered throughout this text are several of my own algorithmic war stories, presenting our successful (and occasionally unsuccessful) algorithm design efforts on real applications. I hope that you will be able to internalize these experiences so that they will serve as models for your own attacks on problems.

Every one of the war stories is true. Of course, the stories improve somewhat in the retelling, and the dialogue has been punched up to make them more interesting to read. However, I have tried to honestly trace the process of going from a raw problem to a solution, so you can watch how this process unfolded.

The *Oxford English Dictionary* defines an *algorist* as "one skillful in reckonings or figuring." In these stories, I have tried to capture some of the mindset of the algorist in action as they attack a problem.

The various war stories usually involve at least one, and often several, problems from the problem catalog in Part II. I reference the appropriate section of the catalog when such a problem occurs. This emphasizes the benefits of modeling your application in terms of standard algorithm problems. By using the catalog, you will be able to pull out what is known about any given problem whenever it is needed.

1.6 War Story: Psychic Modeling

The call came for me out of the blue as I sat in my office.

"Professor Skiena, I hope you can help me. I'm the President of Lotto Systems Group Inc., and we need an algorithm for a problem arising in our latest product."

"Sure," I replied. After all, the dean of my engineering school is always encouraging our faculty to interact more with industry.

"At Lotto Systems Group, we market a program designed to improve our customers' psychic ability to predict winning lottery numbers.[1] In a standard lottery, each ticket consists of six numbers selected from, say, 1 to 44. Thus, any given ticket has only a very small chance of winning. However, after proper training, our clients can visualize, say, 15 numbers out of the 44 and be certain that at least four of them will be on the winning ticket. Are you with me so far?"

"Probably not," I replied. But then I recalled how my dean encourages us to interact with industry.

"Our problem is this. After the psychic has narrowed the choices down to 15 numbers and is certain that at least 4 of them will be on the winning ticket, we must find the most efficient way to exploit this information. Suppose a cash prize is awarded whenever you pick at least three of the correct numbers on your ticket. We need an algorithm to construct the smallest set of tickets that we must buy in order to guarantee that we win at least one prize."

"Assuming the psychic is correct?"

"Yes, assuming the psychic is correct. We need a program that prints out a list of all the tickets that the psychic should buy in order to minimize their investment. Can you help us?"

Maybe they did have psychic ability, for they had come to the right place. Identifying the best subset of tickets to buy was very much a combinatorial algorithm

[1] Yes, this is a true story.

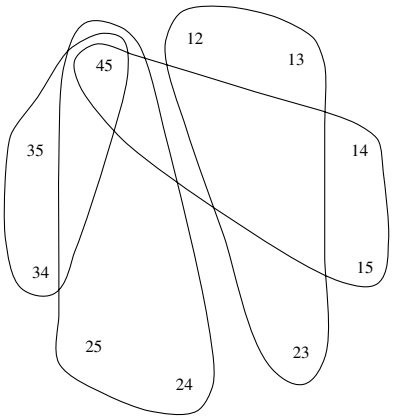

Figure 1.10: Covering all pairs of $\{1,2,3,4,5\}$ with tickets $\{1,2,3\}$, $\{1,4,5\}$, $\{2,4,5\}$, $\{3,4,5\}$

problem. It was going to be some type of covering problem, where each ticket we buy was going to "cover" some of the possible 4-element subsets of the psychic's set. Finding the absolute smallest set of tickets to cover everything was a special instance of the NP-complete problem *set cover* (discussed in Section 18.1 (page 621)), and presumably computationally intractable.

It was indeed a special instance of set cover, completely specified by only four numbers: the size n of the candidate set S (typically $n \approx 15$), the number of slots k for numbers on each ticket (typically $k \approx 6$), the number of psychically-promised correct numbers j from S (say $j = 4$), and finally, the number of matching numbers l necessary to win a prize (say $l = 3$). Figure 1.10 illustrates a covering of a smaller instance, where $n = 5$, $j = k = 3$, and $l = 2$.

"Although it will be hard to find the *exact* minimum set of tickets to buy, with heuristics I should be able to get you pretty close to the cheapest covering ticket set," I told him. "Will that be good enough?"

"So long as it generates better ticket sets than my competitor's program, that will be fine. His system doesn't always guarantee a win. I really appreciate your help on this, Professor Skiena."

"One last thing. If your program can train people to pick lottery winners, why don't you use it to win the lottery yourself?"

"I look forward to talking to you again real soon, Professor Skiena. Thanks for the help."

I hung up the phone and got back to thinking. It seemed like the perfect project to give to a bright undergraduate. After modeling it in terms of sets and subsets, the basic components of a solution seemed fairly straightforward:

- We needed the ability to generate all subsets of k numbers from the candidate set S. Algorithms for generating and ranking/unranking subsets of sets are presented in Section 14.5 (page 452).
- We needed the right formulation of what it meant to have a covering set of purchased tickets. The obvious criteria would be to pick a small set of tickets such that we have purchased at least one ticket containing each of the $\binom{n}{l}$ l-subsets of S that might pay off with the prize.
- We needed to keep track of which prize combinations we have thus far covered. We seek tickets to cover as many thus-far-uncovered prize combinations as possible. The currently covered combinations are a subset of all possible combinations. Data structures for subsets are discussed in Section 12.5 (page 385). The best candidate seemed to be a bit vector, which would answer in constant time "is this combination already covered?"
- We needed a search mechanism to decide which ticket to buy next. For small enough set sizes, we could do an exhaustive search over all possible subsets of tickets and pick the smallest one. For larger problems, a randomized search process like simulated annealing (see Section 7.5.3 (page 254)) would select tickets-to-buy to cover as many uncovered combinations as possible. By repeating this randomized procedure several times and picking the best solution, we would be likely to come up with a good set of tickets.

Excluding the details of the search mechanism, the pseudocode for the bookkeeping looked something like this:

LottoTicketSet(n, k, l)
    Initialize the $\binom{n}{l}$-element bit-vector V to all false
    While there exists a false entry in V
        Select a k-subset T of $\{1, \ldots, n\}$ as the next ticket to buy
        For each of the l-subsets T_i of T, $V[rank(T_i)] = \text{true}$
    Report the set of tickets bought

The bright undergraduate, Fayyaz Younas, rose to the challenge. Based on this framework, he implemented a brute-force search algorithm and found optimal solutions for problems with $n \leq 5$ in a reasonable time. He implemented a random search procedure to solve larger problems, tweaking it for a while before settling on the best variant. Finally, the day arrived when we could call Lotto Systems Group and announce that we had solved the problem.

"Our program found an optimal solution for $n = 15$, $k = 6$, $j = 6$, $l = 3$ meant buying 28 tickets."

"Twenty-eight tickets!" complained the president. "You must have a bug. Look, these five tickets will suffice to cover everything *twice* over: $\{2, 4, 8, 10, 13, 14\}$, $\{4, 5, 7, 8, 12, 15\}$, $\{1, 2, 3, 6, 11, 13\}$, $\{3, 5, 6, 9, 10, 15\}$, $\{1, 7, 9, 11, 12, 14\}$."

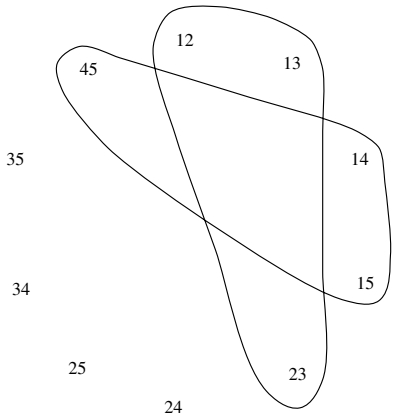

Figure 1.11: Guaranteeing a winning pair from $\{1, 2, 3, 4, 5\}$ using only tickets $\{1, 2, 3\}$ and $\{1, 4, 5\}$

We fiddled with this example for a while before admitting that he was right. *We hadn't modeled the problem correctly!* In fact, we didn't need to explicitly cover all possible winning combinations. Figure 1.11 illustrates the principle by giving a two-ticket solution to our previous four-ticket example. Such unpromising outcomes as $\{2, 3, 4\}$ and $\{3, 4, 5\}$ each agree in one matching pair with tickets from Figure 1.11. We were trying to cover too many combinations, and the penny-pinching psychics were unwilling to pay for such extravagance.

Fortunately, this story has a happy ending. The general outline of our search-based solution still holds for the real problem. All we must fix is which subsets we get credit for covering with a given set of tickets. After this modification, we obtained the kind of results they were hoping for. Lotto Systems Group gratefully accepted our program to incorporate into their product, and hopefully hit the jackpot with it.

The moral of this story is to make sure that you model the problem correctly before trying to solve it. In our case, we came up with a reasonable model, but didn't work hard enough to validate it before we started to program. Our misinterpretation would have become obvious had we worked out a small example by hand and bounced it off our sponsor before beginning work. Our success in recovering from this error is a tribute to the basic correctness of our initial formulation, and our use of well-defined abstractions for such tasks as (1) ranking/unranking k-subsets, (2) the set data structure, and (3) combinatorial search.

Chapter Notes

Every decent algorithm book reflects the design philosophy of its author. For students seeking alternative presentations and viewpoints, we particularly recommend the books of Corman, et. al [CLRS01], Kleinberg/Tardos [KT06], and Manber [Man89].

Formal proofs of algorithm correctness are important, and deserve a fuller discussion than we are able to provide in this chapter. See Gries [Gri89] for a thorough introduction to the techniques of program verification.

The movie scheduling problem represents a very special case of the general *independent set* problem, which is discussed in Section 16.2 (page 528). The restriction limits the allowable input instances to *interval* graphs, where the vertices of the graph G can be represented by intervals on the line and (i, j) is an edge of G iff the intervals overlap. Golumbic [Gol04] provides a full treatment of this interesting and important class of graphs.

Jon Bentley's *Programming Pearls* columns are probably the best known collection of algorithmic "war stories." Originally published in the *Communications of the ACM*, they have been collected in two books [Ben90, Ben99]. Brooks's *The Mythical Man Month* [Bro95] is another wonderful collection of war stories, focused more on software engineering than algorithm design, but they remain a source of considerable wisdom. Every programmer should read all these books, for pleasure as well as insight.

Our solution to the lotto ticket set covering problem is presented in more detail in [YS96].

1.7 Exercises

Finding Counterexamples

1-1. *[3]* Show that $a + b$ can be less than $\min(a, b)$.

1-2. *[3]* Show that $a \times b$ can be less than $\min(a, b)$.

1-3. *[5]* Design/draw a road network with two points a and b such that the fastest route between a and b is not the shortest route.

1-4. *[5]* Design/draw a road network with two points a and b such that the shortest route between a and b is not the route with the fewest turns.

1-5. *[4]* The *knapsack problem* is as follows: given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$, and a target number T, find a subset of S which adds up exactly to T. For example, there exists a subset within $S = \{1, 2, 5, 9, 10\}$ that adds up to $T = 22$ but not $T = 23$.

Find counterexamples to each of the following algorithms for the knapsack problem. That is, giving an S and T such that the subset is selected using the algorithm does not leave the knapsack completely full, even though such a solution exists.

(a) Put the elements of S in the knapsack in left to right order if they fit, i.e. the first-fit algorithm.

(b) Put the elements of S in the knapsack from smallest to largest, i.e. the best-fit algorithm.

(c) Put the elements of S in the knapsack from largest to smallest.

1-6. *[5]* The *set cover problem* is as follows: given a set of subsets $S_1, ..., S_m$ of the universal set $U = \{1, ..., n\}$, find the smallest subset of subsets $T \subset S$ such that $\cup_{t_i \in T} t_i = U$. For example, there are the following subsets, $S_1 = \{1, 3, 5\}$, $S_2 = \{2, 4\}$, $S_3 = \{1, 4\}$, and $S_4 = \{2, 5\}$ The set cover would then be S_1 and S_2.

Find a counterexample for the following algorithm: Select the largest subset for the cover, and then delete all its elements from the universal set. Repeat by adding the subset containing the largest number of uncovered elements until all are covered.

Proofs of Correctness

1-7. *[3]* Prove the correctness of the following recursive algorithm to multiply two natural numbers, for all integer constants $c \geq 2$.

```
function multiply(y, z)
    comment Return the product yz.
1.      if z = 0 then return(0) else
2.      return(multiply(cy, ⌊z/c⌋) + y · (z mod c))
```

1-8. *[3]* Prove the correctness of the following algorithm for evaluating a polynomial. $P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

```
function horner(A, x)
    p = A_n
    for i from n − 1 to 0
        p = p ∗ x + A_i
    return p
```

1-9. *[3]* Prove the correctness of the following sorting algorithm.

```
function bubblesort (A : list[1 . . . n])
    var int i, j
    for i from n to 1
        for j from 1 to i − 1
            if (A[j] > A[j + 1])
                swap the values of A[j] and A[j + 1]
```

Induction

1-10. *[3]* Prove that $\sum_{i=1}^{n} i = n(n+1)/2$ for $n \geq 0$, by induction.

1-11. *[3]* Prove that $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ for $n \geq 0$, by induction.

1-12. *[3]* Prove that $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$ for $n \geq 0$, by induction.

1-13. *[3]* Prove that

$$\sum_{i=1}^{n} i(i+1)(i+2) = n(n+1)(n+2)(n+3)/4$$

1-14. *[5]* Prove by induction on $n \geq 1$ that for every $a \neq 1$,

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

1-15. *[3]* Prove by induction that for $n \geq 1$,

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$$

1-16. *[3]* Prove by induction that $n^3 + 2n$ is divisible by 3 for all $n \geq 0$.

1-17. *[3]* Prove by induction that a tree with n vertices has exactly $n - 1$ edges.

1-18. *[3]* Prove by mathematical induction that the sum of the cubes of the first n positive integers is equal to the square of the sum of these integers, i.e.

$$\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$$

Estimation

1-19. *[3]* Do all the books you own total at least one million pages? How many total pages are stored in your school library?

1-20. *[3]* How many words are there in this textbook?

1-21. *[3]* How many hours are one million seconds? How many days? Answer these questions by doing all arithmetic in your head.

1-22. *[3]* Estimate how many cities and towns there are in the United States.

1-23. *[3]* Estimate how many cubic miles of water flow out of the mouth of the Mississippi River each day. Do not look up any supplemental facts. Describe all assumptions you made in arriving at your answer.

1-24. *[3]* Is disk drive access time normally measured in milliseconds (thousandths of a second) or microseconds (millionths of a second)? Does your RAM memory access a word in more or less than a microsecond? How many instructions can your CPU execute in one year if the machine is left running all the time?

1-25. *[4]* A sorting algorithm takes 1 second to sort 1,000 items on your local machine. How long will it take to sort 10,000 items...

(a) if you believe that the algorithm takes time proportional to n^2, and

(b) if you believe that the algorithm takes time roughly proportional to $n \log n$?

Implementation Projects

1-26. *[5]* Implement the two TSP heuristics of Section 1.1 (page 5). Which of them gives better-quality solutions in practice? Can you devise a heuristic that works better than both of them?

1-27. *[5]* Describe how to test whether a given set of tickets establishes sufficient coverage in the Lotto problem of Section 1.6 (page 23). Write a program to find good ticket sets.

Interview Problems

1-28. *[5]* Write a function to perform integer division without using either the `/` or `*` operators. Find a fast way to do it.

1-29. *[5]* There are 25 horses. At most, 5 horses can race together at a time. You must determine the fastest, second fastest, and third fastest horses. Find the minimum number of races in which this can be done.

1-30. *[3]* How many piano tuners are there in the entire world?

1-31. *[3]* How many gas stations are there in the United States?

1-32. *[3]* How much does the ice in a hockey rink weigh?

1-33. *[3]* How many miles of road are there in the United States?

1-34. *[3]* On average, how many times would you have to flip open the Manhattan phone book at random in order to find a specific name?

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

1-1. "The $3n + 1$ Problem" – Programming Challenges 110101, UVA Judge 100.

1-2. "The Trip" – Programming Challenges 110103, UVA Judge 10137.

1-3. "Australian Voting" – Programming Challenges 110108, UVA Judge 10142.

2

Algorithm Analysis

Algorithms are the most important and durable part of computer science because they can be studied in a language- and machine-independent way. This means that we need techniques that enable us to compare the efficiency of algorithms without implementing them. Our two most important tools are (1) the RAM model of computation and (2) the asymptotic analysis of worst-case complexity.

Assessing algorithmic performance makes use of the "big Oh" notation that, proves essential to compare algorithms and design more efficient ones. While the hopelessly *practical* person may blanch at the notion of theoretical analysis, we present the material because it really is useful in thinking about algorithms.

This method of keeping score will be the most mathematically demanding part of this book. But once you understand the intuition behind these ideas, the formalism becomes a lot easier to deal with.

2.1 The RAM Model of Computation

Machine-independent algorithm design depends upon a hypothetical computer called the *Random Access Machine* or RAM. Under this model of computation, we are confronted with a computer where:

- Each *simple* operation (+, *, –, =, if, call) takes exactly one time step.
- Loops and subroutines are *not* considered simple operations. Instead, they are the composition of many single-step operations. It makes no sense for *sort* to be a single-step operation, since sorting 1,000,000 items will certainly take much longer than sorting 10 items. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_2,

- Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Under the RAM model, we measure run time by counting up the number of steps an algorithm takes on a given problem instance. If we assume that our RAM executes a given number of steps per second, this operation count converts naturally to the actual running time.

The RAM is a simple model of how computers perform. Perhaps it sounds too simple. After all, multiplying two numbers takes more time than adding two numbers on most processors, which violates the first assumption of the model. Fancy compiler loop unrolling and hyperthreading may well violate the second assumption. And certainly memory access times differ greatly depending on whether data sits in cache or on the disk. This makes us zero for three on the truth of our basic assumptions.

And yet, despite these complaints, the RAM proves an *excellent* model for understanding how an algorithm will perform on a real computer. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. We use the RAM model because it is useful in practice.

Every model has a size range over which it is useful. Take, for example, the model that the Earth is flat. You might argue that this is a bad model, since it has been fairly well established that the Earth is in fact round. But, when laying the foundation of a house, the flat Earth model is sufficiently accurate that it can be reliably used. It is so much easier to manipulate a flat-Earth model that it is inconceivable that you would try to think spherically when you don't have to.[1]

The same situation is true with the RAM model of computation. We make an abstraction that is generally very useful. It is quite difficult to design an algorithm such that the RAM model gives you substantially misleading results. The robustness of the RAM enables us to analyze algorithms in a machine-independent way.

Take-Home Lesson: Algorithms can be understood and studied in a language- and machine-independent manner.

2.1.1 Best, Worst, and Average-Case Complexity

Using the RAM model of computation, we can count how many steps our algorithm takes on any given input instance by executing it. However, to understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

To understand the notions of the best, worst, and average-case complexity, think about running an algorithm over all possible instances of data that can be

[1] The Earth is not completely spherical either, but a spherical Earth provides a useful model for such things as longitude and latitude.

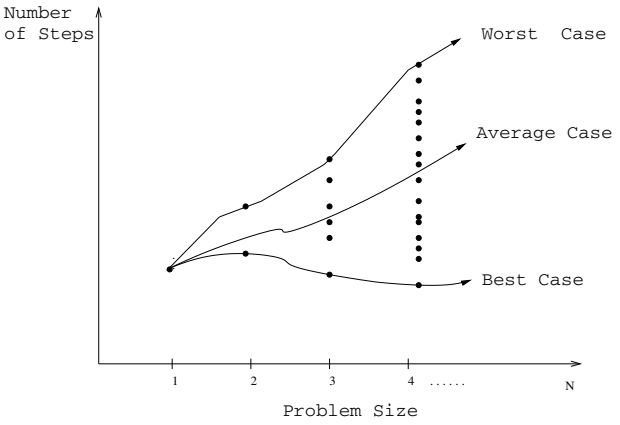

Figure 2.1: Best, worst, and average-case complexity

fed to it. For the problem of sorting, the set of possible input instances consists of all possible arrangements of n keys, over all possible values of n. We can represent each input instance as a point on a graph (shown in Figure 2.1) where the x-axis represents the size of the input problem (for sorting, the number of items to sort), and the y-axis denotes the number of steps taken by the algorithm in this instance.

These points naturally align themselves into columns, because only integers represent possible input size (e.g., it makes no sense to sort 10.57 items). We can define three interesting functions over the plot of these points:

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken in any instance of size n. This represents the curve passing through the highest point in each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken in any instance of size n. This represents the curve passing through the lowest point of each column.
- The *average-case complexity* of the algorithm, which is the function defined by the average number of steps over all instances of size n.

The worst-case complexity proves to be most useful of these three measures in practice. Many people find this counterintuitive. To illustrate why, try to project what will happen if you bring n dollars into a casino to gamble. The best case, that you walk out owning the place, is possible but so unlikely that you should not even think about it. The worst case, that you lose all n dollars, is easy to calculate and distressingly likely to happen. The average case, that the typical bettor loses 87.32% of the money that he brings to the casino, is difficult to establish and its meaning subject to debate. What exactly does *average* mean? Stupid people lose

more than smart people, so are you smarter or stupider than the average person, and by how much? Card counters at blackjack do better on average than customers who accept three or more free drinks. We avoid all these complexities and obtain a very useful result by just considering the worst case.

The important thing to realize is that each of these time complexities define a numerical function, representing time versus problem size. These functions are as well defined as any other numerical function, be it $y = x^2 - 2x + 1$ or the price of Google stock as a function of time. But time complexities are such complicated functions that we must simplify them to work with them. For this, we need the "Big Oh" notation.

2.2 The Big Oh Notation

The best, worst, and average-case time complexities for any given algorithm are numerical functions over the size of possible problem instances. However, it is very difficult to work precisely with these functions, because they tend to:

- *Have too many bumps* – An algorithm such as binary search typically runs a bit faster for arrays of size exactly $n = 2^k - 1$ (where k is an integer), because the array partitions work out nicely. This detail is not particularly significant, but it warns us that the *exact* time complexity function for any algorithm is liable to be very complicated, with little up and down bumps as shown in Figure 2.2.

- *Require too much detail to specify precisely* – Counting the exact number of RAM instructions executed in the worst case requires the algorithm be specified to the detail of a complete computer program. Further, the precise answer depends upon uninteresting coding details (e.g., did he use a case statement or nested ifs?). Performing a precise worst-case analysis like

  $$T(n) = 12754n^2 + 4353n + 834 \lg_2 n + 13546$$

  would clearly be very difficult work, but provides us little extra information than the observation that "the time grows quadratically with n."

It proves to be much easier to talk in terms of simple upper and lower bounds of time-complexity functions using the Big Oh notation. The Big Oh simplifies our analysis by ignoring levels of detail that do not impact our comparison of algorithms.

The Big Oh notation ignores the difference between multiplicative constants. The functions $f(n) = 2n$ and $g(n) = n$ are identical in Big Oh analysis. This makes sense given our application. Suppose a given algorithm in (say) C language ran twice as fast as one with the same algorithm written in Java. This multiplicative

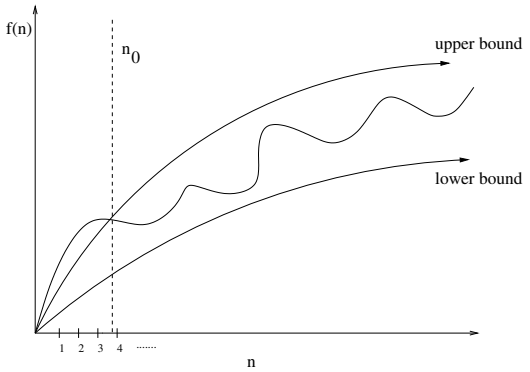

Figure 2.2: Upper and lower bounds valid for $n > n_0$ smooth out the behavior of complex functions

factor of two tells us nothing about the algorithm itself, since both programs implement exactly the same algorithm. We ignore such constant factors when comparing two algorithms.

The formal definitions associated with the Big Oh notation are as follows:

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).
- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Got it? These definitions are illustrated in Figure 2.3. Each of these definitions assumes a constant n_0 beyond which they are always satisfied. We are not concerned about small values of n (i.e. , anything to the left of n_0). After all, we don't really care whether one sorting algorithm sorts six items faster than another, but seek which algorithm proves faster when sorting 10,000 or 1,000,000 items. The Big Oh notation enables us to ignore details and focus on the big picture.

Take-Home Lesson: The Big Oh notation and worst-case analysis are tools that greatly simplify our ability to compare the efficiency of algorithms.

Make sure you understand this notation by working through the following examples. We choose certain constants (c and n_0) in the explanations below because

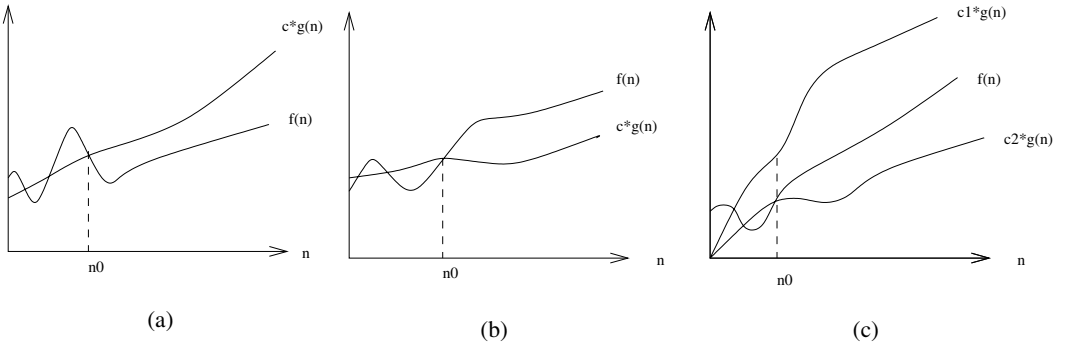

Figure 2.3: Illustrating the big (a) O, (b) Ω, and (c) Θ notations

they work and make a point, but other pairs of constants will do exactly the same job. You are free to choose any constants that maintain the same inequality—ideally constants that make it obvious that the inequality holds:

$3n^2 - 100n + 6 = O(n^2)$, because I choose $c = 3$ and $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, because I choose $c = 1$ and $n^3 > 3n^2 - 100n + 6$ when $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, because for any c I choose $c \times n < 3n^2$ when $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, because I choose $c = 2$ and $2n^2 < 3n^2 - 100n + 6$ when $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, because I choose $c = 3$ and $3n^2 - 100n + 6 < n^3$ when $n > 3$;

$3n^2 - 100n + 6 = \Omega(n)$, because for any c I choose $cn < 3n^2 - 100n + 6$ when $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, because both O and Ω apply;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, because only O applies;

$3n^2 - 100n + 6 \neq \Theta(n)$, because only Ω applies.

The Big Oh notation provides for a rough notion of equality when comparing functions. It is somewhat jarring to see an expression like $n^2 = O(n^3)$, but its meaning can always be resolved by going back to the definitions in terms of upper and lower bounds. It is perhaps most instructive to read the "=" here as meaning *one of the functions that are*. Clearly, n^2 is one of functions that are $O(n^3)$.

Stop and Think: Back to the Definition

Problem: Is $2^{n+1} = \Theta(2^n)$?

Solution: Designing novel algorithms requires cleverness and inspiration. However, applying the Big Oh notation is best done by swallowing any creative instincts you may have. All Big Oh problems can be correctly solved by going back to the definition and working with that.

- *Is* $2^{n+1} = O(2^n)$*?* Well, $f(n) = O(g(n))$ iff (*if and only if*) there exists a constant c such that for all sufficiently large n $f(n) \leq c \cdot g(n)$. Is there? The key observation is that $2^{n+1} = 2 \cdot 2^n$, so $2 \cdot 2^n \leq c \cdot 2^n$ for any $c \geq 2$.
- *Is* $2^{n+1} = \Omega(2^n)$*?* Go back to the definition. $f(n) = \Omega(g(n))$ iff there exists a constant $c > 0$ such that for all sufficiently large n $f(n) \geq c \cdot g(n)$. This would be satisfied for any $0 < c \leq 2$. Together the Big Oh and Ω bounds imply $2^{n+1} = \Theta(2^n)$

∎

Stop and Think: Hip to the Squares?

Problem: Is $(x + y)^2 = O(x^2 + y^2)$.

Solution: Working with the Big Oh means going back to the definition at the slightest sign of confusion. By definition, this expression is valid iff we can find some c such that $(x + y)^2 \leq c(x^2 + y^2)$.

My first move would be to expand the left side of the equation, i.e. $(x + y)^2 = x^2 + 2xy + y^2$. If the middle $2xy$ term wasn't there, the inequality would clearly hold for any $c > 1$. But it is there, so we need to relate the $2xy$ to $x^2 + y^2$. What if $x \leq y$? Then $2xy \leq 2y^2 \leq 2(x^2 + y^2)$. What if $x \geq y$? Then $2xy \leq 2x^2 \leq 2(x^2 + y^2)$. Either way, we now can bound this middle term by two times the right-side function. This means that $(x + y)^2 \leq 3(x^2 + y^2)$, and so the result holds. ∎

2.3 Growth Rates and Dominance Relations

With the Big Oh notation, we cavalierly discard the multiplicative constants. Thus, the functions $f(n) = 0.001n^2$ and $g(n) = 1000n^2$ are treated identically, even though $g(n)$ is a million times larger than $f(n)$ for all values of n.

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

The reason why we are content with coarse Big Oh analysis is provided by Figure 2.4, which shows the growth rate of several common time analysis functions. In particular, it shows how long algorithms that use $f(n)$ operations take to run on a fast computer, where each operation takes one nanosecond (10^{-9} seconds). The following conclusions can be drawn from this table:

- All such algorithms take roughly the same time for $n = 10$.
- Any algorithm with $n!$ running time becomes useless for $n \geq 20$.
- Algorithms whose running time is 2^n have a greater operating range, but become impractical for $n > 40$.
- Quadratic-time algorithms whose running time is n^2 remain usable up to about $n = 10,000$, but quickly deteriorate with larger inputs. They are likely to be hopeless for $n >$ 1,000,000.
- Linear-time and $n \lg n$ algorithms remain practical on inputs of one billion items.
- An $O(\lg n)$ algorithm hardly breaks a sweat for any imaginable value of n.

The bottom line is that even ignoring constant factors, we get an excellent idea of whether a given algorithm is appropriate for a problem of a given size. An algorithm whose running time is $f(n) = n^3$ seconds will beat one whose running time is $g(n) =$ 1,000,000 $\cdot n^2$ seconds only when $n <$ 1,000,000. Such enormous differences in constant factors between algorithms occur far less frequently in practice than large problems do.

2.3.1 Dominance Relations

The Big Oh notation groups functions into a set of classes, such that all the functions in a particular class are equivalent with respect to the Big Oh. Functions $f(n) = 0.34n$ and $g(n) = 234{,}234n$ belong in the same class, namely those that are order $\Theta(n)$. Further, when two functions f and g belong to different classes, they are *different* with respect to our notation. Either $f(n) = O(g(n))$ or $g(n) = O(f(n))$, but not both.

We say that a faster-growing function *dominates* a slower-growing one, just as a faster-growing country eventually comes to dominate the laggard. When f and g belong to different classes (i.e., $f(n) \neq \Theta(g(n))$), we say g *dominates* f when $f(n) = O(g(n))$, sometimes written $g \gg f$.

The good news is that only a few function classes tend to occur in the course of basic algorithm analysis. These suffice to cover almost all the algorithms we will discuss in this text, and are listed in order of increasing dominance:

- *Constant functions,* $f(n) = 1$ – Such functions might measure the cost of adding two numbers, printing out "The Star Spangled Banner," or the growth realized by functions such as $f(n) = \min(n, 100)$. In the big picture, there is no dependence on the parameter n.
- *Logarithmic functions,* $f(n) = \log n$ – Logarithmic time-complexity shows up in algorithms such as binary search. Such functions grow quite slowly as n gets big, but faster than the constant function (which is standing still, after all). Logarithms will be discussed in more detail in Section 2.6 (page 46)
- *Linear functions,* $f(n) = n$ – Such functions measure the cost of looking at each item once (or twice, or ten times) in an n-element array, say to identify the biggest item, the smallest item, or compute the average value.
- *Superlinear functions,* $f(n) = n \lg n$ – This important class of functions arises in such algorithms as Quicksort and Mergesort. They grow just a little faster than linear (see Figure 2.4), just enough to be a different dominance class.
- *Quadratic functions,* $f(n) = n^2$ – Such functions measure the cost of looking at most or all *pairs* of items in an n-element universe. This arises in algorithms such as insertion sort and selection sort.
- *Cubic functions,* $f(n) = n^3$ – Such functions enumerate through all *triples* of items in an n-element universe. These also arise in certain dynamic programming algorithms developed in Chapter 8.
- *Exponential functions,* $f(n) = c^n$ *for a given constant* $c > 1$ – Functions like 2^n arise when enumerating all subsets of n items. As we have seen, exponential algorithms become useless fast, but not as fast as...
- *Factorial functions,* $f(n) = n!$ – Functions like $n!$ arise when generating all permutations or orderings of n items.

The intricacies of dominance relations will be futher discussed in Section 2.9.2 (page 56). However, all you really need to understand is that:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

Take-Home Lesson: Although esoteric functions arise in advanced algorithm analysis, a small variety of time complexities suffice and account for most algorithms that are widely used in practice.

2.4 Working with the Big Oh

You learned how to do simplifications of algebraic expressions back in high school. Working with the Big Oh requires dusting off these tools. *Most* of what you learned there still holds in working with the Big Oh, but not everything.

2.4.1 Adding Functions

The sum of two functions is governed by the dominant one, namely:

$$O(f(n)) + O(g(n)) \rightarrow O(\max(f(n), g(n)))$$

$$\Omega(f(n)) + \Omega(g(n)) \rightarrow \Omega(\max(f(n), g(n)))$$

$$\Theta(f(n)) + \Theta(g(n)) \rightarrow \Theta(\max(f(n), g(n)))$$

This is very useful in simplifying expressions, since it implies that $n^3 + n^2 + n + 1 = O(n^3)$. Everything is small potatoes besides the dominant term.

The intuition is as follows. At least half the bulk of $f(n) + g(n)$ must come from the larger value. The dominant function will, by definition, provide the larger value as $n \rightarrow \infty$. Thus, dropping the smaller function from consideration reduces the value by at most a factor of $1/2$, which is just a multiplicative constant. Suppose $f(n) = O(n^2)$ and $g(n) = O(n^2)$. This implies that $f(n) + g(n) = O(n^2)$ as well.

2.4.2 Multiplying Functions

Multiplication is like repeated addition. Consider multiplication by any constant $c > 0$, be it 1.02 or 1,000,000. Multiplying a function by a constant can not affect its asymptotic behavior, because we can multiply the bounding constants in the Big Oh analysis of $c \cdot f(n)$ by $1/c$ to give appropriate constants for the Big Oh analysis of $f(n)$. Thus:

$$O(c \cdot f(n)) \rightarrow O(f(n))$$

$$\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$$

$$\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$$

Of course, c must be strictly positive (i.e., $c > 0$) to avoid any funny business, since we can wipe out even the fastest growing function by multiplying it by zero.

On the other hand, when two functions in a product are increasing, both are important. The function $O(n! \log n)$ dominates $n!$ just as much as $\log n$ dominates 1. In general,

$$O(f(n)) * O(g(n)) \rightarrow O(f(n) * g(n))$$

$$\Omega(f(n)) * \Omega(g(n)) \rightarrow \Omega(f(n) * g(n))$$

$$\Theta(f(n)) * \Theta(g(n)) \rightarrow \Theta(f(n) * g(n))$$

Stop and Think: Transitive Experience

Problem: Show that Big Oh relationships are transitive. That is, if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Solution: We always go back to the definition when working with the Big Oh. What we need to show here is that $f(n) \leq c_3 h(n)$ for $n > n_3$ given that $f(n) \leq c_1 g(n)$ and $g(n) \leq c_2 h(n)$, for $n > n_1$ and $n > n_2$, respectively. Cascading these inequalities, we get that

$$f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$$

for $n > n_3 = \max(n_1, n_2)$. ∎

2.5 Reasoning About Efficiency

Gross reasoning about an algorithm's running time of is usually easy given a precise written description of the algorithm. In this section, I will work through several examples, perhaps in greater detail than necessary.

2.5.1 Selection Sort

Here we analyze the selection sort algorithm, which repeatedly identifies the smallest remaining unsorted element and puts it at the end of the sorted portion of the array. An animation of selection sort in action appears in Figure 2.5, and the code is shown below:

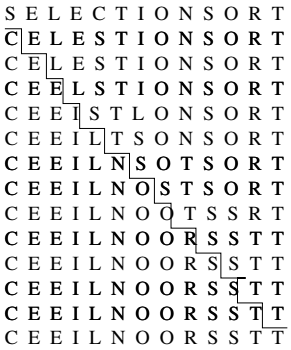

Figure 2.5: Animation of selection sort in action

```
selection_sort(int s[], int n)
{
        int i,j;                /* counters */
        int min;                /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

The outer loop goes around n times. The nested inner loop goes around $n-i-1$ times, where i is the index of the outer loop. The exact number of times the *if* statement is executed is given by:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1$$

What this sum is doing is adding up the integers in decreasing order starting from $n-1$, i.e.

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

How can we reason about such a formula? We must solve the summation formula using the techniques of Section 1.3.5 (page 17) to get an exact value. But, with the Big Oh we are only interested in the *order* of the expression. One way to think about it is that we are adding up $n-1$ terms, whose average value is about $n/2$. This yields $S(n) \approx n(n-1)/2$.

Another way to think about it is in terms of upper and lower bounds. We have n terms at most, each of which is at most $n-1$. Thus, $S(n) \leq n(n-1) = O(n^2)$. We have $n/2$ terms each that are bigger than $n/2$. Thus $S(n) =\geq (n/2) \times (n/2) = \Omega(n^2)$. Together, this tells us that the running time is $\Theta(n^2)$, meaning that selection sort is quadratic.

2.5.2 Insertion Sort

A basic rule of thumb in Big Oh analysis is that worst-case running time follows from multiplying the largest number of times each nested loop can iterate. Consider the insertion sort algorithm presented on page 4, whose inner loops are repeated here:

```
for (i=1; i<n; i++) {
        j=i;
        while ((j>0) && (s[j] < s[j-1])) {
                swap(&s[j],&s[j-1]);
                j = j-1;
        }
}
```

How often does the inner *while* loop iterate? This is tricky because there are two different stopping conditions: one to prevent us from running off the bounds of the array ($j > 0$) and the other to mark when the element finds its proper place in sorted order ($s[j] < s[j-1]$). Since worst-case analysis seeks an upper bound on the running time, we ignore the early termination and assume that this loop *always* goes around i times. In fact, we can assume it *always* goes around n times since $i < n$. Since the outer loop goes around n times, insertion sort must be a quadratic-time algorithm, i.e. $O(n^2)$.

This crude "round it up" analysis always does the job, in that the Big Oh running time bound you get will always be correct. Occasionally, it might be too generous, meaning the actual worst case time might be of a lower order than implied by such analysis. Still, I strongly encourage this kind of reasoning as a basis for simple algorithm analysis.

2.5.3 String Pattern Matching

Pattern matching is the most fundamental algorithmic operation on text strings. This algorithm implements the find command available in any web browser or text editor:

Problem: Substring Pattern Matching
Input: A text string t and a pattern string p.
Output: Does t contain the pattern p as a substring, and if so where?

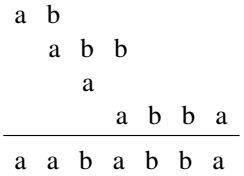

Figure 2.6: Searching for the substring *abba* in the text *aababba*.

Perhaps you are interested finding where "Skiena" appears in a given news article (well, I would be interested in such a thing). This is an instance of string pattern matching with t as the news article and p="Skiena."

There is a fairly straightforward algorithm for string pattern matching that considers the possibility that p may start at each possible position in t and then tests if this is so.

```
int findmatch(char *p, char *t)
{
        int i,j;                        /* counters */
        int m, n;                       /* string lengths */

        m = strlen(p);
        n = strlen(t);

        for (i=0; i<=(n-m); i=i+1) {
                j=0;
                while ((j<m) && (t[i+j]==p[j]))
                        j = j+1;
                if (j == m) return(i);
        }

        return(-1);
}
```

What is the worst-case running time of these two nested loops? The inner *while* loop goes around at most m times, and potentially far less when the pattern match fails. This, plus two other statements, lies within the outer *for* loop. The outer loop goes around at most $n - m$ times, since no complete alignment is possible once we get too far to the right of the text. The time complexity of nested loops multiplies, so this gives a worst-case running time of $O((n - m)(m + 2))$.

We did not count the time it takes to compute the length of the strings using the function *strlen*. Since the implementation of *strlen* is not given, we must guess how long it should take. If we explicitly count the number of characters until we

hit the end of the string; this would take time linear in the length of the string. This suggests that the running time should be $O(n + m + (n - m)(m + 2))$.

Let's use our knowledge of the Big Oh to simplify things. Since $m + 2 = \Theta(m)$, the "+2" isn't interesting, so we are left with $O(n + m + (n - m)m)$. Multiplying this out yields $O(n + m + nm - m^2)$, which still seems kind of ugly.

However, in any interesting problem we know that $n \geq m$, since it is impossible to have p as a substring of t for any pattern longer than the text itself. One consequence of this is that $n + m \leq 2n = \Theta(n)$. Thus our worst-case running time simplifies further to $O(n + nm - m^2)$.

Two more observations and we are done. First, note that $n \leq nm$, since $m \geq 1$ in any interesting pattern. Thus $n + nm = \Theta(nm)$, and we can drop the additive n, simplifying our analysis to $O(nm - m^2)$.

Finally, observe that the $-m^2$ term is negative, and thus only serves to lower the value within. Since the Big Oh gives an upper bound, we can drop any negative term without invalidating the upper bound. That $n \geq m$ implies that $mn \geq m^2$, so the negative term is not big enough to cancel any other term which is left. Thus we can simply express the worst-case running time of this algorithm as $O(nm)$.

After you get enough experience, you will be able to do such an algorithm analysis in your head without even writing the algorithm down. After all, algorithm design for a given task involves mentally rifling through different possibilities and selecting the best approach. This kind of fluency comes with practice, but if you are confused about why a given algorithm runs in $O(f(n))$ time, start by writing it out carefully and then employ the reasoning we used in this section.

2.5.4 Matrix Multiplication

Nested summations often arise in the analysis of algorithms with nested loops. Consider the problem of matrix multiplication:

Problem: Matrix Multiplication
Input: Two matrices, A (of dimension $x \times y$) and B (dimension $y \times z$).
Output: An $x \times z$ matrix C where $C[i][j]$ is the dot product of the ith row of A and the jth column of B.

Matrix multiplication is a fundamental operation in linear algebra, presented with an example in catalog in Section 13.3 (page 401). That said, the elementary algorithm for matrix multiplication is implemented as a tight product of three nested loops:

```
for (i=1; i<=x; i++)
        for (j=1; j<=y; j++) {
                C[i][j] = 0;
                for (k=1; k<=z; k++)
                        C[i][j] += A[i][k] * B[k][j];
        }
```

How can we analyze the time complexity of this algorithm? The number of multiplications $M(x, y, z)$ is given by the following summation:

$$M(x, y, z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1$$

Sums get evaluated from the right inward. The sum of z ones is z, so

$$M(x, y, z) = \sum_{i=1}^{x} \sum_{j=1}^{y} z$$

The sum of y zs is just as simple, yz, so

$$M(x, y, z) = \sum_{i=1}^{x} yz$$

Finally, the sum of x yzs is xyz.

Thus the running of this matrix multiplication algorithm is $O(xyz)$. If we consider the common case where all three dimensions are the same, this becomes $O(n^3)$—i.e. , a cubic algorithm.

2.6 Logarithms and Their Applications

Logarithm is an anagram of algorithm, but that's not why we need to know what logarithms are. You've seen the button on your calculator but may have forgotten why it is there. A *logarithm* is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$. Further, this definition is the same as saying $b^{\log_b y} = y$.

Exponential functions grow at a distressingly fast rate, as anyone who has ever tried to pay off a credit card balance understands. Thus, inverse exponential functions—i.e. logarithms—grow refreshingly slowly. Logarithms arise in any process where things are repeatedly halved. We now look at several examples.

2.6.1 Logarithms and Binary Search

Binary search is a good example of an $O(\log n)$ algorithm. To locate a particular person p in a telephone book containing n names, you start by comparing p against the middle, or $(n/2)$nd name, say *Monroe, Marilyn*. Regardless of whether p belongs before this middle name (*Dean, James*) or after it (*Presley, Elvis*), after only one comparison you can discard one half of all the names in the book. The number of steps the algorithm takes equals the number of times we can halve n until only one name is left. By definition, this is exactly $\log_2 n$. Thus, twenty comparisons suffice to find any name in the million-name Manhattan phone book!

Binary search is one of the most powerful ideas in algorithm design. This power becomes apparent if we imagine being forced to live in a world with only unsorted

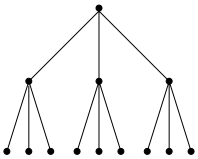

Figure 2.7: A height h tree with d children per node as d^h leaves. Here $h = 2$ and $d = 3$

telephone books. Figure 2.4 shows that $O(\log n)$ algorithms are fast enough to be used on problem instances of essentially unlimited size.

2.6.2 Logarithms and Trees

A binary tree of height 1 can have up to 2 leaf nodes, while a tree of height two can have up to four leaves. What is the height h of a rooted binary tree with n leaf nodes? Note that the number of leaves doubles every time we increase the height by one. To account for n leaves, $n = 2^h$ which implies that $h = \log_2 n$.

What if we generalize to trees that have d children, where $d = 2$ for the case of binary trees? A tree of height 1 can have up to d leaf nodes, while one of height two can have up to d^2 leaves. The number of possible leaves multiplies by d every time we increase the height by one, so to account for n leaves, $n = d^h$ which implies that $h = \log_d n$, as shown in Figure 2.7.

The punch line is that very short trees can have very many leaves, which is the main reason why binary trees prove fundamental to the design of fast data structures.

2.6.3 Logarithms and Bits

There are two bit patterns of length 1 (0 and 1) and four of length 2 (00, 01, 10, and 11). How many bits w do we need to represent any one of n different possibilities, be it one of n items or the integers from 1 to n?

The key observation is that there must be at least n different bit patterns of length w. Since the number of different bit patterns doubles as you add each bit, we need at least w bits where $2^w = n$—i.e., we need $w = \log_2 n$ bits.

2.6.4 Logarithms and Multiplication

Logarithms were particularly important in the days before pocket calculators. They provided the easiest way to multiply big numbers by hand, either implicitly using a slide rule or explicitly by using a book of logarithms.

Logarithms are still useful for multiplication, particularly for exponentiation. Recall that $\log_a(xy) = \log_a(x) + \log_a(y)$; i.e. , the log of a product is the sum of the logs. A direct consequence of this is

$$\log_a n^b = b \cdot \log_a n$$

So how can we compute a^b for any a and b using the $\exp(x)$ and $\ln(x)$ functions on your calculator, where $\exp(x) = e^x$ and $\ln(x) = \log_e(x)$? We know

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a)$$

so the problem is reduced to one multiplication plus one call to each of these functions.

2.6.5 Fast Exponentiation

Suppose that we need to *exactly* compute the value of a^n for some reasonably large n. Such problems occur in primality testing for cryptography, as discussed in Section 13.8 (page 420). Issues of numerical precision prevent us from applying the formula above.

The simplest algorithm performs $n - 1$ multiplications, by computing $a \times a \times \ldots \times a$. However, we can do better by observing that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If n is even, then $a^n = (a^{n/2})^2$. If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$. In either case, we have halved the size of our exponent at the cost of, at most, two multiplications, so $O(\lg n)$ multiplications suffice to compute the final value.

```
function power(a, n)
    if (n = 0) return(1)
    x = power(a, ⌊n/2⌋)
    if (n is even) then return(x^2)
        else return(a × x^2)
```

This simple algorithm illustrates an important principle of divide and conquer. It always pays to divide a job as evenly as possible. This principle applies to real life as well. When n is not a power of two, the problem cannot always be divided perfectly evenly, but a difference of one element between the two sides cannot cause any serious imbalance.

2.6.6 Logarithms and Summations

The *Harmonic numbers* arise as a special case of arithmetic progression, namely $H(n) = S(n, -1)$. They reflect the sum of the progression of simple reciprocals, namely,

$$H(n) = \sum_{i=1}^{n} 1/i \sim \ln n$$

Loss (apply the greatest)	Increase in level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

Figure 2.8: The Federal Sentencing Guidelines for fraud

The Harmonic numbers prove important because they usually explain "where the log comes from" when one magically pops out from algebraic manipulation. For example, the key to analyzing the average case complexity of Quicksort is the summation $S(n) = n \sum_{i=1}^{n} 1/i$. Employing the Harmonic number identity immediately reduces this to $\Theta(n \log n)$.

2.6.7 Logarithms and Criminal Justice

Figure 2.8 will be our final example of logarithms in action. This table appears in the Federal Sentencing Guidelines, used by courts throughout the United States. These guidelines are an attempt to standardize criminal sentences, so that a felon convicted of a crime before one judge receives the same sentence that they would before a different judge. To accomplish this, the judges have prepared an intricate point function to score the depravity of each crime and map it to time-to-serve.

Figure 2.8 gives the actual point function for fraud—a table mapping dollars stolen to points. Notice that the punishment increases by one level each time the amount of money stolen roughly doubles. That means that the level of punishment (which maps roughly linearly to the amount of time served) grows logarithmically with the amount of money stolen.

Think for a moment about the consequences of this. Many a corrupt CEO certainly has. It means that your total sentence grows *extremely* slowly with the amount of money you steal. Knocking off five liquor stores for $10,000 each will get you more time than embezzling $1,000,000 once. The corresponding benefit of stealing really large amounts of money is even greater. The moral of logarithmic growth is clear: *"If you are gonna do the crime, make it worth the time!"*

Take-Home Lesson: Logarithms arise whenever things are repeatedly halved or doubled.

2.7 Properties of Logarithms

As we have seen, stating $b^x = y$ is equivalent to saying that $x = \log_b y$. The b term is known as the *base* of the logarithm. Three bases are of particular importance for mathematical and historical reasons:

- *Base* $b = 2$ – The *binary logarithm*, usually denoted $\lg x$, is a base 2 logarithm. We have seen how this base arises whenever repeated halving (i.e., binary search) or doubling (i.e., nodes in trees) occurs. Most algorithmic applications of logarithms imply binary logarithms.

- *Base* $b = e$ – The *natural log*, usually denoted $\ln x$, is a base $e = 2.71828\ldots$ logarithm. The inverse of $\ln x$ is the exponential function $\exp(x) = e^x$ on your calculator. Thus, composing these functions gives us

$$\exp(\ln x) = x$$

- *Base* $b = 10$ – Less common today is the base-10 or *common logarithm*, usually denoted as $\log x$. This base was employed in slide rules and logarithm books in the days before pocket calculators.

We have already seen one important property of logarithms, namely that

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

The other important fact to remember is that it is easy to convert a logarithm from one base to another. This is a consequence of the formula:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Thus, changing the base of $\log b$ from base-a to base-c simply involves dividing by $\log_c a$. It is easy to convert a common log function to a natural log function, and vice versa.

Two implications of these properties of logarithms are important to appreciate from an algorithmic perspective:

- *The base of the logarithm has no real impact on the growth rate-* Compare the following three values: $\log_2(1,000,000) = 19.9316$, $\log_3(1,000,000) = 12.5754$, and $\log_{100}(1,000,000) = 3$. A big change in the base of the logarithm produces little difference in the value of the log. Changing the base of the log from a to c involves dividing by $\log_c a$. This conversion factor is lost to the Big Oh notation whenever a and c are constants. Thus we are usually justified in ignoring the base of the logarithm when analyzing algorithms.

- *Logarithms cut any function down to size-* The growth rate of the logarithm of any polynomial function is $O(\lg n)$. This follows because

  $$\log_a n^b = b \cdot \log_a n$$

  The power of binary search on a wide range of problems is a consequence of this observation. Note that doing a binary search on a sorted array of n^2 things requires only twice as many comparisons as a binary search on n things.

  Logarithms efficiently cut any function down to size. It is hard to do arithmetic on factorials except for logarithms, since

  $$n! = \Pi_{i=1}^{n} i \rightarrow \log n! = \sum_{i=1}^{n} \log i = \Theta(n \log n)$$

  provides another way for logarithms to pop up in algorithm analysis.

Stop and Think: Importance of an Even Split

Problem: How many queries does binary search take on the million-name Manhattan phone book if each split was 1/3 to 2/3 instead of 1/2 to 1/2?

Solution: When performing binary searches in a telephone book, how important is it that each query split the book exactly in half? Not much. For the Manhattan telephone book, we now use $\log_{3/2}(1,000,000) \approx 35$ queries in the worst case, not a significant change from $\log_2(1,000,000) \approx 20$. The power of binary search comes from its logarithmic complexity, not the base of the log. ∎

2.8 War Story: Mystery of the Pyramids

That look in his eyes should have warned me even before he started talking.

"We want to use a parallel supercomputer for a numerical calculation up to 1,000,000,000, but we need a faster algorithm to do it."

I'd seen that distant look before. Eyes dulled from too much exposure to the raw horsepower of supercomputers—machines so fast that brute force seemed to eliminate the need for clever algorithms; at least until the problems got hard enough.

"I am working with a Nobel prize winner to use a computer on a famous problem in number theory. Are you familiar with Waring's problem?"

I knew some number theory. "Sure. Waring's problem asks whether every integer can be expressed at least one way as the sum of at most four integer squares. For example, $78 = 8^2 + 3^2 + 2^2 + 1^2 = 7^2 + 5^2 + 2^2$. I remember proving that four squares suffice to represent any integer in my undergraduate number theory class. Yes, it's a famous problem but one that got solved about 200 years ago."

"No, we are interested in a different version of Waring's problem. A *pyramidal number* is a number of the form $(m^3 - m)/6$, for $m \geq 2$. Thus the first several pyramidal numbers are 1, 4, 10, 20, 35, 56, 84, 120, and 165. The conjecture since 1928 is that every integer can be represented by the sum of at most five such pyramidal numbers. We want to use a supercomputer to prove this conjecture on all numbers from 1 to 1,000,000,000."

"Doing a billion of anything will take a substantial amount of time," I warned. "The time you spend to compute the minimum representation of each number will be critical, because you are going to do it one billion times. Have you thought about what kind of an algorithm you are going to use?"

"We have already written our program and run it on a parallel supercomputer. It works very fast on smaller numbers. Still, it takes much too much time as soon as we get to 100,000 or so."

Terrific, I thought. Our supercomputer junkie had discovered asymptotic growth. No doubt his algorithm ran in something like quadratic time, and he got burned as soon as n got large.

"We need a faster program in order to get to one billion. Can you help us? Of course, we can run it on our parallel supercomputer when you are ready."

I am a sucker for this kind of challenge, finding fast algorithms to speed up programs. I agreed to think about it and got down to work.

I started by looking at the program that the other guy had written. He had built an array of all the $\Theta(n^{1/3})$ pyramidal numbers from 1 to n inclusive.[2] To test each number k in this range, he did a brute force test to establish whether it was the sum of two pyramidal numbers. If not, the program tested whether it was the sum of three of them, then four, and finally five, until it first got an answer. About 45% of the integers are expressible as the sum of three pyramidal numbers. Most of the remaining 55% require the sum of four, and usually each of these can be represented in many different ways. Only 241 integers are known to require the sum of five pyramidal numbers, the largest being 343,867. For about half of the n numbers, this algorithm presumably went through all of the three-tests and at least

[2]Why $n^{1/3}$? Recall that pyramidal numbers are of the form $(m^3 - m)/6$. The largest m such that the resulting number is at most n is roughly $\sqrt[3]{6n}$, so there are $\Theta(n^{1/3})$ such numbers.

some of the four-tests before terminating. Thus, the total time for this algorithm would be at least $O(n \times (n^{1/3})^3) = O(n^2)$ time, where $n = 1{,}000{,}000{,}000$. No wonder his program cried "Uncle."

Anything that was going to do significantly better on a problem this large had to avoid explicitly testing all triples. For each value of k, we were seeking the smallest set of pyramidal numbers that add up to exactly to k. This problem is called the *knapsack problem*, and is discussed in Section 13.10 (page 427). In our case, the weights are the set of pyramidal numbers no greater than n, with an additional constraint that the knapsack holds exactly k items.

A standard approach to solving knapsack precomputes the sum of smaller subsets of the items for use in computing larger subsets. If we have a table of all sums of two numbers and want to know whether k is expressible as the sum of three numbers, we can ask whether k is expressible as the sum of a single number plus a number in this two-table.

Therefore I needed a table of all integers less than n that can be expressed as the sum of two of the 1,818 pyramidal numbers less than 1,000,000,000. There can be at most $1,818^2 = 3{,}305{,}124$ of them. Actually, there are only about half this many after we eliminate duplicates and any sum bigger than our target. Building a sorted array storing these numbers would be no big deal. Let's call this sorted data structure of all pair-sums the *two*-table.

To find the minimum decomposition for a given k, I would first check whether it was one of the 1,818 pyramidal numbers. If not, I would then check whether k was in the sorted table of the sums of two pyramidal numbers. To see whether k was expressible as the sum of three such numbers, all I had to do was check whether $k - p[i]$ was in the *two*-table for $1 \leq i \leq 1,818$. This could be done quickly using binary search. To see whether k was expressible as the sum of four pyramidal numbers, I had to check whether $k - two[i]$ was in the two-table for any $1 \leq i \leq |two|$. However, since almost every k was expressible in many ways as the sum of four pyramidal numbers, this test would terminate quickly, and the total time taken would be dominated by the cost of the threes. Testing whether k was the sum of three pyramidal numbers would take $O(n^{1/3} \lg n)$. Running this on each of the n integers gives an $O(n^{4/3} \lg n)$ algorithm for the complete job. Comparing this to his $O(n^2)$ algorithm for $n = 1{,}000{,}000{,}000$ suggested that my algorithm was a cool *30,000* times faster than his original!

My first attempt to code this solved up to $n = 1,000,000$ on my ancient Sparc ELC in about 20 minutes. From here, I experimented with different data structures to represent the sets of numbers and different algorithms to search these tables. I tried using hash tables and bit vectors instead of sorted arrays, and experimented with variants of binary search such as interpolation search (see Section 14.2 (page 441)). My reward for this work was solving up to n =1,000,000 in under three minutes, a factor of six improvement over my original program.

With the real thinking done, I worked to tweak a little more performance out of the program. I avoided doing a sum-of-four computation on any k when $k - 1$ was

the sum-of-three, since 1 is a pyramidal number, saving about 10% of the total run time using this trick alone. Finally, I got out my profiler and tried some low-level tricks to squeeze a little more performance out of the code. For example, I saved another 10% by replacing a single procedure call with in line code.

At this point, I turned the code over to the supercomputer guy. What he did with it is a depressing tale, which is reported in Section 7.10 (page 268).

In writing up this war story, I went back to rerun my program more than ten years later. On my desktop SunBlade 150, getting to 1,000,000 now took 27.0 seconds using the gcc compiler without turning on any compiler optimization. With Level 4 optimization, the job ran in just 14.0 seconds—quite a tribute to the quality of the optimizer. The run time on my desktop machine improved by a factor of about three over the four-year period prior to my first edition of this book, with an additional 5.3 times over the last 11 years. These speedups are probably typical for most desktops.

The primary lesson of this war story is to show the enormous potential for algorithmic speedups, as opposed to the fairly limited speedup obtainable via more expensive hardware. I sped his program up by about 30,000 times. His million-dollar computer had 16 processors, each reportedly five times faster on integer computations than the $3,000 machine on my desk. That gave a maximum potential speedup of less than 100 times. Clearly, the algorithmic improvement was the big winner here, as it is certain to be in any sufficiently large computation.

2.9 Advanced Analysis (*)

Ideally, each of us would be fluent in working with the mathematical techniques of asymptotic analysis. And ideally, each of us would be rich and good looking as well.

In this section I will survey the major techniques and functions employed in advanced algorithm analysis. I consider this optional material—it will not be used elsewhere in the textbook section of this book. That said, it will make some of the complexity functions reported in the Hitchhiker's Guide far less mysterious.

2.9.1 Esoteric Functions

The bread-and-butter classes of complexity functions were presented in Section 2.3.1 (page 39). More esoteric functions also make appearances in advanced algorithm analysis. Although we will not see them much in this book, it is still good business to know what they mean and where they come from:

- *Inverse Ackerman's function* $f(n) = \alpha(n)$ – This function arises in the detailed analysis of several algorithms, most notably the Union-Find data structure discussed in Section 6.1.3 (page 198).

  The exact definition of this function and why it arises will not be discussed further. It is sufficient to think of it as geek talk for the slowest-growing

complexity function. Unlike the constant function $f(n) = 1$, it eventually gets to infinity as $n \to \infty$, but it certainly takes its time about it. The value of $\alpha(n) < 5$ for any value of n that can be written in this physical universe.

- $f(n) = \log \log n$ – The "log log" function is just that—the logarithm of the logarithm of n. One natural example of how it might arise is doing a binary search on a sorted array of only $\lg n$ items.

- $f(n) = \log n / \log \log n$ – This function grows a little slower than $\log n$ because it is divided by an even slower growing function.

  To see where this arises, consider an n-leaf rooted tree of degree d. For binary trees, i.e. when $d = 2$, the height h is given

  $$n = 2^h \to h = \lg n$$

  by taking the logarithm of both sides of the equation. Now consider the height of such a tree when the degree $d = \log n$. Then

  $$n = (\log n)^h \to h = \log n / \log \log n$$

- $f(n) = \log^2 n$ – This is the product of log functions—i.e., $(\log n) \times (\log n)$. It might arise if we wanted to count the bits looked at in doing a binary search on n items, each of which was an integer from 1 to (say) n^2. Each such integer requires a $\lg(n^2) = 2 \lg n$ bit representation, and we look at $\lg n$ of them, for a total of $2 \lg^2 n$ bits.

  The "log squared" function typically arises in the design of intricate nested data structures, where each node in (say) a binary tree represents another data structure, perhaps ordered on a different key.

- $f(n) = \sqrt{n}$ – The square root is not so esoteric, but represents the class of "sublinear polynomials" since $\sqrt{n} = n^{1/2}$. Such functions arise in building d-dimensional grids that contain n points. A $\sqrt{n} \times \sqrt{n}$ square has area n, and an $n^{1/3} \times n^{1/3} \times n^{1/3}$ cube has volume n. In general, a d-dimensional hypercube of length $n^{1/d}$ on each side has volume d.

- $f(n) = n^{(1+\epsilon)}$ – Epsilon (ϵ) is the mathematical symbol to denote a constant that can be made arbitrarily small but never quite goes away.

  It arises in the following way. Suppose I design an algorithm that runs in $2^c n^{(1+1/c)}$ time, and I get to pick whichever c I want. For $c = 2$, this is $4n^{3/2}$ or $O(n^{3/2})$. For $c = 3$, this is $8n^{4/3}$ or $O(n^{4/3})$, which is better. Indeed, the exponent keeps getting better the larger I make c.

  The problem is that I cannot make c arbitrarily large before the 2^c term begins to dominate. Instead, we report this algorithm as running in $O(n^{1+\epsilon})$, and leave the best value of ϵ to the beholder.

2.9.2 Limits and Dominance Relations

The dominance relation between functions is a consequence of the theory of limits, which you may recall from Calculus. We say that $f(n)$ *dominates* $g(n)$ if $\lim_{n\to\infty} g(n)/f(n) = 0$.

Let's see this definition in action. Suppose $f(n) = 2n^2$ and $g(n) = n^2$. Clearly $f(n) > g(n)$ for all n, but it does not dominate since

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} n^2/2n^2 = \lim_{n\to\infty} 1/2 \neq 0$$

This is to be expected because both functions are in the class $\Theta(n^2)$. What about $f(n) = n^3$ and $g(n) = n^2$? Since

$$\lim_{n\to\infty} g(n)/f(n) = \lim_{n\to\infty} n^2/n^3 = \lim_{n\to\infty} 1/n = 0$$

the higher-degree polynomial dominates. This is true for any two polynomials, namely that n^a dominates n^b if $a > b$ since

$$\lim_{n\to\infty} n^b/n^a = \lim_{n\to\infty} n^{b-a} \to 0$$

Thus $n^{1.2}$ dominates $n^{1.1999999}$.

Now consider two exponential functions, say $f(n) = 3^n$ and $g(n) = 2^n$. Since

$$\lim_{n\to\infty} g(n)/f(n) = 2^n/3^n = \lim_{n\to\infty} (2/3)^n = 0$$

the exponential with the higher base dominates.

Our ability to prove dominance relations from scratch depends upon our ability to prove limits. Let's look at one important pair of functions. Any polynomial (say $f(n) = n^\epsilon$) dominates logarithmic functions (say $g(n) = \lg n$). Since $n = 2^{\lg n}$,

$$f(n) = (2^{\lg n})^\epsilon = 2^{\epsilon \lg n}$$

Now consider

$$\lim_{n\to\infty} g(n)/f(n) = \lg n/2^{\epsilon \lg n}$$

In fact, this does go to 0 as $n \to \infty$.

Take-Home Lesson: By interleaving the functions here with those of Section 2.3.1 (page 39), we see where everything fits into the dominance pecking order:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg$$
$$\log^2 n \gg \log n \gg \log n/\log\log n \gg \log\log n \gg \alpha(n) \gg 1$$

Chapter Notes

Most other algorithm texts devote considerably more efforts to the formal analysis of algorithms than we have here, and so we refer the theoretically-inclined reader

elsewhere for more depth. Algorithm texts more heavily stressing analysis include [CLRS01, KT06].

The book *Concrete Mathematics* by Knuth, Graham, and Patashnik [GKP89] offers an interesting and thorough presentation of mathematics for the analysis of algorithms. Niven and Zuckerman [NZ80] is an nice introduction to number theory, including Waring's problem, discussed in the war story.

The notion of dominance also gives rise to the "Little Oh" notation. We say that $f(n) = o(g(n))$ iff $g(n)$ dominates $f(n)$. Among other things, the Little Oh proves useful for asking questions. Asking for an $o(n^2)$ algorithm means you want one that is better than quadratic in the worst case—and means you would be willing to settle for $O(n^{1.999} \log^2 n)$.

2.10 Exercises

Program Analysis

2-1. *[3]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using the Big Oh notation.

```
function mystery(n)
    r := 0
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            for k := 1 to j do
                r := r + 1
    return(r)
```

2-2. *[3]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using Big Oh notation.

```
function pesky(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                r := r + 1
    return(r)
```

2-3. *[5]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using Big Oh notation.

```
function prestiferous(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                for l := 1 to i + j − k do
```

```
                    r := r + 1
            return(r)
```

2-4. *[8]* What value is returned by the following function? Express your answer as a function of n. Give the worst-case running time using Big Oh notation.

```
function conundrum(n)
    r := 0
    for i := 1 to n do
        for j := i + 1 to n do
            for k := i + j − 1 to n do
                r := r + 1
    return(r)
```

2-5. *[5]* Suppose the following algorithm is used to evaluate the polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

```
p := a_0;
xpower := 1;
for i := 1 to n do
        xpower := x * xpower;
        p := p + a_i * xpower
end
```

(a) How many multiplications are done in the worst-case? How many additions?

(b) How many multiplications are done on the average?

(c) Can you improve this algorithm?

2-6. *[3]* Prove that the following algorithm for computing the maximum value in an array $A[1..n]$ is correct.

```
function max(A)
                    m := A[1]
                    for i := 2 to n do
                                if A[i] > m then m := A[i]
                    return (m)
```

Big Oh

2-7. *[3]* True or False?

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

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

2-8. *[3]* For each of the following pairs of functions, either $f(n)$ is in $O(g(n))$, $f(n)$ is in $\Omega(g(n))$, or $f(n) = \Theta(g(n))$. Determine which relationship is correct and briefly explain why.

(a) $f(n) = \log n^2$; $g(n) = \log n + 5$

(b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$

(c) $f(n) = \log^2 n$; $g(n) = \log n$

(d) $f(n) = n$; $g(n) = \log^2 n$

(e) $f(n) = n \log n + n$; $g(n) = \log n$

(f) $f(n) = 10$; $g(n) = \log 10$

(g) $f(n) = 2^n$; $g(n) = 10n^2$

(h) $f(n) = 2^n$; $g(n) = 3^n$

2-9. *[3]* For each of the following pairs of functions $f(n)$ and $g(n)$, determine whether $f(n) = O(g(n))$, $g(n) = O(f(n))$, or both.

(a) $f(n) = (n^2 - n)/2$, $g(n) = 6n$

(b) $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$

(c) $f(n) = n \log n$, $g(n) = n\sqrt{n}/2$

(d) $f(n) = n + \log n$, $g(n) = \sqrt{n}$

(e) $f(n) = 2(\log n)^2$, $g(n) = \log n + 1$

(f) $f(n) = 4n \log n + n$, $g(n) = (n^2 - n)/2$

2-10. *[3]* Prove that $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.

2-11. *[3]* Prove that $n^2 = O(2^n)$.

2-12. *[3]* For each of the following pairs of functions $f(n)$ and $g(n)$, give an appropriate positive constant c such that $f(n) \leq c \cdot g(n)$ for all $n > 1$.

(a) $f(n) = n^2 + n + 1$, $g(n) = 2n^3$

(b) $f(n) = n\sqrt{n} + n^2$, $g(n) = n^2$

(c) $f(n) = n^2 - n + 1$, $g(n) = n^2/2$

2-13. *[3]* Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

2-14. *[3]* Prove that if $f_1(N) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

2-15. *[3]* Prove that if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

2-16. *[5]* Prove for all $k \geq 1$ and all sets of constants $\{a_k, a_{k-1}, \ldots, a_1, a_0\} \in R$,

$$a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 = O(n^k)$$

2-17. *[5]* Show that for any real constants a and b, $b > 0$

$$(n + a)^b = \Theta(n^b)$$

2-18. *[5]* List the functions below from the lowest to the highest order. If any two or more are of the same order, indicate which.

n	2^n	$n \lg n$	$\ln n$
$n - n^3 + 7n^5$	$\lg n$	$\sqrt{n}$	e^n
$n^2 + \lg n$	n^2	2^{n-1}	$\lg \lg n$
n^3	$(\lg n)^2$	$n!$	$n^{1+\varepsilon}$ where $0 < \varepsilon < 1$

2-19. *[5]* List the functions below from the lowest to the highest order. If any two or more are of the same order, indicate which.

$\sqrt{n}$	n	2^n
$n \log n$	$n - n^3 + 7n^5$	$n^2 + \log n$
n^2	n^3	$\log n$
$n^{\frac{1}{3}} + \log n$	$(\log n)^2$	$n!$
$\ln n$	$\frac{n}{\log n}$	$\log \log n$
$(1/3)^n$	$(3/2)^n$	6

2-20. *[5]* Find two functions $f(n)$ and $g(n)$ that satisfy the following relationship. If no such f and g exist, write "None."

(a) $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$

(b) $f(n) = \Theta(g(n))$ and $f(n) = o(g(n))$

(c) $f(n) = \Theta(g(n))$ and $f(n) \neq O(g(n))$

(d) $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$

2-21. *[5]* True or False?

(a) $2n^2 + 1 = O(n^2)$

(b) $\sqrt{n} = O(\log n)$

(c) $\log n = O(\sqrt{n})$

(d) $n^2(1 + \sqrt{n}) = O(n^2 \log n)$

(e) $3n^2 + \sqrt{n} = O(n^2)$

(f) $\sqrt{n} \log n = O(n)$

(g) $\log n = O(n^{-1/2})$

2-22. *[5]* For each of the following pairs of functions $f(n)$ and $g(n)$, state whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$, or none of the above.

(a) $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$

(b) $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$

(c) $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$

2-23. *[3]* For each of these questions, briefly explain your answer.

(a) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?

(b) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?

(c) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?

(d) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?

(e) Is the function $f(n) = \Theta(n^2)$, where $f(n) = 100n^2$ for even n and $f(n) = 20n^2 - n\log_2 n$ for odd n?

2-24. *[3]* For each of the following, answer *yes*, *no*, or *can't tell*. Explain your reasoning.

(a) Is $3^n = O(2^n)$?

(b) Is $\log 3^n = O(\log 2^n)$?

(c) Is $3^n = \Omega(2^n)$?

(d) Is $\log 3^n = \Omega(\log 2^n)$?

2-25. *[5]* For each of the following expressions $f(n)$ find a simple $g(n)$ such that $f(n) = \Theta(g(n))$.

(a) $f(n) = \sum_{i=1}^{n} \frac{1}{i}$.

(b) $f(n) = \sum_{i=1}^{n} \lceil \frac{1}{i} \rceil$.

(c) $f(n) = \sum_{i=1}^{n} \log i$.

(d) $f(n) = \log(n!)$.

2-26. *[5]* Place the following functions into increasing asymptotic order.

$f_1(n) = n^2 \log_2 n$, $f_2(n) = n(\log_2 n)^2$, $f_3(n) = \sum_{i=0}^{n} 2^i$, $f_4(n) = \log_2(\sum_{i=0}^{n} 2^i)$.

2-27. *[5]* Place the following functions into increasing asymptotic order. If two or more of the functions are of the same asymptotic order then indicate this.

$f_1(n) = \sum_{i=1}^{n} \sqrt{i}$, $f_2(n) = (\sqrt{n}) \log n$, $f_3(n) = n\sqrt{\log n}$, $f_4(n) = 12n^{\frac{3}{2}} + 4n$,

2-28. *[5]* For each of the following expressions $f(n)$ find a simple $g(n)$ such that $f(n) = \Theta(g(n))$. (You should be able to prove your result by exhibiting the relevant parameters, but this is not required for the homework.)

(a) $f(n) = \sum_{i=1}^{n} 3i^4 + 2i^3 - 19i + 20$.

(b) $f(n) = \sum_{i=1}^{n} 3(4^i) + 2(3^i) - i^{19} + 20$.

(c) $f(n) = \sum_{i=1}^{n} 5^i + 3^{2i}$.

2-29. *[5]* Which of the following are true?

(a) $\sum_{i=1}^{n} 3^i = \Theta(3^{n-1})$.

(b) $\sum_{i=1}^{n} 3^i = \Theta(3^n)$.

(c) $\sum_{i=1}^{n} 3^i = \Theta(3^{n+1})$.

2-30. *[5]* For each of the following functions f find a simple function g such that $f(n) = \Theta(g(n))$.

(a) $f_1(n) = (1000)2^n + 4^n$.

(b) $f_2(n) = n + n\log n + \sqrt{n}$.

(c) $f_3(n) = \log(n^{20}) + (\log n)^{10}$.

(d) $f_4(n) = (0.99)^n + n^{100}$.

2-31. *[5]* For each pair of expressions (A, B) below, indicate whether A is O, o, Ω, ω, or Θ of B. Note that zero, one or more of these relations may hold for a given pair; list all correct ones.

	A	B
(a)	n^{100}	2^n
(b)	$(\lg n)^{12}$	$\sqrt{n}$
(c)	$\sqrt{n}$	$n^{\cos(\pi n/8)}$
(d)	10^n	100^n
(e)	$n^{\lg n}$	$(\lg n)^n$
(f)	$\lg(n!)$	$n \lg n$

Summations

2-32. *[5]* Prove that:

$$1^2 - 2^2 + 3^2 - 4^2 + \ldots + (-1)^{k-1}k^2 = (-1)^{k-1}k(k+1)/2$$

2-33. *[5]* Find an expression for the sum of the ith row of the following triangle, and prove its correctness. Each entry is the sum of the three entries directly above it. All non existing entries are considered 0.

				1				
			1	1	1			
		1	2	3	2	1		
	1	3	6	7	6	3	1	
1	4	10	16	19	16	10	4	1

2-34. *[3]* Assume that Christmas has n days. Exactly how many presents did my "true love" send me? (Do some research if you do not understand this question.)

2-35. *[5]* Consider the following code fragment.

```
for i=1 to n do
    for j=i to 2*i do
        output ‘‘foobar’’
```

Let $T(n)$ denote the number of times 'foobar' is printed as a function of n.

a. Express $T(n)$ as a summation (actually two nested summations).

b. Simplify the summation. Show your work.

2-36. *[5]* Consider the following code fragment.

```
for i=1 to n/2 do
    for j=i to n-i do
        for k=1 to j do
output ‘‘foobar’’
```

Assume n is even. Let $T(n)$ denote the number of times 'foobar' is printed as a function of n.

(a) Express $T(n)$ as three nested summations.

(b) Simplify the summation. Show your work.

2-37. *[6]* When you first learned to multiply numbers, you were told that $x \times y$ means adding x a total of y times, so $5 \times 4 = 5+5+5+5 = 20$. What is the time complexity of multiplying two n-digit numbers in base b (people work in base 10, of course, while computers work in base 2) using the repeated addition method, as a function of n and b. Assume that single-digit by single-digit addition or multiplication takes $O(1)$ time. (Hint: how big can y be as a function of n and b?)

2-38. *[6]* In grade school, you learned to multiply long numbers on a digit-by-digit basis, so that $127 \times 211 = 127 \times 1 + 127 \times 10 + 127 \times 200 = 26,397$. Analyze the time complexity of multiplying two n-digit numbers with this method as a function of n (assume constant base size). Assume that single-digit by single-digit addition or multiplication takes $O(1)$ time.

Logarithms

2-39. *[5]* Prove the following identities on logarithms:

(a) Prove that $\log_a(xy) = \log_a x + \log_a y$

(b) Prove that $\log_a x^y = y \log_a x$

(c) Prove that $\log_a x = \frac{\log_b x}{\log_b a}$

(d) Prove that $x^{\log_b y} = y^{\log_b x}$

2-40. *[3]* Show that $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$

2-41. *[3]* Prove that that the binary representation of $n \geq 1$ has $\lfloor \lg_2 n \rfloor + 1$ bits.

2-42. *[5]* In one of my research papers I give a comparison-based sorting algorithm that runs in $O(n \log(\sqrt{n}))$. Given the existence of an $\Omega(n \log n)$ lower bound for sorting, how can this be possible?

Interview Problems

2-43. *[5]* You are given a set S of n numbers. You must pick a subset S' of k numbers from S such that the probability of each element of S occurring in S' is equal (i.e., each is selected with probability k/n). You may make only one pass over the numbers. What if n is unknown?

2-44. *[5]* We have 1,000 data items to store on 1,000 nodes. Each node can store copies of exactly three different items. Propose a replication scheme to minimize data loss as nodes fail. What is the expected number of data entries that get lost when three random nodes fail?

2-45. *[5]* Consider the following algorithm to find the minimum element in an array of numbers $A[0, \ldots, n]$. One extra variable *tmp* is allocated to hold the current minimum value. Start from A[0]; "tmp" is compared against $A[1]$, $A[2]$, $\ldots$, $A[N]$ in order. When $A[i] < tmp$, $tmp = A[i]$. What is the expected number of times that the assignment operation $tmp = A[i]$ is performed?

2-46. *[5]* You have a 100-story building and a couple of marbles. You must identify the lowest floor for which a marble will break if you drop it from this floor. How fast can you find this floor if you are given an infinite supply of marbles? What if you have only two marbles?

2-47. *[5]* You are given 10 bags of gold coins. Nine bags contain coins that each weigh 10 grams. One bag contains all false coins that weigh one gram less. You must identify this bag in just one weighing. You have a digital balance that reports the weight of what is placed on it.

2-48. *[5]* You have eight balls all of the same size. Seven of them weigh the same, and one of them weighs slightly more. How can you find the ball that is heavier by using a balance and only two weighings?

2-49. *[5]* Suppose we start with n companies that eventually merge into one big company. How many different ways are there for them to merge?

2-50. *[5]* A *Ramanujam number* can be written two different ways as the sum of two cubes—i.e. , there exist distinct a, b, c, and d such that $a^3 + b^3 = c^3 + d^3$. Generate all Ramanujam numbers where $a, b, c, d < n$.

2-51. *[7]* Six pirates must divide $300 dollars among themselves. The division is to proceed as follows. The senior pirate proposes a way to divide the money. Then the pirates vote. If the senior pirate gets at least half the votes he wins, and that division remains. If he doesn't, he is killed and then the next senior-most pirate gets a chance to do the division. Now you have to tell what will happen and why (i.e. , how many pirates survive and how the division is done)? All the pirates are intelligent and the first priority is to stay alive and the next priority is to get as much money as possible.

2-52. *[7]* Reconsider the pirate problem above, where only one indivisible dollar is to be divided. Who gets the dollar and how many are killed?

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

2-1. "Primary Arithmetic" – Programming Challenges 110501, UVA Judge 10035.

2-2. "A Multiplication Game" – Programming Challenges 110505, UVA Judge 847.

2-3. "Light, More Light" – Programming Challenges 110701, UVA Judge 10110.

3

Data Structures

Changing a data structure in a slow program can work the same way an organ transplant does in a sick patient. Important classes of *abstract data types* such as containers, dictionaries, and priority queues, have many different but functionally equivalent *data structures* that implement them. Changing the data structure does not change the correctness of the program, since we presumably replace a correct implementation with a different correct implementation. However, the new implementation of the data type realizes different tradeoffs in the time to execute various operations, so the total performance can improve dramatically. Like a patient in need of a transplant, only one part might need to be replaced in order to fix the problem.

But it is better to be born with a good heart than have to wait for a replacement. The maximum benefit from good data structures results from designing your program around them in the first place. We assume that the reader has had some previous exposure to elementary data structures and pointer manipulation. Still, data structure (CS II) courses these days focus more on data abstraction and object orientation than the nitty-gritty of how structures should be represented in memory. We will review this material to make sure you have it down.

In data structures, as with most subjects, it is more important to really understand the basic material than have exposure to more advanced concepts. We will focus on each of the three fundamental abstract data types (containers, dictionaries, and priority queues) and see how they can be implemented with arrays and lists. Detailed discussion of the tradeoffs between more sophisticated implementations is deferred to the relevant catalog entry for each of these data types.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_3,

3.1 Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either *contiguous* or *linked*, depending upon whether they are based on arrays or pointers:

- *Contiguously-allocated structures* are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- *Linked data structures* are composed of distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

In this section, we review the relative advantages of contiguous and linked data structures. These tradeoffs are more subtle than they appear at first glance, so I encourage readers to stick with me here even if you may be familiar with both types of structures.

3.1.1 Arrays

The *array* is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.

A good analogy likens an array to a street full of houses, where each array element is equivalent to a house, and the index is equivalent to the house number. Assuming all the houses are equal size and numbered sequentially from 1 to n, we can compute the exact position of each house immediately from its address.[1]

Advantages of contiguously-allocated arrays include:

- *Constant-time access given the index* – Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index.
- *Space efficiency* – Arrays consist purely of data, so no space is wasted with links or other formatting information. Further, end-of-record information is not needed because arrays are built from fixed-size records.
- *Memory locality* – A common programming idiom involves iterating through all the elements of a data structure. Arrays are good for this because they exhibit excellent memory locality. Physical continuity between successive data accesses helps exploit the high-speed *cache memory* on modern computer architectures.

The downside of arrays is that we cannot adjust their size in the middle of a program's execution. Our program will fail soon as we try to add the $(n +$

[1] Houses in Japanese cities are traditionally numbered in the order they were built, not by their physical location. This makes it extremely difficult to locate a Japanese address without a detailed map.

1)st customer, if we only allocate room for n records. We can compensate by allocating extremely large arrays, but this can waste space, again restricting what our programs can do.

Actually, we *can* efficiently enlarge arrays as we need them, through the miracle of *dynamic arrays*. Suppose we start with an array of size 1, and double its size from m to $2m$ each time we run out of space. This doubling process involves allocating a new contiguous array of size $2m$, copying the contents of the old array to the lower half of the new one, and returning the space used by the old array to the storage allocation system.

The apparent waste in this procedure involves the recopying of the old contents on each expansion. How many times might an element have to be recopied after a total of n insertions? Well, the first inserted element will have been recopied when the array expands after the first, second, fourth, eighth, ... insertions. It will take $\log_2 n$ doublings until the array gets to have n positions. However, most elements do not suffer much upheaval. Indeed, the $(n/2+1)$st through nth elements will move at most once and might never have to move at all.

If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given by

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

Thus, each of the n elements move only two times on average, and the total work of managing the dynamic array is the same $O(n)$ as it would have been if a single array of sufficient size had been allocated in advance!

The primary thing lost using dynamic arrays is the guarantee that each array access takes constant time *in the worst case*. Now all the queries will be fast, except for those relatively few queries triggering array doubling. What we get instead is a promise that the nth array access will be completed quickly enough that the *total* effort expended so far will still be $O(n)$. Such *amortized* guarantees arise frequently in the analysis of data structures.

3.1.2 Pointers and Linked Structures

Pointers are the connections that hold the pieces of linked structures together. Pointers represent the address of a location in memory. A variable storing a pointer to a given data item can provide more freedom than storing a copy of the item itself. A cell-phone number can be thought of as a pointer to its owner as they move about the planet.

Pointer syntax and power differ significantly across programming languages, so we begin with a quick review of pointers in C language. A pointer `p` is assumed to

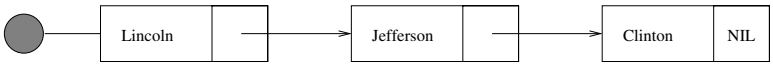

Figure 3.1: Linked list example showing data and pointer fields

give the address in memory where a particular chunk of data is located.[2] Pointers in C have types declared at compiler time, denoting the data type of the items they can point to. We use `*p` to denote the item that is pointed to by pointer `p`, and `&x` to denote the address (i.e. , pointer) of a particular variable `x`. A special `NULL` pointer value is used to denote structure-terminating or unassigned pointers.

All linked data structures share certain properties, as revealed by the following linked list type declaration:

```
typedef struct list {
        item_type item;                 /* data item */
        struct list *next;              /* point to successor */
} list;
```

In particular:

- Each node in our data structure (here `list`) contains one or more data fields (here `item`) that retain the data that we need to store.
- Each node contains a pointer field to at least one other node (here `next`). This means that much of the space used in linked data structures has to be devoted to pointers, not data.
- Finally, we need a pointer to the head of the structure, so we know where to access it.

The list is the simplest linked structure. The three basic operations supported by lists are searching, insertion, and deletion. In *doubly-linked lists*, each node points both to its predecessor and its successor element. This simplifies certain operations at a cost of an extra pointer field per node.

Searching a List

Searching for item x in a linked list can be done iteratively or recursively. We opt for recursively in the implementation below. If x is in the list, it is either the first element or located in the smaller rest of the list. Eventually, we reduce the problem to searching in an empty list, which clearly cannot contain x.

[2] C permits direct manipulation of memory addresses in ways which may horrify Java programmers, but we will avoid doing any such tricks.

```
list *search_list(list *l, item_type x)
{
        if (l == NULL) return(NULL);

        if (l->item == x)
                return(l);
        else
                return( search_list(l->next, x) );
}
```

Insertion into a List

Insertion into a singly-linked list is a nice exercise in pointer manipulation, as shown below. Since we have no need to maintain the list in any particular order, we might as well insert each new item in the simplest place. Insertion at the beginning of the list avoids any need to traverse the list, but does require us to update the pointer (denoted `l`) to the head of the data structure.

```
void insert_list(list **l, item_type x)
{
        list *p;                        /* temporary pointer */

        p = malloc( sizeof(list) );
        p->item = x;
        p->next = *l;
        *l = p;
}
```

Two C-isms to note. First, the `malloc` function allocates a chunk of memory of sufficient size for a new node to contain `x`. Second, the funny double star (`**l`) denotes that `l` is a *pointer to a pointer* to a list node. Thus the last line, `*l=p;` copies `p` to the place pointed to `l`, which is the external variable maintaining access to the head of the list.

Deletion From a List

Deletion from a linked list is somewhat more complicated. First, we must find a pointer to the *predecessor* of the item to be deleted. We do this recursively:

```
list *predecessor_list(list *l, item_type x)
{
        if ((l == NULL) || (l->next == NULL)) {
                printf("Error: predecessor sought on null list.\n");
                return(NULL);
        }

        if ((l->next)->item == x)
                return(l);
        else
                return( predecessor_list(l->next, x) );
}
```

The predecessor is needed because it points to the doomed node, so its `next` pointer must be changed. The actual deletion operation is simple, once ruling out the case that the to-be-deleted element does not exist. Special care must be taken to reset the pointer to the head of the list (`l`) when the first element is deleted:

```
delete_list(list **l, item_type x)
{
        list *p;                        /* item pointer */
        list *pred;                     /* predecessor pointer */
        list *search_list(), *predecessor_list();

        p = search_list(*l,x);
        if (p != NULL) {
                pred = predecessor_list(*l,x);
                if (pred == NULL)       /* splice out out list */
                        *l = p->next;
                else
                        pred->next = p->next;

                free(p);                /* free memory used by node */
        }
}
```

C language requires explicit deallocation of memory, so we must `free` the deleted node after we are finished with it to return the memory to the system.

3.1.3 Comparison

The relative advantages of linked lists over static arrays include:

- Overflow on linked structures can never occur unless the memory is actually full.

- Insertions and deletions are *simpler* than for contiguous (array) lists.
- With large records, moving pointers is easier and faster than moving the items themselves.

while the relative advantages of arrays include:

- Linked structures require extra space for storing pointer fields.
- Linked lists do not allow efficient random access to items.
- Arrays allow better memory locality and cache performance than random pointer jumping.

Take-Home Lesson: Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

One final thought about these fundamental structures is that they can be thought of as recursive objects:

- *Lists* – Chopping the first element off a linked list leaves a smaller linked list. This same argument works for strings, since removing characters from string leaves a string. Lists are recursive objects.
- *Arrays* – Splitting the first k elements off of an n element array gives two smaller arrays, of size k and $n-k$, respectively. Arrays are recursive objects.

This insight leads to simpler list processing, and efficient divide-and-conquer algorithms such as quicksort and binary search.

3.2 Stacks and Queues

We use the term *container* to denote a data structure that permits storage and retrieval of data items *independent of content*. By contrast, dictionaries are abstract data types that retrieve based on key values or content, and will be discussed in Section 3.3 (page 72).

Containers are distinguished by the particular retrieval order they support. In the two most important types of containers, this retrieval order depends on the insertion order:

- *Stacks* – Support retrieval by last-in, first-out (LIFO) order. Stacks are simple to implement and very efficient. For this reason, stacks are probably the right container to use when retrieval order doesn't matter at all, such as when processing batch jobs. The *put* and *get* operations for stacks are usually called *push* and *pop*:

  - *Push(x,s)*: Insert item x at the top of stack s.
  - *Pop(s)*: Return (and remove) the top item of stack s.

  LIFO order arises in many real-world contexts. People crammed into a subway car exit in LIFO order. Food inserted into my refrigerator usually exits the same way, despite the incentive of expiration dates. Algorithmically, LIFO tends to happen in the course of executing recursive algorithms.

- *Queues* – Support retrieval in first in, first out (FIFO) order. This is surely the fairest way to control waiting times for services. You want the container holding jobs to be processed in FIFO order to minimize the *maximum* time spent waiting. Note that the *average* waiting time will be the same regardless of whether FIFO or LIFO is used. Many computing applications involve data items with infinite patience, which renders the question of maximum waiting time moot.

  Queues are somewhat trickier to implement than stacks and thus are most appropriate for applications (like certain simulations) where the order is important. The *put* and *get* operations for queues are usually called *enqueue* and *dequeue*.

  - *Enqueue(x,q)*: Insert item x at the back of queue q.
  - *Dequeue(q)*: Return (and remove) the front item from queue q.

  We will see queues later as the fundamental data structure controlling breadth-first searches in graphs.

Stacks and queues can be effectively implemented using either arrays or linked lists. The key issue is whether an upper bound on the size of the container is known in advance, thus permitting the use of a statically-allocated array.

3.3 Dictionaries

The *dictionary* data type permits access to data items by content. You stick an item into a dictionary so you can find it when you need it.

The primary operations of dictionary support are:

- *Search(D,k)* – Given a search key k, return a pointer to the element in dictionary D whose key value is k, if one exists.
- *Insert(D,x)* – Given a data item x, add it to the set in the dictionary D.
- *Delete(D,x)* – Given a pointer to a given data item x in the dictionary D, remove it from D.

Certain dictionary data structures also efficiently support other useful operations:

- *Max(D)* or *Min(D)* – Retrieve the item with the largest (or smallest) key from D. This enables the dictionary to serve as a priority queue, to be discussed in Section 3.5 (page 83).
- *Predecessor(D,k)* or *Successor(D,k)* – Retrieve the item from D whose key is immediately before (or after) k in sorted order. These enable us to iterate through the elements of the data structure.

Many common data processing tasks can be handled using these dictionary operations. For example, suppose we want to remove all duplicate names from a mailing list, and print the results in sorted order. Initialize an empty dictionary D, whose search key will be the record name. Now read through the mailing list, and for each record *search* to see if the name is already in D. If not, *insert* it into D. Once finished, we must extract the remaining names out of the dictionary. By starting from the first item *Min(D)* and repeatedly calling *Successor* until we obtain *Max(D)*, we traverse all elements in sorted order.

By defining such problems in terms of abstract dictionary operations, we avoid the details of the data structure's representation and focus on the task at hand.

In the rest of this section, we will carefully investigate simple dictionary implementations based on arrays and linked lists. More powerful dictionary implementations such as binary search trees (see Section 3.4 (page 77)) and hash tables (see Section 3.7 (page 89)) are also attractive options in practice. A complete discussion of different dictionary data structures is presented in the catalog in Section 12.1 (page 367). We encourage the reader to browse through the data structures section of the catalog to better learn what your options are.

Stop and Think: Comparing Dictionary Implementations (I)

Problem: What are the asymptotic worst-case running times for each of the seven fundamental dictionary operations (search, insert, delete, successor, predecessor, minimum, and maximum) when the data structure is implemented as:

- An unsorted array.
- A sorted array.

Solution: This problem (and the one following it) reveal some of the inherent tradeoffs of data structure design. A given data representation may permit efficient implementation of certain operations at the cost that other operations are expensive.

In addition to the array in question, we will assume access to a few extra variables such as n—the number of elements currently in the array. Note that we must *maintain* the value of these variables in the operations where they change (e.g., insert and delete), and charge these operations the cost of this maintenance.

The basic dictionary operations can be implemented with the following costs on unsorted and sorted arrays, respectively:

Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$
Successor(L, x)	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(1)$

We must understand the implementation of each operation to see why. First, we discuss the operations when maintaining an *unsorted* array A.

- *Search* is implemented by testing the search key k against (potentially) each element of an unsorted array. Thus, search takes linear time in the worst case, which is when key k is not found in A.
- *Insertion* is implemented by incrementing n and then copying item x to the nth cell in the array, $A[n]$. The bulk of the array is untouched, so this operation takes constant time.
- *Deletion* is somewhat trickier, hence the superscript(*) in the table. The definition states that we are given a pointer x to the element to delete, so we need not spend any time searching for the element. But removing the xth element from the array A leaves a hole that must be filled. We could fill the hole by moving each of the elements $A[x+1]$ to $A[n]$ up one position, but this requires $\Theta(n)$ time when the first element is deleted. The following idea is better: just write over $A[x]$ with $A[n]$, and decrement n. This only takes constant time.
- The definition of the traversal operations, *Predecessor* and *Successor*, refer to the item appearing before/after x *in sorted order*. Thus, the answer is not simply $A[x-1]$ (or $A[x+1]$), because in an unsorted array an element's physical predecessor (successor) is not necessarily its logical predecessor (successor). Instead, the predecessor of $A[x]$ is the biggest element smaller than $A[x]$. Similarly, the successor of $A[x]$ is the smallest element larger than $A[x]$. Both require a sweep through all n elements of A to determine the winner.
- *Minimum* and *Maximum* are similarly defined with respect to sorted order, and so require linear sweeps to identify in an unsorted array.

Implementing a dictionary using a *sorted* array completely reverses our notions of what is easy and what is hard. Searches can now be done in $O(\log n)$ time, using binary search, because we know the median element sits in $A[n/2]$. Since the upper and lower portions of the array are also sorted, the search can continue recursively on the appropriate portion. The number of halvings of n until we get to a single element is $\lceil \lg n \rceil$.

The sorted order also benefits us with respect to the other dictionary retrieval operations. The minimum and maximum elements sit in $A[1]$ and $A[n]$, while the predecessor and successor to $A[x]$ are $A[x-1]$ and $A[x+1]$, respectively.

Insertion and deletion become more expensive, however, because making room for a new item or filling a hole may require moving many items arbitrarily. Thus both become linear-time operations. ∎

Take-Home Lesson: Data structure design must balance all the different operations it supports. The fastest data structure to support both operations A and B may well not be the fastest structure to support either operation A or B.

Stop and Think: Comparing Dictionary Implementations (II)

Problem: What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list.
- A doubly-linked unsorted list.
- A singly-linked sorted list.
- A doubly-linked sorted list.

Solution: Two different issues must be considered in evaluating these implementations: singly- vs. doubly-linked lists and sorted vs. unsorted order. Subtle operations are denoted with a superscript:

Dictionary operation	Singly unsorted	Double unsorted	Singly sorted	Doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

As with unsorted arrays, search operations are destined to be slow while maintenance operations are fast.

- *Insertion/Deletion* – The complication here is deletion from a singly-linked list. The definition of the *Delete* operation states we are given a pointer x to the item to be deleted. But what we *really* need is a pointer to the element pointing to x in the list, because that is the node that needs to be changed. We can do nothing without this list predecessor, and so must spend linear time searching for it on a singly-linked list. Doubly-linked lists avoid this problem, since we can immediately retrieve the list predecessor of x.

  Deletion is faster for sorted doubly-linked lists than sorted arrays, because splicing out the deleted element from the list is more efficient than filling the hole by moving array elements. The predecessor pointer problem again complicates deletion from singly-linked sorted lists.

- *Search* – Sorting provides less benefit for linked lists than it did for arrays. Binary search is no longer possible, because we can't access the median element without traversing all the elements before it. What sorted lists *do* provide is quick termination of unsuccessful searches, for if we have not found *Abbott* by the time we hit *Costello* we can deduce that he doesn't exist. Still, searching takes linear time in the worst case.

- *Traversal operations* – The predecessor pointer problem again complicates implementing *Predecessor*. The logical successor is equivalent to the node successor for both types of sorted lists, and hence can be implemented in constant time.

- *Maximum* – The maximum element sits at the tail of the list, which would normally require $\Theta(n)$ time to reach in either singly- or doubly-linked lists.

  However, we can maintain a separate pointer to the list tail, provided we pay the maintenance costs for this pointer on every insertion and deletion. The tail pointer can be updated in constant time on doubly-linked lists: on insertion check whether `last->next` still equals `NULL`, and on deletion set `last` to point to the list predecessor of `last` if the last element is deleted.

  We have no efficient way to find this predecessor for singly-linked lists. So why can we implement maximum in $\Theta(1)$ on singly-linked lists? The trick is to charge the cost to each deletion, which *already* took linear time. Adding an extra linear sweep to update the pointer does not harm the asymptotic complexity of *Delete*, while gaining us *Maximum* in constant time as a reward for clear thinking. ∎

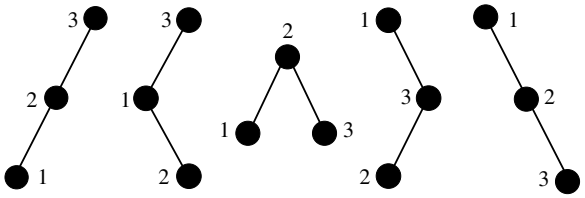

Figure 3.2: The five distinct binary search trees on three nodes

3.4 Binary Search Trees

We have seen data structures that allow fast search or flexible update, but not fast search *and* flexible update. Unsorted, doubly-linked lists supported insertion and deletion in $O(1)$ time but search took linear time in the worse case. Sorted arrays support binary search and logarithmic query times, but at the cost of linear-time update.

Binary search requires that we have fast access to *two elements*—specifically the median elements above and below the given node. To combine these ideas, we need a "linked list" with two pointers per node. This is the basic idea behind binary search trees.

A *rooted binary tree* is recursively defined as either being (1) empty, or (2) consisting of a node called the root, together with two rooted binary trees called the left and right subtrees, respectively. The order among "brother" nodes matters in rooted trees, so left is different from right. Figure 3.2 gives the shapes of the five distinct binary trees that can be formed on three nodes.

A binary *search* tree labels each node in a binary tree with a single key such that for any node labeled x, all nodes in the left subtree of x have keys $< x$ while all nodes in the right subtree of x have keys $> x$. This search tree labeling scheme is very special. For any binary tree on n nodes, and any set of n keys, there is *exactly* one labeling that makes it a binary search tree. The allowable labelings for three-node trees are given in Figure 3.2.

3.4.1 Implementing Binary Search Trees

Binary tree nodes have *left* and *right* pointer fields, an (optional) *parent* pointer, and a data field. These relationships are shown in Figure 3.3; a type declaration for the tree structure is given below:

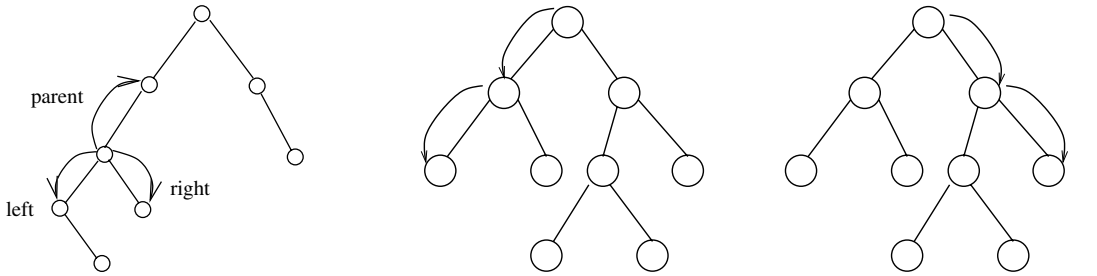

Figure 3.3: Relationships in a binary search tree (left). Finding the minimum (center) and maximum (right) elements in a binary search tree

```
typedef struct tree {
        item_type item;                 /* data item */
        struct tree *parent;            /* pointer to parent */
        struct tree *left;              /* pointer to left child */
        struct tree *right;             /* pointer to right child */
} tree;
```

The basic operations supported by binary trees are searching, traversal, insertion, and deletion.

Searching in a Tree

The binary search tree labeling uniquely identities where each key is located. Start at the root. Unless it contains the query key x, proceed either left or right depending upon whether x occurs before or after the root key. This algorithm works because both the left and right subtrees of a binary search tree are themselves binary search trees. This recursive structure yields the recursive search algorithm below:

```
tree *search_tree(tree *l, item_type x)
{
        if (l == NULL) return(NULL);

        if (l->item == x) return(l);

        if (x < l->item)
                return( search_tree(l->left, x) );
        else
                return( search_tree(l->right, x) );
}
```

This search algorithm runs in $O(h)$ time, where h denotes the height of the tree.

Finding Minimum and Maximum Elements in a Tree

Implementing the *find-minimum* operation requires knowing where the minimum element is in the tree. By definition, the smallest key must reside in the left subtree of the root, since all keys in the left subtree have values less than that of the root. Therefore, as shown in Figure 3.3, the minimum element must be the leftmost descendent of the root. Similarly, the maximum element must be the rightmost descendent of the root.

```
tree *find_minimum(tree *t)
{
        tree *min;                      /* pointer to minimum */

        if (t == NULL) return(NULL);

        min = t;
        while (min->left != NULL)
                min = min->left;
        return(min);
}
```

Traversal in a Tree

Visiting all the nodes in a rooted binary tree proves to be an important component of many algorithms. It is a special case of traversing all the nodes and edges in a graph, which will be the foundation of Chapter 5.

A prime application of tree traversal is listing the labels of the tree nodes. Binary search trees make it easy to report the labels in sorted order. By definition, all the keys smaller than the root must lie in the left subtree of the root, and all keys bigger than the root in the right subtree. Thus, visiting the nodes recursively in accord with such a policy produces an *in-order* traversal of the search tree:

```
void traverse_tree(tree *l)
{
        if (l != NULL) {
                traverse_tree(l->left);
                process_item(l->item);
                traverse_tree(l->right);
        }

}
```

Each item is processed once during the course of traversal, which runs in $O(n)$ time, where n denotes the number of nodes in the tree.

Alternate traversal orders come from changing the position of `process_item` relative to the traversals of the left and right subtrees. Processing the item first yields a *pre-order* traversal, while processing it last gives a *post-order* traversal. These make relatively little sense with search trees, but prove useful when the rooted tree represents arithmetic or logical expressions.

Insertion in a Tree

There is only one place to insert an item x into a binary search tree T where we know we can find it again. We must replace the `NULL` pointer found in T after an unsuccessful query for the key k.

This implementation uses recursion to combine the search and node insertion stages of key insertion. The three arguments to `insert_tree` are (1) a pointer `l` to the pointer linking the search subtree to the rest of the tree, (2) the key `x` to be inserted, and (3) a `parent` pointer to the parent node containing `l`. The node is allocated and linked in on hitting the `NULL` pointer. Note that we pass the *pointer* to the appropriate left/right pointer in the node during the search, so the assignment `*l = p;` links the new node into the tree:

```
insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p;                          /* temporary pointer */

    if (*l == NULL) {
           p = malloc(sizeof(tree)); /* allocate new node */
           p->item = x;
           p->left = p->right = NULL;
           p->parent = parent;
           *l = p;                    /* link into parent's record */
           return;
    }

    if (x < (*l)->item)
           insert_tree(&((*l)->left), x, *l);
    else
           insert_tree(&((*l)->right), x, *l);
}
```

Allocating the node and linking it in to the tree is a constant-time operation after the search has been performed in $O(h)$ time.

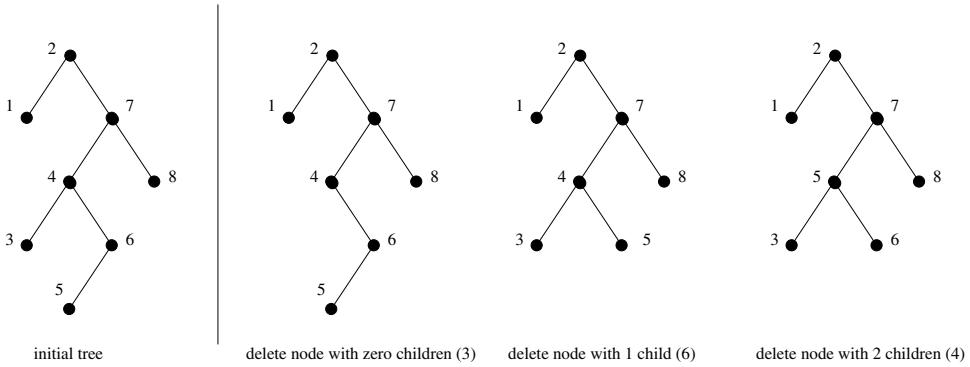

Figure 3.4: Deleting tree nodes with 0, 1, and 2 children

Deletion from a Tree

Deletion is somewhat trickier than insertion, because removing a node means appropriately linking its two descendant subtrees back into the tree somewhere else. There are three cases, illustrated in Figure 3.4. Leaf nodes have no children, and so may be deleted by simply clearing the pointer to the given node.

The case of the doomed node having one child is also straightforward. There is one parent and one grandchild, and we can link the grandchild directly to the parent without violating the in-order labeling property of the tree.

But what of a to-be-deleted node with two children? Our solution is to relabel this node with the key of its immediate successor in sorted order. This successor must be the smallest value in the right subtree, specifically the leftmost descendant in the right subtree (`p`). Moving this to the point of deletion results in a properly-labeled binary search tree, and reduces our deletion problem to physically removing a node with at most one child—a case that has been resolved above.

The full implementation has been omitted here because it looks a little ghastly, but the code follows logically from the description above.

The worst-case complexity analysis is as follows. Every deletion requires the cost of at most two search operations, each taking $O(h)$ time where h is the height of the tree, plus a constant amount of pointer manipulation.

3.4.2 How Good Are Binary Search Trees?

When implemented using binary search trees, all three dictionary operations take $O(h)$ time, where h is the height of the tree. The smallest height we can hope for occurs when the tree is perfectly balanced, where $h = \lceil \log n \rceil$. This is very good, but the tree must be perfectly balanced.

Our insertion algorithm puts each new item at a leaf node where it should have been found. This makes the shape (and more importantly height) of the tree a function of the order in which we insert the keys.

Unfortunately, bad things can happen when building trees through insertion. The data structure has no control over the order of insertion. Consider what happens if the user inserts the keys in sorted order. The operations `insert(`a`)`, followed by `insert(`b`)`, `insert(`c`)`, `insert(`d`)`, ... will produce a skinny linear height tree where only right pointers are used.

Thus binary trees can have heights ranging from $\lg n$ to n. But how tall are they on average? The average case analysis of algorithms can be tricky because we must carefully specify what we mean by *average*. The question is well defined if we consider each of the $n!$ possible insertion orderings equally likely and average over those. If so, we are in luck, because with high probability the resulting tree will have $O(\log n)$ height. This will be shown in Section 4.6 (page 123).

This argument is an important example of the power of *randomization*. We can often develop simple algorithms that offer good performance with high probability. We will see that a similar idea underlies the fastest known sorting algorithm, quicksort.

3.4.3 Balanced Search Trees

Random search trees are *usually* good. But if we get unlucky with our order of insertion, we can end up with a linear-height tree in the worst case. This worst case is outside of our direct control, since we must build the tree in response to the requests given by our potentially nasty user.

What would be better is an insertion/deletion procedure which *adjusts* the tree a little after each insertion, keeping it close enough to be balanced so the maximum height is logarithmic. Sophisticated *balanced* binary search tree data structures have been developed that guarantee the height of the tree always to be $O(\log n)$. Therefore, all dictionary operations (insert, delete, query) take $O(\log n)$ time each. Implementations of balanced tree data structures such as red-black trees and splay trees are discussed in Section 12.1 (page 367).

From an algorithm design viewpoint, it is important to know that these trees exist and that they can be used as black boxes to provide an efficient dictionary implementation. When figuring the costs of dictionary operations for algorithm analysis, we can assume the worst-case complexities of balanced binary trees to be a fair measure.

> *Take-Home Lesson:* Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.

Stop and Think: Exploiting Balanced Search Trees

Problem: You are given the task of reading n numbers and then printing them out in sorted order. Suppose you have access to a balanced dictionary data structure, which supports the operations search, insert, delete, minimum, maximum, successor, and predecessor each in $O(\log n)$ time.

1. How can you sort in $O(n \log n)$ time using only insert and in-order traversal?
2. How can you sort in $O(n \log n)$ time using only minimum, successor, and insert?
3. How can you sort in $O(n \log n)$ time using only minimum, insert, delete, search?

Solution: The first problem allows us to do insertion and inorder-traversal. We can build a search tree by inserting all n elements, then do a traversal to access the items in sorted order:

```
Sort1()
    initialize-tree(t)
    While (not EOF)
        read(x);
        insert(x,t)
    Traverse(t)
```

```
Sort2()
    initialize-tree(t)
    While (not EOF)
        read(x);
        insert(x,t);
    y = Minimum(t)
    While (y ≠ NULL) do
        print(y → item)
        y = Successor(y,t)
```

```
Sort3()
    initialize-tree(t)
    While (not EOF)
        read(x);
        insert(x,t);
    y = Minimum(t)
    While (y ≠ NULL) do
        print(y→item)
        Delete(y,t)
        y = Minimum(t)
```

The second problem allows us to use the minimum and successor operations after constructing the tree. We can start from the minimum element, and then repeatedly find the successor to traverse the elements in sorted order.

The third problem does not give us successor, but does allow us delete. We can repeatedly find and delete the minimum element to once again traverse all the elements in sorted order.

Each of these algorithms does a linear number of logarithmic-time operations, and hence runs in $O(n \log n)$ time. The key to exploiting balanced binary search trees is using them as black boxes. ∎

3.5 Priority Queues

Many algorithms process items in a specific order. For example, suppose you must schedule jobs according to their importance relative to other jobs. Scheduling the

jobs requires sorting them by importance, and then evaluating them in this sorted order.

Priority queues are data structures that provide more flexibility than simple sorting, because they allow new elements to enter a system at arbitrary intervals. It is much more cost-effective to insert a new job into a priority queue than to re-sort everything on each such arrival.

The basic priority queue supports three primary operations:

- *Insert(Q,x)*– Given an item x with key k, insert it into the priority queue Q.
- *Find-Minimum(Q)* or *Find-Maximum(Q)*– Return a pointer to the item whose key value is smaller (larger) than any other key in the priority queue Q.
- *Delete-Minimum(Q)* or *Delete-Maximum(Q)*– Remove the item from the priority queue Q whose key is minimum (maximum).

Many naturally occurring processes are accurately modeled by priority queues. Single people maintain a priority queue of potential dating candidates—mentally if not explicitly. One's impression on meeting a new person maps directly to an attractiveness or desirability score. Desirability serves as the *key* field for inserting this new entry into the "little black book" priority queue data structure. Dating is the process of extracting the most desirable person from the data structure (Find-Maximum), spending an evening to evaluate them better, and then reinserting them into the priority queue with a possibly revised score.

Take-Home Lesson: Building algorithms around data structures such as dictionaries and priority queues leads to both clean structure and good performance.

Stop and Think: Basic Priority Queue Implementations

Problem: What is the worst-case time complexity of the three basic priority queue operations (insert, find-minimum, and delete-minimum) when the basic data structure is

- An unsorted array.
- A sorted array.
- A balanced binary search tree.

Solution: There is surprising subtlety in implementing these three operations, even when using a data structure as simple as an unsorted array. The unsorted array

dictionary (discussed on page 73) implemented insertion and deletion in constant time, and search and minimum in linear time. A linear time implementation of delete-minimum can be composed from *find-minimum*, followed by *search*, followed by *delete*.

For sorted arrays, we can implement insert and delete in linear time, and minimum in constant time. However, all priority queue deletions involve only the minimum element. By storing the sorted array in reverse order (largest value on top), the minimum element will be the last one in the array. Deleting the tail element requires no movement of any items, just decrementing the number of remaining items n, and so delete-minimum can be implemented in constant time.

All this is fine, yet the following table claims we can implement find-minimum in constant time for each data structure:

	Unsorted array	Sorted array	Balanced tree
Insert(Q, x)	$O(1)$	$O(n)$	$O(\log n)$
Find-Minimum(Q)	$O(1)$	$O(1)$	$O(1)$
Delete-Minimum(Q)	$O(n)$	$O(1)$	$O(\log n)$

The trick is using an extra variable to store a pointer/index to the minimum entry in each of these structures, so we can simply return this value whenever we are asked to find-minimum. Updating this pointer on each insertion is easy—we update it if and only if the newly inserted value is less than the current minimum. But what happens on a delete-minimum? We can delete the minimum entry *have*, then do an honest find-minimum to restore our canned value. The honest find-minimum takes linear time on an unsorted array and logarithmic time on a tree, and hence can be folded into the cost of each deletion. ∎

Priority queues are very useful data structures. Indeed, they will be the hero of two of our war stories, including the next one. A particularly nice priority queue implementation (the heap) will be discussed in the context of sorting in Section 4.3 (page 108). Further, a complete set of priority queue implementations is presented in Section 12.2 (page 373) of the catalog.

3.6 War Story: Stripping Triangulations

Geometric models used in computer graphics are commonly represented as a triangulated surface, as shown in Figure 3.5(l). High-performance rendering engines have special hardware for rendering and shading triangles. This hardware is so fast that the bottleneck of rendering is the cost of feeding the triangulation structure into the hardware engine.

Although each triangle can be described by specifying its three endpoints, an alternative representation is more efficient. Instead of specifying each triangle in isolation, suppose that we partition the triangles into *strips* of adjacent triangles

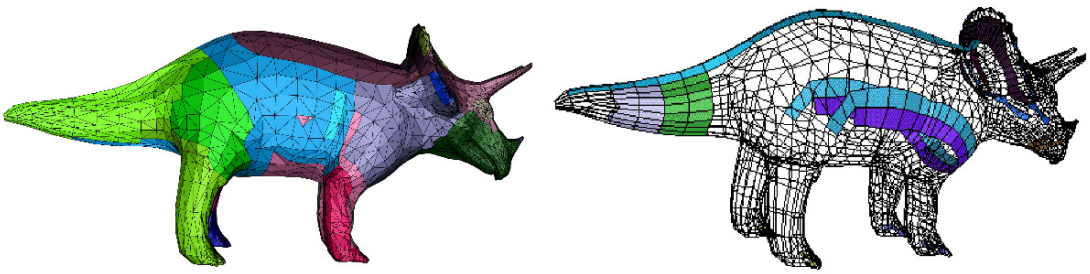

Figure 3.5: (l) A triangulated model of a dinosaur (r) Several triangle strips in the model

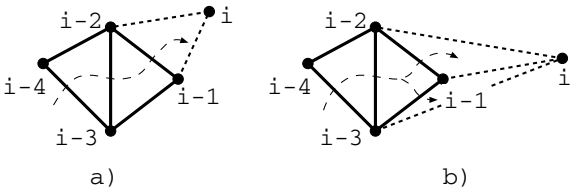

Figure 3.6: Partitioning a triangular mesh into strips: (a) with left-right turns (b) with the flexibility of arbitrary turns

and walk along the strip. Since each triangle shares two vertices in common with its neighbors, we save the cost of retransmitting the two extra vertices and any associated information. To make the description of the triangles unambiguous, the *OpenGL* triangular-mesh renderer assumes that all turns alternate from left to right (as shown in Figure 3.6).

The task of finding a small number of strips that cover each triangle in a mesh can be thought of as a graph problem. The graph of interest has a vertex for every *triangle* of the mesh, and an edge between every pair of vertices representing adjacent triangles. This *dual graph* representation captures all the information about the triangulation (see Section 15.12 (page 520)) needed to partition it into triangular strips.

Once we had the dual graph available, the project could begin in earnest. We sought to partition the vertices into as few paths or strips as possible. Partitioning it into one path implied that we had discovered a Hamiltonian path, which by definition visits each vertex exactly once. Since finding a Hamiltonian path is NP-complete (see Section 16.5 (page 538)), we knew not to look for an optimal algorithm, but concentrate instead on heuristics.

The simplest heuristic for strip cover would start from an arbitrary triangle and then do a left-right walk until the walk ends, either by hitting the boundary of

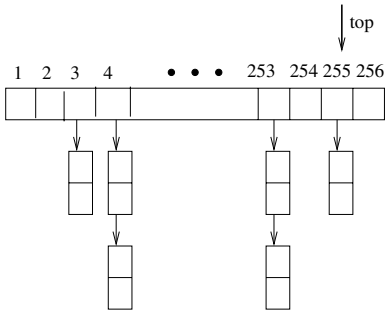

Figure 3.7: A bounded height priority queue for triangle strips

the object or a previously visited triangle. This heuristic had the advantage that it would be fast and simple, although there is no reason why it should find the smallest possible set of left-right strips for a given triangulation.

The *greedy* heuristic would be more likely to result in a small number of strips however. Greedy heuristics always try to grab the best possible thing first. In the case of the triangulation, the natural greedy heuristic would identify the starting triangle that yields the longest left-right strip, and peel that one off first.

Being greedy does not guarantee you the best possible solution either, since the first strip you peel off might break apart a lot of potential strips we might have wanted to use later. Still, being greedy is a good rule of thumb if you want to get rich. Since removing the longest strip would leave the fewest number of triangles for later strips, the greedy heuristic should outperform the naive heuristic.

But how much time does it take to find the largest strip to peel off next? Let k be the length of the walk possible from an average vertex. Using the simplest possible implementation, we could walk from each of the n vertices to find the largest remaining strip to report in $O(kn)$ time. Repeating this for each of the roughly n/k strips we extract yields an $O(n^2)$-time implementation, which would be hopelessly slow on a typical model of 20,000 triangles.

How could we speed this up? It seems wasteful to rewalk from each triangle after deleting a single strip. We could maintain the lengths of all the possible future strips in a data structure. However, whenever we peel off a strip, we must update the lengths of all affected strips. These strips will be shortened because they walked through a triangle that now no longer exists. There are two aspects of such a data structure:

- *Priority Queue* – Since we were repeatedly identifying the longest remaining strip, we needed a priority queue to store the strips ordered according to length. The next strip to peel always sat at the top of the queue. Our priority queue had to permit reducing the priority of arbitrary elements of the queue whenever we updated the strip lengths to reflect what triangles were peeled

Model name	Triangle count	Naive cost	Greedy cost	Greedy time
Diver	3,798	8,460	4,650	6.4 sec
Heads	4,157	10,588	4,749	9.9 sec
Framework	5,602	9,274	7,210	9.7 sec
Bart Simpson	9,654	24,934	11,676	20.5 sec
Enterprise	12,710	29,016	13,738	26.2 sec
Torus	20,000	40,000	20,200	272.7 sec
Jaw	75,842	104,203	95,020	136.2 sec

Figure 3.8: A comparison of the naive versus greedy heuristics for several triangular meshes

away. Because all of the strip lengths were bounded by a fairly small integer (hardware constraints prevent any strip from having more than 256 vertices), we used a bounded-height priority queue (an array of buckets shown in Figure 3.7 and described in Section 12.2 (page 373)). An ordinary heap would also have worked just fine.

To update the queue entry associated with each triangle, we needed to quickly find where it was. This meant that we also needed a ...

- *Dictionary* – For each triangle in the mesh, we needed to find where it was in the queue. This meant storing a pointer to each triangle in a dictionary. By integrating this dictionary with the priority queue, we built a data structure capable of a wide range of operations.

Although there were various other complications, such as quickly recalculating the length of the strips affected by the peeling, the key idea needed to obtain better performance was to use the priority queue. Run time improved by several orders of magnitude after employing this data structure.

How much better did the greedy heuristic do than the naive heuristic? Consider the table in Figure 3.8. In all cases, the greedy heuristic led to a set of strips that cost less, as measured by the total size of the strips. The savings ranged from about 10% to 50%, which is quite remarkable since the greatest possible improvement (going from three vertices per triangle down to one) yields a savings of only 66.6%.

After implementing the greedy heuristic with our priority queue data structure, the program ran in $O(n \cdot k)$ time, where n is the number of triangles and k is the length of the average strip. Thus the torus, which consisted of a small number of very long strips, took longer than the jaw, even though the latter contained over three times as many triangles.

There are several lessons to be gleaned from this story. First, when working with a large enough data set, only linear or near linear algorithms (say $O(n \log n)$) are likely to be fast enough. Second, choosing the right data structure is often the key to getting the time complexity down to this point. Finally, using smart heuristic

like greedy is likely to significantly improve quality over the naive approach. How much the improvement will be can only be determined by experimentation.

3.7 Hashing and Strings

Hash tables are a *very* practical way to maintain a dictionary. They exploit the fact that looking an item up in an array takes constant time once you have its index. A hash function is a mathematical function that maps keys to integers. We will use the value of our hash function as an index into an array, and store our item at that position.

The first step of the hash function is usually to map each key to a big integer. Let α be the size of the alphabet on which a given string S is written. Let `char(c)` be a function that maps each symbol of the alphabet to a unique integer from 0 to $\alpha - 1$. The function

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

maps each string to a unique (but large) integer by treating the characters of the string as "digits" in a base-α number system.

The result is unique identifier numbers, but they are so large they will quickly exceed the number of slots in our hash table (denoted by m). We must reduce this number to an integer between 0 and $m-1$, by taking the remainder of $H(S)$ mod m. This works on the same principle as a roulette wheel. The ball travels a long distance around and around the circumference-m wheel $\lfloor H(S)/m \rfloor$ times before settling down to a random bin. If the table size is selected with enough finesse (ideally m is a large prime not too close to $2^i - 1$), the resulting hash values should be fairly uniformly distributed.

3.7.1 Collision Resolution

No matter how good our hash function is, we had better be prepared for collisions, because two distinct keys will occasionally hash to the same value. *Chaining* is the easiest approach to collision resolution. Represent the hash table as an array of m linked lists, as shown in Figure 3.9. The ith list will contain all the items that hash to the value of i. Thus search, insertion, and deletion reduce to the corresponding problem in linked lists. If the n keys are distributed uniformly in a table, each list will contain roughly n/m elements, making them a constant size when $m \approx n$.

Chaining is very natural, but devotes a considerable amount of memory to pointers. This is space that could be used to make the table larger, and hence the "lists" smaller.

The alternative is something called *open addressing*. The hash table is maintained as an array of elements (not buckets), each initialized to null, as shown in Figure 3.10. On an insertion, we check to see if the desired position is empty. If so,

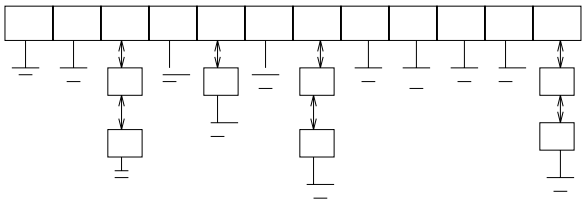

Figure 3.9: Collision resolution by chaining

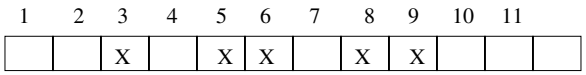

Figure 3.10: Collision resolution by open addressing

we insert it. If not, we must find some other place to insert it instead. The simplest possibility (called *sequential probing*) inserts the item in the next open spot in the table. If the table is not too full, the contiguous runs of items should be fairly small, hence this location *should* be only a few slots from its intended position.

Searching for a given key now involves going to the appropriate hash value and checking to see if the item there is the one we want. If so, return it. Otherwise we must keep checking through the length of the run.

Deletion in an open addressing scheme can get ugly, since removing one element might break a chain of insertions, making some elements inaccessible. We have no alternative but to reinsert all the items in the run following the new hole.

Chaining and open addressing both require $O(m)$ to initialize an m-element hash table to null elements prior to the first insertion. Traversing all the elements in the table takes $O(n + m)$ time for chaining, because we have to scan all m buckets looking for elements, even if the actual number of inserted items is small. This reduces to $O(m)$ time for open addressing, since n must be at most m.

When using chaining with doubly-linked lists to resolve collisions in an m-element hash table, the dictionary operations for n items can be implemented in the following expected and worst case times:

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

Pragmatically, a hash table is often the best data structure to maintain a dictionary. The applications of hashing go far beyond dictionaries, however, as we will see below.

3.7.2 Efficient String Matching via Hashing

Strings are sequences of characters where the order of the characters matters, since *ALGORITHM* is different than *LOGARITHM*. Text strings are fundamental to a host of computing applications, from programming language parsing/compilation, to web search engines, to biological sequence analysis.

The primary data structure for representing strings is an array of characters. This allows us constant-time access to the ith character of the string. Some auxiliary information must be maintained to mark the end of the string—either a special end-of-string character or (perhaps more usefully) a count of the n characters in the string.

The most fundamental operation on text strings is substring search, namely:

Problem: Substring Pattern Matching
Input: A text string t and a pattern string p.
Output: Does t contain the pattern p as a substring, and if so where?

The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character. As demonstrated in Section 2.5.3 (page 43), this runs in $O(nm)$ time, where $n = |t|$ and $m = |p|$.

This quadratic bound is worst-case. More complicated, worst-case linear-time search algorithms do exist: see Section 18.3 (page 628) for a complete discussion. But here we give a linear *expected-time* algorithm for string matching, called the Rabin-Karp algorithm. It is based on hashing. Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t. If these two strings are identical, clearly the resulting hash values must be the same. If the two strings are different, the hash values will *almost certainly* be different. These false positives should be so rare that we can easily spend the $O(m)$ time it takes to explicitly check the identity of two strings whenever the hash values agree.

This reduces string matching to $n-m+2$ hash value computations (the $n-m+1$ windows of t, plus one hash of p), plus what *should be* a very small number of $O(m)$ time verification steps. The catch is that it takes $O(m)$ time to compute a hash function on an m-character string, and $O(n)$ such computations seems to leave us with an $O(mn)$ algorithm again.

But let's look more closely at our previously defined hash function, applied to the m characters starting from the jth position of string S:

$$H(S, j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j})$$

What changes if we now try to compute $H(S, j+1)$—the hash of the next window of m characters? Note that $m-1$ characters are the same in both windows, although this differs by one in the number of times they are multiplied by α. A little algebra reveals that

$$H(S, j+1) = \alpha(H(S, j) - \alpha^{m-1} char(s_j)) + char(s_{j+m})$$

This means that once we know the hash value from the j position, we can find the hash value from the $(j+1)$st position for the cost of two multiplications, one addition, and one subtraction. This can be done in constant time (the value of α^{m-1} can be computed once and used for all hash value computations). This math works even if we compute $H(S, j)$ mod M, where M is a reasonably large prime number, thus keeping the size of our hash values small (at most M) even when the pattern string is long.

Rabin-Karp is a good example of a randomized algorithm (if we pick M in some random way). We get no guarantee the algorithm runs in $O(n+m)$ time, because we may get unlucky and have the hash values regularly collide with spurious matches. Still, the odds are heavily in our favor—if the hash function returns values uniformly from 0 to $M-1$, the probability of a false collision should be $1/M$. This is quite reasonable: if $M \approx n$, there should only be one false collision per string, and if $M \approx n^k$ for $k \geq 2$, the odds are great we will never see any false collisions.

3.7.3 Duplicate Detection Via Hashing

The key idea of hashing is to represent a large object (be it a key, a string, or a substring) using a single number. The goal is a representation of the large object by an entity that can be manipulated in constant time, such that it is relatively unlikely that two different large objects map to the same value.

Hashing has a variety of clever applications beyond just speeding up search. I once heard Udi Manber—then Chief Scientist at Yahoo—talk about the algorithms employed at his company. The three most important algorithms at Yahoo, he said, were hashing, hashing, and hashing.

Consider the following problems with nice hashing solutions:

- *Is a given document different from all the rest in a large corpus?* – A search engine with a huge database of n documents spiders yet another webpage. How can it tell whether this adds something new to add to the database, or is just a duplicate page that exists elsewhere on the Web?

  Explicitly comparing the new document D to all n documents is hopelessly inefficient for a large corpus. But we can hash D to an integer, and compare it to the hash codes of the rest of the corpus. Only when there is a collision is D a possible duplicate. Since we expect few spurious collisions, we can explicitly compare the few documents sharing the exact hash code with little effort.

- *Is part of this document plagiarized from a document in a large corpus?* – A lazy student copies a portion of a Web document into their term paper. "The Web is a big place," he smirks. "How will anyone ever find which one?"

  This is a more difficult problem than the previous application. Adding, deleting, or changing even one character from a document will completely change its hash code. Thus the hash codes produced in the previous application cannot help for this more general problem.

  However, we *could* build a hash table of all overlapping windows (substrings) of length w in all the documents in the corpus. Whenever there is a match of hash codes, there is likely a common substring of length w between the two documents, which can then be further investigated. We should choose w to be long enough so such a co-occurrence is very unlikely to happen by chance.

  The biggest downside of this scheme is that the size of the hash table becomes as large as the documents themselves. Retaining a small but well-chosen subset of these hash codes (say those which are exact multiples of 100) for each document leaves us likely to detect sufficiently long duplicate strings.

- *How can I convince you that a file isn't changed?* – In a closed-bid auction, each party submits their bid in secret before the announced deadline. If you knew what the other parties were bidding, you could arrange to bid \$1 more than the highest opponent and walk off with the prize as cheaply as possible. Thus the "right" auction strategy is to hack into the computer containing the bids just prior to the deadline, read the bids, and then magically emerge the winner.

  How can this be prevented? What if everyone submits a hash code of their actual bid prior to the deadline, and then submits the full bid after the deadline? The auctioneer will pick the largest full bid, but checks to make sure the hash code matches that submitted prior to the deadline. Such *cryptographic hashing* methods provide a way to ensure that the file you give me today is the same as original, because any changes to the file will result in changing the hash code.

Although the worst-case bounds on anything involving hashing are dismal, with a proper hash function we can confidently expect good behavior. Hashing is a fundamental idea in randomized algorithms, yielding linear expected-time algorithms for problems otherwise $\Theta(n \log n)$, or $\Theta(n^2)$ in the worst case.

3.8 Specialized Data Structures

The basic data structures described thus far all represent an unstructured set of items so as to facilitate retrieval operations. These data structures are well known to most programmers. Not as well known are data structures for representing more

structured or specialized kinds of objects, such as points in space, strings, and graphs.

The design principles of these data structures are the same as for basic objects. There exists a set of basic operations we need to perform repeatedly. We seek a data structure that supports these operations very efficiently. These efficient, specialized data structures are important for efficient graph and geometric algorithms so one should be aware of their existence. Details appear throughout the catalog.

- *String data structures* – Character strings are typically represented by arrays of characters, perhaps with a special character to mark the end of the string. Suffix trees/arrays are special data structures that preprocess strings to make pattern matching operations faster. See Section 12.3 (page 377) for details.

- *Geometric data structures* – Geometric data typically consists of collections of data points and regions. Regions in the plane can be described by polygons, where the boundary of the polygon is given by a chain of line segments. Polygons can be represented using an array of points $(v_1, \ldots, v_n, v_1)$, such that (v_i, v_{i+1}) is a segment of the boundary. Spatial data structures such as *kd*-trees organize points and regions by geometric location to support fast search. For more details, see Section 12.6 (page 389).

- *Graph data structures* – Graphs are typically represented using either adjacency matrices or adjacency lists. The choice of representation can have a substantial impact on the design of the resulting graph algorithms, as discussed in Chapter 6 and in the catalog in Section 12.4.

- *Set data structures* – Subsets of items are typically represented using a dictionary to support fast membership queries. Alternately, *bit vectors* are boolean arrays such that the ith bit represents true if i is in the subset. Data structures for manipulating sets is presented in the catalog in Section 12.5. The union-find data structure for maintaining set partitions will be covered in Section 6.1.3 (page 198).

3.9 War Story: String 'em Up

The human genome encodes all the information necessary to build a person. This project has already had an enormous impact on medicine and molecular biology. Algorists have become interested in the human genome project as well, for several reasons:

- DNA sequences can be accurately represented as strings of characters on the four-letter alphabet (A,C,T,G). Biologist's needs have sparked new interest in old algorithmic problems such as string matching (see Section 18.3 (page 628)) as well as creating new problems such as shortest common superstring (see Section 18.9 (page 654)).

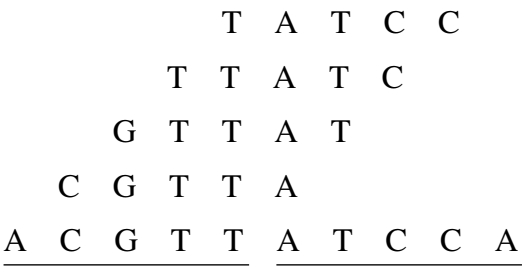

Figure 3.11: The concatenation of two fragments can be in S only if all sub-fragments are

- DNA sequences are very *long* strings. The human genome is approximately three billion base pairs (or characters) long. Such large problem size means that asymptotic (Big-Oh) complexity analysis is usually fully justified on biological problems.

- Enough money is being invested in genomics for computer scientists to want to claim their piece of the action.

One of my interests in computational biology revolved around a proposed technique for DNA sequencing called sequencing by hybridization (SBH). This procedure attaches a set of probes to an array, forming a *sequencing chip*. Each of these probes determines whether or not the probe string occurs as a substring of the DNA target. The target DNA can now be sequenced based on the constraints of which strings are (and are not) substrings of the target.

We sought to identify all the strings of length $2k$ that are possible substrings of an unknown string S, given the set of all length k substrings of S. For example, suppose we know that AC, CA, and CC are the only length-2 substrings of S. It is possible that $ACCA$ is a substring of S, since the center substring is one of our possibilities. However, $CAAC$ *cannot* be a substring of S, since AA is not a substring of S. We needed to find a fast algorithm to construct all the consistent length-$2k$ strings, since S could be very long.

The simplest algorithm to build the $2k$ strings would be to concatenate all $O(n^2)$ pairs of k-strings together, and then test to make sure that all $(k-1)$ length-k substrings spanning the boundary of the concatenation were in fact substrings, as shown in Figure 3.11. For example, the nine possible concatenations of AC, CA, and CC are $ACAC$, $ACCA$, $ACCC$, $CAAC$, $CACA$, $CACC$, $CCAC$, $CCCA$, and $CCCC$. Only $CAAC$ can be eliminated because of the absence of AA.

We needed a fast way of testing whether the $k-1$ substrings straddling the concatenation were members of our dictionary of permissible k-strings. The time it takes to do this depends upon which dictionary data structure we use. A binary search tree could find the correct string within $O(\log n)$ comparisons, where each

comparison involved testing which of two length-k strings appeared first in alphabetical order. The total time using such a binary search tree would be $O(k \log n)$.

That seemed pretty good. So my graduate student, Dimitris Margaritis, used a binary search tree data structure for our implementation. It worked great up until the moment we ran it.

"I've tried the fastest computer in our department, but our program is too slow," Dimitris complained. "It takes forever on string lengths of only 2,000 characters. We will never get up to 50,000."

We profiled our program and discovered that almost all the time was spent searching in this data structure. This was no surprise since we did this $k-1$ times for each of the $O(n^2)$ possible concatenations. We needed a faster dictionary data structure, since search was the innermost operation in such a deep loop.

"How about using a hash table?" I suggested. "It should take $O(k)$ time to hash a k-character string and look it up in our table. That should knock off a factor of $O(\log n)$, which will mean something when $n \approx 2{,}000$."

Dimitris went back and implemented a hash table implementation for our dictionary. Again, it worked great up until the moment we ran it.

"Our program is still too slow," Dimitris complained. "Sure, it is now about ten times faster on strings of length 2,000. So now we can get up to about 4,000 characters. Big deal. We will never get up to 50,000."

"We should have expected this," I mused. "After all, $\lg_2(2,000) \approx 11$. We need a faster data structure to search in our dictionary of strings."

"But what can be faster than a hash table?" Dimitris countered. "To look up a k-character string, you must read all k characters. Our hash table already does $O(k)$ searching."

"Sure, it takes k comparisons to test the first substring. But maybe we can do better on the second test. Remember where our dictionary queries are coming from. When we concatenate $ABCD$ with $EFGH$, we are first testing whether $BCDE$ is in the dictionary, then $CDEF$. These strings differ from each other by only one character. We should be able to exploit this so each subsequent test takes constant time to perform. . . ."

"We can't do that with a hash table," Dimitris observed. "The second key is not going to be anywhere near the first in the table. A binary search tree won't help, either. Since the keys $ABCD$ and $BCDE$ differ according to the first character, the two strings will be in different parts of the tree."

"But we can use a suffix tree to do this," I countered. "A suffix tree is a trie containing all the suffixes of a given set of strings. For example, the suffixes of $ACAC$ are $\{ACAC, CAC, AC, C\}$. Coupled with suffixes of string $CACT$, we get the suffix tree of Figure 3.12. By following a pointer from $ACAC$ to its longest proper suffix CAC, we get to the right place to test whether $CACT$ is in our set of strings. One character comparison is all we need to do from there."

Suffix trees are amazing data structures, discussed in considerably more detail in Section 12.3 (page 377). Dimitris did some reading about them, then built a nice

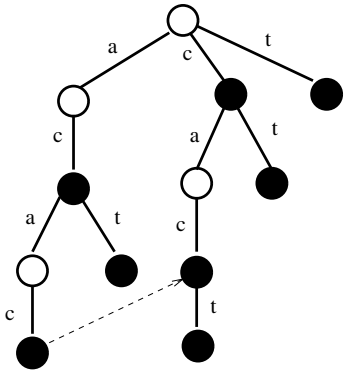

Figure 3.12: Suffix tree on $ACAC$ and $CACT$, with the pointer to the suffix of $ACAC$

suffix tree implementation for our dictionary. Once again, it worked great up until the moment we ran it.

"Now our program is faster, but it runs out of memory," Dimitris complained. "The suffix tree builds a path of length k for each suffix of length k, so all told there can be $\Theta(n^2)$ nodes in the tree. It crashes when we go beyond 2,000 characters. We will never get up to strings with 50,000 characters."

I wasn't ready to give up yet. "There is a way around the space problem, by using compressed suffix trees," I recalled. "Instead of explicitly representing long paths of character nodes, we can refer back to the original string." Compressed suffix trees always take linear space, as described in Section 12.3 (page 377).

Dimitris went back one last time and implemented the compressed suffix tree data structure. *Now* it worked great! As shown in Figure 3.13, we ran our simulation for strings of length $n = 65{,}536$ without incident. Our results showed that interactive SBH could be a very efficient sequencing technique. Based on these simulations, we were able to arouse interest in our technique from biologists. Making the actual wet laboratory experiments feasible provided another computational challenge, which is reported in Section 7.7 (page 263).

The take-home lessons for programmers from Figure 3.13 should be apparent. We isolated a single operation (dictionary string search) that was being performed repeatedly and optimized the data structure we used to support it. We started with a simple implementation (binary search trees) in the hopes that it would suffice, and then used profiling to reveal the trouble when it didn't. When an improved dictionary structure still did not suffice, we looked deeper into the kind of queries we were performing, so that we could identify an even better data structure. Finally, we didn't give up until we had achieved the level of performance we needed. In algorithms, as in life, persistence usually pays off.

String length	Binary tree	Hash table	Suffix tree	Compressed tree
8	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0
32	0.1	0.0	0.0	0.0
64	0.3	0.4	0.3	0.0
128	2.4	1.1	0.5	0.0
256	17.1	9.4	3.8	0.2
512	31.6	67.0	6.9	1.3
1,024	1,828.9	96.6	31.5	2.7
2,048	11,441.7	941.7	553.6	39.0
4,096	$>$ 2 days	5,246.7	out of	45.4
8,192		$>$ 2 days	memory	642.0
16,384				1,614.0
32,768				13,657.8
65,536				39,776.9

Figure 3.13: Run times (in seconds) for the SBH simulation using various data structures

Chapter Notes

Optimizing hash table performance is surprisingly complicated for such a conceptually simple data structure. The importance of short runs in open addressing has to more sophisticated schemes than sequential probing for optimal hash table performance. For more details, see Knuth [Knu98].

Our triangle strip optimizing program, *stripe*, is described in [ESV96]. Hashing techniques for plagiarism detection are discussed in [SWA03].

Surveys of algorithmic issues in DNA sequencing by hybridization include [CK94, PL94]. Our work on interactive SBH reported in the war story is reported in [MS95a].

3.10 Exercises

Stacks, Queues, and Lists

3-1. *[3]* A common problem for compilers and text editors is determining whether the parentheses in a string are balanced and properly nested. For example, the string (((())())() contains properly nested pairs of parentheses, which the strings)()(and ()) do not. Give an algorithm that returns true if a string contains properly nested and balanced parentheses, and false if otherwise. For full credit, identify the position of the first offending parenthesis if the string is not properly nested and balanced.

3-2. *[3]* Write a program to reverse the direction of a given singly-linked list. In other words, after the reversal all pointers should now point backwards. Your algorithm should take linear time.

3-3. *[5]* We have seen how dynamic arrays enable arrays to grow while still achieving constant-time amortized performance. This problem concerns extending dynamic arrays to let them both grow and shrink on demand.

(a) Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example sequence of insertions and deletions where this strategy gives a bad amortized cost.

(b) Then, give a better underflow strategy than that suggested above, one that achieves constant amortized cost per deletion.

Trees and Other Dictionary Structures

3-4. *[3]* Design a dictionary data structure in which search, insertion, and deletion can all be processed in $O(1)$ time in the worst case. You may assume the set elements are integers drawn from a finite set $1, 2, .., n$, and initialization can take $O(n)$ time.

3-5. *[3]* Find the overhead fraction (the ratio of data space over total space) for each of the following binary tree implementations on n nodes:

(a) All nodes store data, two child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.

(b) Only leaf nodes store data; internal nodes store two child pointers. The data field requires four bytes and each pointer requires two bytes.

3-6. *[5]* Describe how to modify any balanced tree data structure such that search, insert, delete, minimum, and maximum still take $O(\log n)$ time each, but successor and predecessor now take $O(1)$ time each. Which operations have to be modified to support this?

3-7. *[5]* Suppose you have access to a balanced dictionary data structure, which supports each of the operations search, insert, delete, minimum, maximum, successor, and predecessor in $O(\log n)$ time. Explain how to modify the insert and delete operations so they still take $O(\log n)$ but now minimum and maximum take $O(1)$ time. (Hint: think in terms of using the abstract dictionary operations, instead of mucking about with pointers and the like.)

3-8. *[6]* Design a data structure to support the following operations:

- *insert(x,T)* – Insert item x into the set T.
- *delete(k,T)* – Delete the kth smallest element from T.
- *member(x,T)* – Return true iff $x \in T$.

All operations must take $O(\log n)$ time on an n-element set.

3-9. *[8]* A *concatenate* operation takes two sets S_1 and S_2, where every key in S_1 is smaller than any key in S_2, and merges them together. Give an algorithm to concatenate two binary search trees into one binary search tree. The worst-case running time should be $O(h)$, where h is the maximal height of the two trees.

Applications of Tree Structures

3-10. *[5]* In the *bin-packing problem*, we are given n metal objects, each weighing between zero and one kilogram. Our goal is to find the smallest number of bins that will hold the n objects, with each bin holding one kilogram at most.

- The *best-fit heuristic* for bin packing is as follows. Consider the objects in the order in which they are given. For each object, place it into the partially filled bin with the smallest amount of extra room *after* the object is inserted.. If no such bin exists, start a new bin. Design an algorithm that implements the best-fit heuristic (taking as input the n weights $w_1, w_2, ..., w_n$ and outputting the number of bins used) in $O(n \log n)$ time.
- Repeat the above using the *worst-fit heuristic*, where we put the next object in the partially filled bin with the largest amount of extra room *after* the object is inserted.

3-11. *[5]* Suppose that we are given a sequence of n values $x_1, x_2, ..., x_n$ and seek to quickly answer repeated queries of the form: given i and j, find the smallest value in $x_i, \ldots, x_j$.

(a) Design a data structure that uses $O(n^2)$ space and answers queries in $O(1)$ time.

(b) Design a data structure that uses $O(n)$ space and answers queries in $O(\log n)$ time. For partial credit, your data structure can use $O(n \log n)$ space and have $O(\log n)$ query time.

3-12. *[5]* Suppose you are given an input set S of n numbers, and a black box that if given any sequence of real numbers and an integer k instantly and correctly answers whether there is a subset of input sequence whose sum is exactly k. Show how to use the black box $O(n)$ times to find a subset of S that adds up to k.

3-13. *[5]* Let $A[1..n]$ be an array of real numbers. Design an algorithm to perform any sequence of the following operations:

- *Add(i,y)* – Add the value y to the ith number.
- *Partial-sum(i)* – Return the sum of the first i numbers, i.e. $\sum_{j=1}^{i} A[j]$.

There are no insertions or deletions; the only change is to the values of the numbers. Each operation should take $O(\log n)$ steps. You may use one additional array of size n as a work space.

3-14. *[8]* Extend the data structure of the previous problem to support insertions and deletions. Each element now has both a *key* and a *value*. An element is accessed by its key. The addition operation is applied to the values, but the elements are accessed by its key. The *Partial_sum* operation is different.

- *Add(k,y)* – Add the value y to the item with key k.
- *Insert(k,y)* – Insert a new item with key k and value y.
- *Delete(k)* – Delete the item with key k.

- *Partial-sum(k)* – Return the sum of all the elements currently in the set whose key is less than y, i.e. $\sum_{x_j < y} x_i$.

The worst case running time should still be $O(n \log n)$ for any sequence of $O(n)$ operations.

3-15. *[8]* Design a data structure that allows one to search, insert, and delete an integer X in $O(1)$ time (i.e. , constant time, independent of the total number of integers stored). Assume that $1 \leq X \leq n$ and that there are $m + n$ units of space available, where m is the maximum number of integers that can be in the table at any one time. (Hint: use two arrays $A[1..n]$ and $B[1..m]$.) You are not allowed to initialize either A or B, as that would take $O(m)$ or $O(n)$ operations. This means the arrays are full of random garbage to begin with, so you must be very careful.

Implementation Projects

3-16. *[5]* Implement versions of several different dictionary data structures, such as linked lists, binary trees, balanced binary search trees, and hash tables. Conduct experiments to assess the relative performance of these data structures in a simple application that reads a large text file and reports exactly one instance of each word that appears within it. This application can be efficiently implemented by maintaining a dictionary of all distinct words that have appeared thus far in the text and inserting/reporting each word that is not found. Write a brief report with your conclusions.

3-17. *[5]* A Caesar shift (see Section 18.6 (page 641)) is a very simple class of ciphers for secret messages. Unfortunately, they can be broken using statistical properties of English. Develop a program capable of decrypting Caesar shifts of sufficiently long texts.

Interview Problems

3-18. *[3]* What method would you use to look up a word in a dictionary?

3-19. *[3]* Imagine you have a closet full of shirts. What can you do to organize your shirts for easy retrieval?

3-20. *[4]* Write a function to find the middle node of a singly-linked list.

3-21. *[4]* Write a function to compare whether two binary trees are identical. Identical trees have the same key value at each position and the same structure.

3-22. *[4]* Write a program to convert a binary search tree into a linked list.

3-23. *[4]* Implement an algorithm to reverse a linked list. Now do it without recursion.

3-24. *[5]* What is the best data structure for maintaining URLs that have been visited by a Web crawler? Give an algorithm to test whether a given URL has already been visited, optimizing both space and time.

3-25. *[4]* You are given a search string and a magazine. You seek to generate all the characters in search string by cutting them out from the magazine. Give an algorithm to efficiently determine whether the magazine contains all the letters in the search string.

3-26. *[4]* Reverse the words in a sentence—i.e., "My name is Chris" becomes "Chris is name My." Optimize for time and space.

3-27. *[5]* Determine whether a linked list contains a loop as quickly as possible without using any extra storage. Also, identify the location of the loop.

3-28. *[5]* You have an unordered array X of n integers. Find the array M containing n elements where M_i is the product of all integers in X except for X_i. You may not use division. You can use extra memory. (Hint: There are solutions faster than $O(n^2)$.)

3-29. *[6]* Give an algorithm for finding an ordered word pair (e.g., "New York") occurring with the greatest frequency in a given webpage. Which data structures would you use? Optimize both time and space.

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

3-1. "Jolly Jumpers" – Programming Challenges 110201, UVA Judge 10038.

3-2. "Crypt Kicker" – Programming Challenges 110204, UVA Judge 843.

3-3. "Where's Waldorf?" – Programming Challenges 110302, UVA Judge 10010.

3-4. "Crypt Kicker II" – Programming Challenges 110304, UVA Judge 850.

4

Sorting and Searching

Typical computer science students study the basic sorting algorithms at least three times before they graduate: first in introductory programming, then in data structures, and finally in their algorithms course. Why is sorting worth so much attention? There are several reasons:

- Sorting is the basic building block that many other algorithms are built around. By understanding sorting, we obtain an amazing amount of power to solve other problems.
- Most of the interesting ideas used in the design of algorithms appear in the context of sorting, such as divide-and-conquer, data structures, and randomized algorithms.
- Computers have historically spent more time sorting than doing anything else. A quarter of all mainframe cycles were spent sorting data [Knu98]. Sorting remains the most ubiquitous combinatorial algorithm problem in practice.
- Sorting is the most thoroughly studied problem in computer science. Literally dozens of different algorithms are known, most of which possess some particular advantage over all other algorithms in certain situations.

In this chapter, we will discuss sorting, stressing how sorting can be applied to solving other problems. In this sense, sorting behaves more like a data structure than a problem in its own right. We then give detailed presentations of several fundamental algorithms: heapsort, mergesort, quicksort, and distribution sort as examples of important algorithm design paradigms. Sorting is also represented by Section 14.1 (page 436) in the problem catalog.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_4,

4.1 Applications of Sorting

We will review several sorting algorithms and their complexities over the course of this chapter. But the punch-line is this: clever sorting algorithms exist that run in $O(n \log n)$. This is a *big* improvement over naive $O(n^2)$ sorting algorithms for large values of n. Consider the following table:

n	$n^2/4$	$n \lg n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*

You might still get away with using a quadratic-time algorithm even if $n = 10,000$, but quadratic-time sorting is clearly ridiculous once $n \geq 100,000$.

Many important problems can be reduced to sorting, so we can use our clever $O(n \log n)$ algorithms to do work that might otherwise seem to require a quadratic algorithm. An important algorithm design technique is to use sorting as a basic building block, because many other problems become easy once a set of items is sorted.

Consider the following applications:

- *Searching* – Binary search tests whether an item is in a dictionary in $O(\log n)$ time, provided the keys are all sorted. Search preprocessing is perhaps the single most important application of sorting.

- *Closest pair* – Given a set of n numbers, how do you find the pair of numbers that have the smallest difference between them? Once the numbers are sorted, the closest pair of numbers must lie next to each other somewhere in sorted order. Thus, a linear-time scan through them completes the job, for a total of $O(n \log n)$ time including the sorting.

- *Element uniqueness* – Are there any duplicates in a given set of n items? This is a special case of the closest-pair problem above, where we ask if there is a pair separated by a gap of zero. The most efficient algorithm sorts the numbers and then does a linear scan though checking all adjacent pairs.

- *Frequency distribution* – Given a set of n items, which element occurs the largest number of times in the set? If the items are sorted, we can sweep from left to right and count them, since all identical items will be lumped together during sorting.

  To find out how often an arbitrary element k occurs, look up k using binary search in a sorted array of keys. By walking to the left of this point until the first the element is not k and then doing the same to the right, we can find

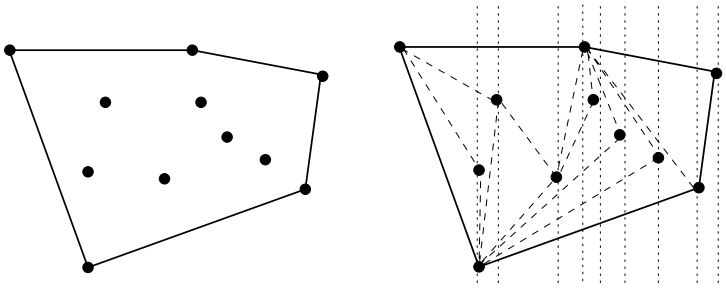

Figure 4.1: The convex hull of a set of points (l), constructed by left-to-right insertion.

this count in $O(\log n + c)$ time, where c is the number of occurrences of k. Even better, the number of instances of k can be found in $O(\log n)$ time by using binary search to look for the positions of both $k - \epsilon$ and $k + \epsilon$ (where ϵ is arbitrarily small) and then taking the difference of these positions.

- *Selection* – What is the kth largest item in an array? If the keys are placed in sorted order, the kth largest can be found in constant time by simply looking at the kth position of the array. In particular, the median element (see Section 14.3 (page 445)) appears in the $(n/2)$nd position in sorted order.

- *Convex hulls* – What is the polygon of smallest area that contains a given set of n points in two dimensions? The convex hull is like a rubber band stretched over the points in the plane and then released. It compresses to just cover the points, as shown in Figure 4.1(l). The convex hull gives a nice representation of the shape of the points and is an important building block for more sophisticated geometric algorithms, as discussed in the catalog in Section 17.2 (page 568).

  But how can we use sorting to construct the convex hull? Once you have the points sorted by x-coordinate, the points can be inserted from left to right into the hull. Since the right-most point is always on the boundary, we know that it will appear in the hull. Adding this new right-most point may cause others to be deleted, but we can quickly identify these points because they lie inside the polygon formed by adding the new point. See the example in Figure 4.1(r). These points will be neighbors of the previous point we inserted, so they will be easy to find and delete. The total time is linear after the sorting has been done.

While a few of these problems (namely median and selection) can be solved in linear time using more sophisticated algorithms, sorting provides quick and easy solutions to all of these problems. It is a rare application where the running time

of sorting proves to be the bottleneck, especially a bottleneck that could have otherwise been removed using more clever algorithmics. Never be afraid to spend time sorting, provided you use an efficient sorting routine.

Take-Home Lesson: Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.

Stop and Think: Finding the Intersection

Problem: Give an efficient algorithm to determine whether two sets (of size m and n, respectively) are disjoint. Analyze the worst-case complexity in terms of m and n, considering the case where m is substantially smaller than n.

Solution: At least three algorithms come to mind, all of which are variants of sorting and searching:

- *First sort the big set* – The big set can be sorted in $O(n \log n)$ time. We can now do a binary search with each of the m elements in the second, looking to see if it exists in the big set. The total time will be $O((n+m) \log n)$.

- *First sort the small set* – The small set can be sorted in $O(m \log m)$ time. We can now do a binary search with each of the n elements in the big set, looking to see if it exists in the small one. The total time will be $O((n+m) \log m)$.

- *Sort both sets* – Observe that once the two sets are sorted, we no longer have to do binary search to detect a common element. We can compare the smallest elements of the two sorted sets, and discard the smaller one if they are not identical. By repeating this idea recursively on the now smaller sets, we can test for duplication in linear time after sorting. The total cost is $O(n \log n + m \log m + n + m)$.

So, which of these is the fastest method? Clearly small-set sorting trumps big-set sorting, since $\log m < \log n$ when $m < n$. Similarly, $(n+m) \log m$ must be asymptotically less than $n \log n$, since $n + m < 2n$ when $m < n$. Thus, sorting the small set is the best of these options. Note that this is linear when m is constant in size.

Note that *expected* linear time can be achieved by hashing. Build a hash table containing the elements of both sets, and verify that collisions in the same bucket are in fact identical elements. In practice, this may be the best solution. ∎

4.2 Pragmatics of Sorting

We have seen many algorithmic applications of sorting, and we will see several efficient sorting algorithms. One issue stands between them: in what order do we want our items sorted?

The answers to this basic question are application-specific. Consider the following considerations:

- *Increasing or decreasing order?* – A set of keys S are sorted in *ascending* order when $S_i \leq S_{i+1}$ for all $1 \leq i < n$. They are in *descending order* when $S_i \geq S_{i+1}$ for all $1 \leq i < n$. Different applications call for different orders.

- *Sorting just the key or an entire record?* – Sorting a data set involves maintaining the integrity of complex data records. A mailing list of names, addresses, and phone numbers may be sorted by names as the key field, but it had better retain the linkage between names and addresses. Thus, we need to specify which field is the key field in any complex record, and understand the full extent of each record.

- *What should we do with equal keys?* Elements with equal key values will all bunch together in any total order, but sometimes the relative order among these keys matters. Suppose an encyclopedia contains both Michael Jordan (the basketball player) and Michael Jordan (the statistician). Which entry should appear first? You may need to resort to secondary keys, such as article size, to resolve ties in a meaningful way.

  Sometimes it is required to leave the items in the same relative order as in the original permutation. Sorting algorithms that automatically enforce this requirement are called *stable*. Unfortunately few fast algorithms are stable. Stability can be achieved for any sorting algorithm by adding the initial position as a secondary key.

  Of course we could make no decision about equal key order and let the ties fall where they may. But beware, certain efficient sort algorithms (such as quicksort) can run into quadratic performance trouble unless explicitly engineered to deal with large numbers of ties.

- *What about non-numerical data?* – Alphabetizing is the sorting of text strings. Libraries have very complete and complicated rules concerning the relative *collating sequence* of characters and punctuation. Is *Skiena* the same key as *skiena*? Is *Brown-Williams* before or after *Brown America*, and before or after *Brown, John*?

The right way to specify such matters to your sorting algorithm is with an application-specific pairwise-element *comparison function*. Such a comparison function takes pointers to record items a and b and returns "$<$" if $a < b$, "$>$" if $a > b$, or "$=$" if $a = b$.

By abstracting the pairwise ordering decision to such a comparison function, we can implement sorting algorithms independently of such criteria. We simply pass the comparison function in as an argument to the sort procedure. Any reasonable programming language has a built-in sort routine as a library function. You are almost always better off using this than writing your own routine. For example, the standard library for C contains the `qsort` function for sorting:

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
          int (*compare) (const void *, const void *));
```

The key to using `qsort` is realizing what its arguments do. It sorts the first `nel` elements of an array (pointed to by `base`), where each element is `width`-bytes long. Thus we can sort arrays of 1-byte characters, 4-byte integers, or 100-byte records, all by changing the value of `width`.

The ultimate desired order is determined by the `compare` function. It takes as arguments pointers to two `width`-byte elements, and returns a negative number if the first belongs before the second in sorted order, a positive number if the second belongs before the first, or zero if they are the same. Here is a comparison function to sort integers in increasing order:

```
int intcompare(int *i, int *j)
{
    if (*i > *j) return (1);
    if (*i < *j) return (-1);
    return (0);
}
```

This comparison function can be used to sort an array `a`, of which the first `n` elements are occupied, as follows:

```
    qsort(a, n, sizeof(int), intcompare);
```

`qsort` suggests that quicksort is the algorithm implemented in this library function, although this is usually irrelevant to the user.

4.3 Heapsort: Fast Sorting via Data Structures

Sorting is a natural laboratory for studying algorithm design paradigms, since many useful techniques lead to interesting sorting algorithms. The next several sections will introduce algorithmic design techniques motivated by particular sorting algorithms.

The alert reader should ask why we review the standard sorting when you are better off *not* implementing them and using built-in library functions instead. The answer is that the design techniques are very important for other algorithmic problems you are likely to encounter.

We start with data structure design, because one of the most dramatic algorithmic improvements via appropriate data structures occurs in sorting. Selection sort is a simple-to-code algorithm that repeatedly extracts the smallest remaining element from the unsorted part of the set:

```
SelectionSort(A)
    For i = 1 to n do
        Sort[i] = Find-Minimum from A
        Delete-Minimum from A
    Return(Sort)
```

A C language implementation of selection sort appeared back in Section 2.5.1 (page 41). There we partitioned the input array into sorted and unsorted regions. To find the smallest item, we performed a linear sweep through the unsorted portion of the array. The smallest item is then swapped with the ith item in the array before moving on to the next iteration. Selection sort performs n iterations, where the average iteration takes $n/2$ steps, for a total of $O(n^2)$ time.

But what if we improve the data structure? It takes $O(1)$ time to remove a particular item from an unsorted array once it has been located, but $O(n)$ time to find the smallest item. These are exactly the operations supported by priority queues. So what happens if we replace the data structure with a better priority queue implementation, either a heap or a balanced binary tree? Operations within the loop now take $O(\log n)$ time each, instead of $O(n)$. Using such a priority queue implementation speeds up selection sort from $O(n^2)$ to $O(n \log n)$.

The name typically given to this algorithm, *heapsort*, obscures the relationship between them, but heapsort is nothing but an implementation of selection sort using the right data structure.

4.3.1 Heaps

Heaps are a simple and elegant data structure for efficiently supporting the priority queue operations insert and extract-min. They work by maintaining a partial order on the set of elements which is weaker than the sorted order (so it can be efficient to maintain) yet stronger than random order (so the minimum element can be quickly identified).

Power in any hierarchically-structured organization is reflected by a tree, where each node in the tree represents a person, and edge (x, y) implies that x directly supervises (or dominates) y. The fellow at the root sits at the "top of the heap."

In this spirit, a *heap-labeled tree* is defined to be a binary tree such that the key labeling of each node *dominates* the key labeling of each of its children. In a

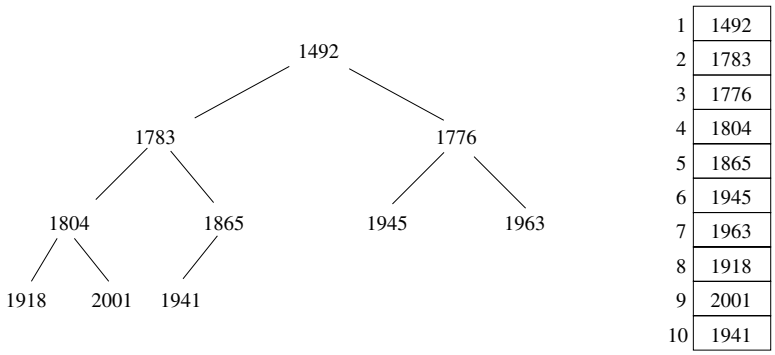

Figure 4.2: A heap-labeled tree of important years from American history (l), with the corresponding implicit heap representation (r)

min-heap, a node dominates its children by containing a smaller key than they do, while in a *max-heap* parent nodes dominate by being bigger. Figure 4.2(l) presents a min-heap ordered tree of red-letter years in American history (kudos to you if you can recall what happened each year).

The most natural implementation of this binary tree would store each key in a node with pointers to its two children. As with binary search trees, the memory used by the pointers can easily outweigh the size of keys, which is the data we are really interested in.

The heap is a slick data structure that enables us to represent binary trees without using any pointers. We will store data as an array of keys, and use the position of the keys to *implicitly* satisfy the role of the pointers.

We will store the root of the tree in the first position of the array, and its left and right children in the second and third positions, respectively. In general, we will store the 2^l keys of the lth level of a complete binary tree from left-to-right in positions 2^{l-1} to $2^l - 1$, as shown in Figure 4.2(r). We assume that the array starts with index 1 to simplify matters.

```
typedef struct {
        item_type q[PQ_SIZE+1];       /* body of queue */
        int n;                        /* number of queue elements */
} priority_queue;
```

What is especially nice about this representation is that the positions of the parent and children of the key at position k are readily determined. The *left* child of k sits in position $2k$ and the right child in $2k + 1$, while the parent of k holds court in position $\lfloor n/2 \rfloor$. Thus we can move around the tree without any pointers.

```
pq_parent(int n)
{
        if (n == 1) return(-1);
        else return((int) n/2);     /* implicitly take floor(n/2) */
}

pq_young_child(int n)
{
        return(2 * n);
}
```

So, we can store any binary tree in an array without pointers. What is the catch? Suppose our height h tree was sparse, meaning that the number of nodes $n < 2^h$. All missing internal nodes still take up space in our structure, since we must represent a full binary tree to maintain the positional mapping between parents and children.

Space efficiency thus demands that we not allow holes in our tree—i.e., that each level be packed as much as it can be. If so, only the last level may be incomplete. By packing the elements of the last level as far to the left as possible, we can represent an n-key tree using exactly n elements of the array. If we did not enforce these structural constraints, we might need an array of size 2^n to store the same elements. Since all but the last level is always filled, the height h of an n element heap is logarithmic because:

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$.

This implicit representation of binary trees saves memory, but is less flexible than using pointers. We cannot store arbitrary tree topologies without wasting large amounts of space. We cannot move subtrees around by just changing a single pointer, only by explicitly moving each of the elements in the subtree. This loss of flexibility explains why we cannot use this idea to represent binary search trees, but it works just fine for heaps.

Stop and Think: Who's where in the heap?

Problem: How can we efficiently search for a particular key in a heap?

Solution: We can't. Binary search does not work because a heap is not a binary search tree. We know almost nothing about the relative order of the $n/2$ leaf elements in a heap—certainly nothing that lets us avoid doing linear search through them. ∎

4.3.2 Constructing Heaps

Heaps can be constructed incrementally, by inserting each new element into the left-most open spot in the array, namely the $(n+1)$st position of a previously n-element heap. This ensures the desired balanced shape of the heap-labeled tree, but does not necessarily maintain the dominance ordering of the keys. The new key might be less than its parent in a min-heap, or greater than its parent in a max-heap.

The solution is to swap any such dissatisfied element with its parent. The old parent is now happy, because it is properly dominated. The other child of the old parent is still happy, because it is now dominated by an element even more extreme than its previous parent. The new element is now happier, but may still dominate its new parent. We now recur at a higher level, *bubbling up* the new key to its proper position in the hierarchy. Since we replace the root of a subtree by a larger one at each step, we preserve the heap order elsewhere.

```
pq_insert(priority_queue *q, item_type x)
{
  if (q->n >= PQ_SIZE)
         printf("Warning: priority queue overflow insert x=%d\n",x);
  else {
         q->n = (q->n) + 1;
         q->q[ q->n ] = x;
         bubble_up(q, q->n);
  }
}

bubble_up(priority_queue *q, int p)
{
    if (pq_parent(p) == -1) return; /* at root of heap, no parent */

    if (q->q[pq_parent(p)] > q->q[p]) {
            pq_swap(q,p,pq_parent(p));
            bubble_up(q, pq_parent(p));
    }
}
```

This swap process takes constant time at each level. Since the height of an n-element heap is $\lfloor \lg n \rfloor$, each insertion takes at most $O(\log n)$ time. Thus an initial heap of n elements can be constructed in $O(n \log n)$ time through n such insertions:

```
pq_init(priority_queue *q)
{
        q->n = 0;
}
```

```
make_heap(priority_queue *q, item_type s[], int n)
{
        int i;                          /* counter */

        pq_init(q);
        for (i=0; i<n; i++)
                pq_insert(q, s[i]);
}
```

4.3.3 Extracting the Minimum

The remaining priority queue operations are identifying and deleting the dominant element. Identification is easy, since the top of the heap sits in the first position of the array.

Removing the top element leaves a hole in the array. This can be filled by moving the element from the *right-most* leaf (sitting in the nth position of the array) into the first position.

The shape of the tree has been restored but (as after insertion) the labeling of the root may no longer satisfy the heap property. Indeed, this new root may be dominated by both of its children. The root of this min-heap should be the smallest of three elements, namely the current root and its two children. If the current root is dominant, the heap order has been restored. If not, the dominant child should be swapped with the root and the problem pushed down to the next level.

This dissatisfied element *bubbles down* the heap until it dominates all its children, perhaps by becoming a leaf node and ceasing to have any. This percolate-down operation is also called *heapify*, because it merges two heaps (the subtrees below the original root) with a new key.

```
item_type extract_min(priority_queue *q)
{
        int min = -1;                   /* minimum value */

        if (q->n <= 0) printf("Warning: empty priority queue.\n");
        else {
                min = q->q[1];

                q->q[1] = q->q[ q->n ];
                q->n = q->n - 1;
                bubble_down(q,1);
        }

        return(min);
}
```

```
bubble_down(priority_queue *q, int p)
{
        int c;                          /* child index */
        int i;                          /* counter */
        int min_index;                  /* index of lightest child */

        c = pq_young_child(p);
        min_index = p;

        for (i=0; i<=1; i++)
                if ((c+i) <= q->n) {
                   if (q->q[min_index] > q->q[c+i]) min_index = c+i;
                }

        if (min_index != p) {
                pq_swap(q,p,min_index);
                bubble_down(q, min_index);
        }
}
```

We will reach a leaf after $\lfloor \lg n \rfloor$ `bubble_down` steps, each constant time. Thus root deletion is completed in $O(\log n)$ time.

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \log n)$ sorting algorithm, named *Heapsort*.

```
heapsort(item_type s[], int n)
{
        int i;                  /* counters */
        priority_queue q;       /* heap for heapsort */

        make_heap(&q,s,n);

        for (i=0; i<n; i++)
                s[i] = extract_min(&q);
}
```

Heapsort is a great sorting algorithm. It is simple to program; indeed, the complete implementation has been presented above. It runs in worst-case $O(n \log n)$ time, which is the best that can be expected from any sorting algorithm. It is an *in-place* sort, meaning it uses no extra memory over the array containing the elements to be sorted. Although other algorithms prove slightly faster in practice, you won't go wrong using heapsort for sorting data that sits in the computer's main memory.

Priority queues are very useful data structures. Recall they were the hero of the war story described in Section 3.6 (page 85). A complete set of priority queue implementations is presented in catalog Section 12.2 (page 373).

4.3.4 Faster Heap Construction (*)

As we have seen, a heap can be constructed on n elements by incremental insertion in $O(n \log n)$ time. Surprisingly, heaps can be constructed even faster by using our `bubble_down` procedure and some clever analysis.

Suppose we pack the n keys destined for our heap into the first n elements of our priority-queue array. The shape of our heap will be right, but the dominance order will be all messed up. How can we restore it?

Consider the array in reverse order, starting from the last (nth) position. It represents a leaf of the tree and so dominates its nonexistent children. The same is the case for the last $n/2$ positions in the array, because all are leaves. If we continue to walk backwards through the array we will finally encounter an internal node with children. This element may not dominate its children, but its children represent well-formed (if small) heaps.

This is exactly the situation the `bubble_down` procedure was designed to handle, restoring the heap order of arbitrary root element sitting on top of two sub-heaps. Thus we can create a heap by performing $n/2$ non-trivial calls to the `bubble_down` procedure:

```
make_heap(priority_queue *q, item_type s[], int n)
{
        int i;                          /* counter */

        q->n = n;
        for (i=0; i<n; i++) q->q[i+1] = s[i];

        for (i=q->n; i>=1; i--) bubble_down(q,i);
}
```

Multiplying the number of calls to `bubble_down` (n) times an upper bound on the cost of each operation ($O(\log n)$) gives us a running time analysis of $O(n \log n)$. This would make it no faster than the incremental insertion algorithm described above.

But note that it is indeed an *upper bound*, because only the last insertion will actually take $\lfloor \lg n \rfloor$ steps. Recall that `bubble_down` takes time proportional to the height of the heaps it is merging. Most of these heaps are extremely small. In a full binary tree on n nodes, there are $n/2$ nodes that are leaves (i.e. , height 0), $n/4$

nodes that are height 1, $n/8$ nodes that are height 2, and so on. In general, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil h \leq n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \leq 2n$$

Since this sum is not quite a geometric series, we can't apply the usual identity to get the sum, but rest assured that the puny contribution of the numerator (h) is crushed by the denominator (2^h). The series quickly converges to linear.

Does it matter that we can construct heaps in linear time instead of $O(n \log n)$? Usually not. The construction time did not dominate the complexity of heapsort, so improving the construction time does not improve its worst-case performance. Still, it is an impressive display of the power of careful analysis, and the free lunch that geometric series convergence can sometimes provide.

Stop and Think: Where in the Heap?

Problem: Given an array-based heap on n elements and a real number x, efficiently determine whether the kth smallest element in the heap is greater than or equal to x. Your algorithm should be $O(k)$ in the worst-case, independent of the size of the heap. Hint: you do not have to find the kth smallest element; you need only determine its relationship to x.

Solution: There are at least two different ideas that lead to correct but inefficient algorithms for this problem:

1. Call extract-min k times, and test whether all of these are less than x. This explicitly sorts the first k elements and so gives us more information than the desired answer, but it takes $O(k \log n)$ time to do so.

2. The kth smallest element cannot be deeper than the kth level of the heap, since the path from it to the root must go through elements of decreasing value. Thus we can look at all the elements on the first k levels of the heap, and count how many of them are less than x, stopping when we either find k of them or run out of elements. This is correct, but takes $O(\min(n, 2^k))$ time, since the top k elements have 2^k elements.

An $O(k)$ solution can look at only k elements smaller than x, plus at most $O(k)$ elements greater than x. Consider the following recursive procedure, called at the root with $i = 1$ with $count = k$:

```
int heap_compare(priority_queue *q, int i, int count, int x)
{
        if ((count <= 0) || (i > q->n) return(count);

        if (q->q[i] < x) {
            count = heap_compare(q, pq_young_child(i), count-1, x);
            count = heap_compare(q, pq_young_child(i)+1, count, x);
        }

        return(count);
}
```

If the root of the min-heap is $\geq x$, then no elements in the heap can be less than x, as by definition the root must be the smallest element. This procedure searches the children of all nodes of weight smaller than x until either (a) we have found k of them, when it returns 0, or (b) they are exhausted, when it returns a value greater than zero. Thus it will find enough small elements if they exist.

But how long does it take? The only nodes whose children we look at are those $< x$, and at most k of these in total. Each have at most visited two children, so we visit at most $3k$ nodes, for a total time of $O(k)$. ∎

4.3.5 Sorting by Incremental Insertion

Now consider a different approach to sorting via efficient data structures. Select an arbitrary element from the unsorted set, and put it in the proper position in the sorted set.

```
InsertionSort(A)
    A[0] = -∞
    for i = 2 to n do
        j = i
        while (A[j] < A[j - 1]) do
            swap(A[j], A[j - 1])
            j = j - 1
```

A C language implementation of insertion sort appeared in Section 2.5.2 (page 43). Although insertion sort takes $O(n^2)$ in the worst case, it performs considerably better if the data is almost sorted, since few iterations of the inner loop suffice to sift it into the proper position.

Insertion sort is perhaps the simplest example of the *incremental insertion* technique, where we build up a complicated structure on n items by first building it on $n-1$ items and then making the necessary changes to add the last item. Incremental insertion proves a particularly useful technique in geometric algorithms.

Note that faster sorting algorithms based on incremental insertion follow from more efficient data structures. Insertion into a balanced search tree takes $O(\log n)$ per operation, or a total of $O(n \log n)$ to construct the tree. An in-order traversal reads through the elements in sorted order to complete the job in linear time.

4.4 War Story: Give me a Ticket on an Airplane

I came into this particular job seeking justice. I'd been retained by an air travel company to help design an algorithm to find the cheapest available airfare from city x to city y. Like most of you, I suspect, I'd been baffled at the crazy price fluctuations of ticket prices under modern "yield management." The price of flights seems to soar far more efficiently than the planes themselves. The problem, it seemed to me, was that airlines never wanted to show the true cheapest price. By doing my job right, I could make damned sure they would show it to me next time.

"Look," I said at the start of the first meeting. "This can't be so hard. Construct a graph with vertices corresponding to airports, and add an edge between each airport pair (u, v) which shows a direct flight from u to v. Set the weight of this edge equal to the cost of the cheapest available ticket from u to v. Now the cheapest fair from x to y is given by the shortest x-y path in this graph. This path/fare can be found using Dijkstra's shortest path algorithm. Problem solved!" I announced, waiving my hand with a flourish.

The assembled cast of the meeting nodded thoughtfully, then burst out laughing. It was I who needed to learn something about the overwhelming complexity of air travel pricing. There are literally millions of different fares available at any time, with prices changing several times daily. Restrictions on the availability of a particular fare in a particular context is enforced by a complicated set of pricing rules. These rules are an industry-wide kludge—a complicated structure with little in the way of consistent logical principles, which is exactly what we would need to search efficiently for the minimum fare. My favorite rule exceptions applied only to the country of Malawi. With a population of only 12 million and per-capita income of $596 (179th in the world), they prove to be an unexpected powerhouse shaping world aviation price policy. Accurately pricing any air itinerary requires at least implicit checks to ensure the trip doesn't take us through Malawi.

Part of the real problem is that there can easily be 100 different fares for the first flight leg, say from Los Angeles (LAX) to Chicago (ORD), and a similar number for each subsequent leg, say from Chicago to New York (JFK). The cheapest possible LAX-ORD fare (maybe an AARP children's price) might not be combinable with the cheapest ORD-JFK fare (perhaps a pre-Ramadan special that can only be used with subsequent connections to Mecca).

After being properly chastised for oversimplifying the problem, I got down to work. I started by reducing the problem to the simplest interesting case. "So, you

X	Y
$100	$50
$110	$75
$130	$125

X+Y	
$150	(1,1)
$160	(2,1)
$175	(1,2)
$180	(3,1)
$185	(2,2)
$205	(2,3)
$225	(1,3)
$235	(2,3)
$255	(3,3)

Figure 4.3: Sorting the pairwise sums of lists X and Y.

need to find the cheapest two-hop fare that passes your rule tests. Is there a way to decide in advance which pairs will pass without explicitly testing them?"

"No, there is no way to tell," they assured me. "We can only consult a black box routine to decide whether a particular price is available for the given itinerary/travelers."

"So our goal is to call this black box on the fewest number of combinations. This means evaluating all possible fare combinations in order from cheapest to most expensive, and stopping as soon as we encounter the first legal combination."

"Right."

"Why not construct the $m \times n$ possible price pairs, sort them in terms of cost, and evaluate them in sorted order? Clearly this can be done in $O(nm \log(nm))$ time."[1]

"That is basically what we do now, but it is quite expensive to construct the full set of $m \times n$ pairs, since the first one might be all we need."

I caught a whiff of an interesting problem. "So what you really want is an efficient data structure to repeatedly return the *next* most expensive pair without constructing all the pairs in advance."

This was indeed an interesting problem. Finding the largest element in a set under insertion and deletion is *exactly* what priority queues are good for. The catch here is that we could not seed the priority queue with all values in advance. We had to insert new pairs into the queue after each evaluation.

I constructed some examples, like the one in Figure 4.3. We could represent each fare by the list indexes of its two components. The cheapest single fare will certainly be constructed by adding up the cheapest component from both lists,

[1]The question of whether all such sums can be sorted faster than nm arbitrary integers is a notorious open problem in algorithm theory. See [Fre76, Lam92] for more on $X + Y$ sorting, as the problem is known.

described $(1,1)$. The second cheapest fare would be made from the head of one list and the second element of another, and hence would be either $(1,2)$ or $(2,1)$. Then it gets more complicated. The third cheapest could either be the unused pair above or $(1,3)$ or $(3,1)$. Indeed it would have been $(3,1)$ in the example above if the third fare of X had been \$120.

"Tell me," I asked. "Do we have time to sort the two respective lists of fares in increasing order?"

"Don't have to." the leader replied. "They come out in sorted order from the database."

Good news. That meant there was a natural order to the pair values. We never need to evaluate the pairs $(i+1,j)$ or $(i,j+1)$ before (i,j), because they clearly define more expensive fares.

"Got it!," I said. "We will keep track of index pairs in a priority queue, with the sum of the fare costs as the key for the pair. Initially we put only pair $(1,1)$ on the queue. If it proves it is not feasible, we put its two successors on—namely $(1,2)$ and $(2,1)$. In general, we enqueue pairs $(i+1,j)$ and $(i,j+1)$ after evaluating/rejecting pair (i,j). We will get through all the pairs in the right order if we do so."

The gang caught on quickly. "Sure. But what about duplicates? We will construct pair (x,y) two different ways, both when expanding $(x-1,y)$ and $(x,y-1)$."

"You are right. We need an extra data structure to guard against duplicates. The simplest might be a hash table to tell us whether a given pair exists in the priority queue before we insert a duplicate. In fact, we will never have more than n active pairs in our data structure, since there can only be one pair for each distinct value of the first coordinate."

And so it went. Our approach naturally generalizes to itineraries with more than two legs, (a complexity which grows with the number of legs). The best-first evaluation inherent in our priority queue enabled the system to stop as soon as it found the provably cheapest fare. This proved to be fast enough to provide interactive response to the user. That said, I haven't noticed my travel tickets getting any cheaper.

4.5 Mergesort: Sorting by Divide-and-Conquer

Recursive algorithms reduce large problems into smaller ones. A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements. This algorithm is called *mergesort*, recognizing the importance of the interleaving operation:

Mergesort($A[1,n]$)
    Merge(MergeSort($A[1,\lfloor n/2 \rfloor]$), MergeSort($A[\lfloor n/2 \rfloor + 1, n]$))

The basis case of the recursion occurs when the subarray to be sorted consists of a single element, so no rearrangement is possible. A trace of the execution of

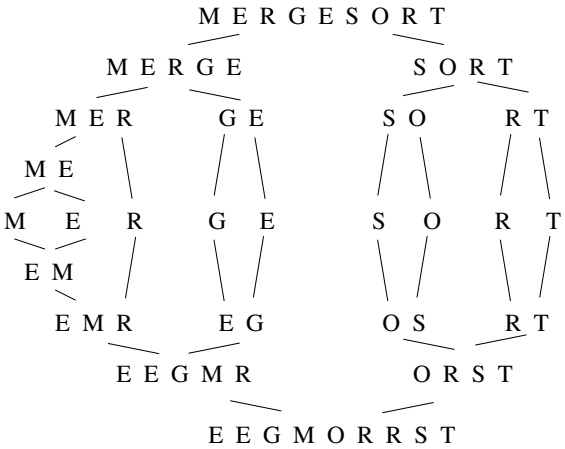

Figure 4.4: Animation of mergesort in action

mergesort is given in Figure 4.4. Picture the action as it happens during an in-order traversal of the top tree, with the array-state transformations reported in the bottom, reflected tree.

The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list. We could concatenate them into one list and call heapsort or some other sorting algorithm to do it, but that would just destroy all the work spent sorting our component lists.

Instead we can *merge* the two lists together. Observe that the smallest overall item in two lists sorted in increasing order (as above) must sit at the top of one of the two lists. This smallest element can be removed, leaving two sorted lists behind—one slightly shorter than before. The second smallest item overall must be atop one of these lists. Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most $n-1$ comparisons or $O(n)$ total work.

What is the total running time of mergesort? It helps to think about how much work is done at each level of the execution tree. If we assume for simplicity that n is a power of two, the kth level consists of all the 2^k calls to `mergesort` processing subranges of $n/2^k$ elements.

The work done on the $(k=0)$th level involves merging two sorted lists, each of size $n/2$, for a total of at most $n-1$ comparisons. The work done on the $(k=1)$th level involves merging two pairs of sorted lists, each of size $n/4$, for a total of at most $n-2$ comparisons. In general, the work done on the kth level involves merging 2^k pairs sorted list, each of size $n/2^{k+1}$, for a total of at most $n-2^k$ comparisons. *Linear work is done merging all the elements on each level.* Each of the n elements

appears in exactly one subproblem on each level. The most expensive case (in terms of comparsions) is actually the top level.

The number of elements in a subproblem gets halved at each level. Thus the number of times we can halve n until we get to 1 is $\lceil \lg_2 n \rceil$. Because the recursion goes $\lg n$ levels deep, and a linear amount of work is done per level, mergesort takes $O(n \log n)$ time in the worst case.

Mergesort is a great algorithm for sorting linked lists, because it does not rely on random access to elements as does heapsort or quicksort. Its primary disadvantage is the need for an auxilliary buffer when sorting arrays. It is easy to merge two sorted linked lists without using any extra space, by just rearranging the pointers. However, to merge two sorted arrays (or portions of an array), we need use a third array to store the result of the merge to avoid stepping on the component arrays. Consider merging $\{4, 5, 6\}$ with $\{1, 2, 3\}$, packed from left to right in a single array. Without a buffer, we would overwrite the elements of the top half during merging and lose them.

Mergesort is a classic divide-and-conquer algorithm. We are ahead of the game whenever we can break one large problem into two smaller problems, because the smaller problems are easier to solve. The trick is taking advantage of the two partial solutions to construct a solution of the full problem, as we did with the merge operation.

Implementation

The divide-and-conquer `mergesort` routine follows naturally from the pseudocode:

```
mergesort(item_type s[], int low, int high)
{
        int i;                  /* counter */
        int middle;             /* index of middle element */

        if (low < high) {
                middle = (low+high)/2;
                mergesort(s,low,middle);
                mergesort(s,middle+1,high);
                merge(s, low, middle, high);
        }
}
```

More challenging turns out to be the details of how the merging is done. The problem is that we must put our merged array somewhere. To avoid losing an element by overwriting it in the course of the merge, we first copy each subarray to a separate queue and merge these elements back into the array. In particular:

```
merge(item_type s[], int low, int middle, int high)
{
  int i;                    /* counter */
  queue buffer1, buffer2; /* buffers to hold elements for merging */

  init_queue(&buffer1);
  init_queue(&buffer2);

  for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
  for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

  i = low;
  while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
          if (headq(&buffer1) <= headq(&buffer2))
                  s[i++] = dequeue(&buffer1);
          else
                  s[i++] = dequeue(&buffer2);
  }

  while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
  while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

4.6 Quicksort: Sorting by Randomization

Suppose we select a random item p from the n items we seek to sort. *Quicksort* (shown in action in Figure 4.5) separates the $n-1$ other items into two piles: a low pile containing all the elements that appear before p in sorted order and a high pile containing all the elements that appear after p in sorted order. Low and high denote the array positions we place the respective piles, leaving a single slot between them for p.

Such partitioning buys us two things. First, the pivot element p ends up in the exact array position it will reside in the the final sorted order. Second, after partitioning no element flops to the other side in the final sorted order. *Thus we can now sort the elements to the left and the right of the pivot independently!* This gives us a recursive sorting algorithm, since we can use the partitioning approach to sort each subproblem. The algorithm must be correct since each element ultimately ends up in the proper position:

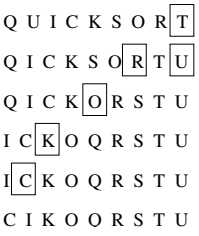

Figure 4.5: Animation of quicksort in action

```
quicksort(item_type s[], int l, int h)
{
        int p;                  /* index of partition */

        if ((h-l)>0) {
                p = partition(s,l,h);
                quicksort(s,l,p-1);
                quicksort(s,p+1,h);
        }
}
```

We can partition the array in one linear scan for a particular pivot element by maintaining three sections of the array: less than the pivot (to the left of `firsthigh`), greater than or equal to the pivot (between `firsthigh` and `i`), and unexplored (to the right of `i`), as implemented below:

```
int partition(item_type s[], int l, int h)
{
        int i;              /* counter */
        int p;              /* pivot element index */
        int firsthigh;      /* divider position for pivot element */

        p = h;
        firsthigh = l;
        for (i=l; i<h; i++)
                if (s[i] < s[p]) {
                        swap(&s[i],&s[firsthigh]);
                        firsthigh ++;
                }
        swap(&s[p],&s[firsthigh]);

        return(firsthigh);
}
```

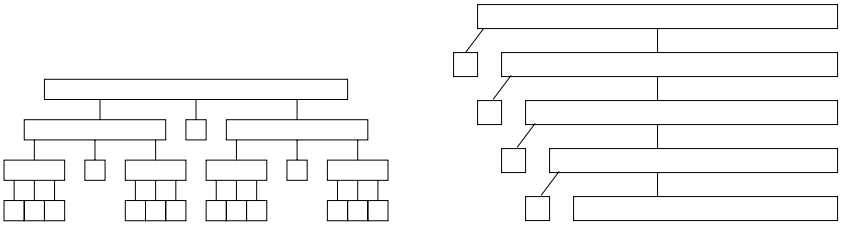

Figure 4.6: The best-case (l) and worst-case (r) recursion trees for quicksort

Since the partitioning step consists of at most n swaps, it takes linear time in the number of keys. But how long does the entire quicksort take? As with mergesort, quicksort builds a recursion tree of nested subranges of the n-element array. As with mergesort, quicksort spends linear time processing (now `partition`ing instead of `merge`ing) the elements in each subarray on each level. As with mergesort, quicksort runs in $O(n \cdot h)$ time, where h is the height of the recursion tree.

The difficulty is that the height of the tree depends upon where the pivot element ends up in each partition. If we get very lucky and *happen* to repeatedly pick the median element as our pivot, the subproblems are always half the size of the previous level. The height represents the number of times we can halve n until we get down to 1, or at most $\lceil \lg_2 n \rceil$. This happy situation is shown in Figure 4.6(l), and corresponds to the best case of quicksort.

Now suppose we consistently get unlucky, and our pivot element always splits the array as unequally as possible. This implies that the pivot element is always the biggest or smallest element in the sub-array. After this pivot settles into its position, we are left with one subproblem of size $n - 1$. We spent linear work and reduced the size of our problem by one measly element, as shown in Figure 4.6(r). It takes a tree of height $n - 1$ to chop our array down to one element per level, for a worst case time of $\Theta(n^2)$.

Thus, the worst case for quicksort is worse than heapsort or mergesort. To justify its name, quicksort had better be good in the average case. Understanding why requires some intuition about random sampling.

4.6.1 Intuition: The Expected Case for Quicksort

The expected performance of quicksort depends upon the height of the partition tree constructed by random pivot elements at each step. Mergesort ran in $O(n \log n)$ time because we split the keys into two equal halves, sorted them recursively, and then merged the halves in linear time. Thus, whenever our pivot element is near the center of the sorted array (i.e. , the pivot is close to the median element), we get a good split and realize the same performance as mergesort.

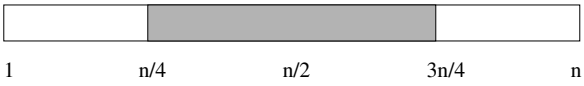

Figure 4.7: Half the time, the pivot is close to the median element

I will give an intuitive explanation of why quicksort is $O(n \log n)$ in the average case. How likely is it that a randomly selected pivot is a good one? The best possible selection for the pivot would be the median key, because exactly half of elements would end up left, and half the elements right, of the pivot. Unfortunately, we only have a probability of $1/n$ of randomly selecting the median as pivot, which is quite small.

Suppose a key is a *good enough* pivot if it lies is in the center half of the sorted space of keys—i.e. , those ranked from $n/4$ to $3n/4$ in the space of all keys to be sorted. Such *good enough* pivot elements are quite plentiful, since half the elements lie closer to the middle than one of the two ends (see Figure 4.7). Thus, on each selection we will pick a *good enough* pivot with probability of $1/2$.

Can you flip a coin so it comes up tails each time? Not without cheating. If you flip a fair coin n times, it will come out heads about half the time. Let heads denote the chance of picking a *good enough* pivot.

The worst possible *good enough* pivot leaves the bigger half of the space partition with $3n/4$ items. What is the height h_g of a quicksort partition tree constructed repeatedly from the worst-possible *good enough* pivot? The deepest path through this tree passes through partitions of size $n, (3/4)n, (3/4)^2 n, \ldots$, down to 1. How many times can we multiply n by $3/4$ until it gets down to 1?

$$(3/4)^{h_g} n = 1 \Rightarrow n = (4/3)^{h_g}$$

so $h_g = \log_{4/3} n$.

But only half of all randomly selected pivots will be *good enough*. The rest we classify as *bad*. The worst of these bad pivots will do essentially nothing to reduce the partition size along the deepest path. The deepest path from the root through a typical randomly-constructed quicksort partition tree will pass through roughly equal numbers of good-enough and bad pivots. Since the expected number of good splits and bad splits is the same, the bad splits can only double the height of the tree, so $h \approx 2h_g = 2 \log_{4/3} n$, which is clearly $\Theta(\log n)$.

On average, random quicksort partition trees (and by analogy, binary search trees under random insertion) are very good. More careful analysis shows the average height after n insertions is approximately $2 \ln n$. Since $2 \ln n \approx 1.386 \lg_2 n$, this is only 39% taller than a perfectly balanced binary tree. Since quicksort does $O(n)$ work partitioning on each level, the average time is $O(n \log n)$. If we are *extremely* unlucky and our randomly selected elements always are among the largest or smallest element in the array, quicksort turns into selection sort and runs in $O(n^2)$. However, the odds against this are vanishingly small.

4.6.2 Randomized Algorithms

There is an important subtlety about the expected case $O(n \log n)$ running time for quicksort. Our quicksort implementation above selected the last element in each sub-array as the pivot. Suppose this program were given a sorted array as input. If so, at each step it would pick the worst possible pivot and run in quadratic time.

For any deterministic method of pivot selection, there exists a worst-case input instance which will doom us to quadratic time. The analysis presented above made no claim stronger than:

> "Quicksort runs in $\Theta(n \log n)$ time, with high probability, *if* you give me randomly ordered data to sort."

But now suppose we add an initial step to our algorithm where we randomly permute the order of the n elements before we try to sort them. Such a permutation can be constructed in $O(n)$ time (see Section 13.7 for details). This might seem like wasted effort, but it provides the guarantee that we can expect $\Theta(n \log n)$ running time *whatever* the initial input was. The worst case performance still can happen, but it depends only upon how unlucky we are. There is no longer a well-defined "worst case" input. We now can say

> "Randomized quicksort runs in $\Theta(n \log n)$ time on *any* input, with high probability."

Alternately, we could get the same guarantee by selecting a random element to be the pivot at each step.

Randomization is a powerful tool to improve algorithms with bad worst-case but good average-case complexity. It can be used to make algorithms more robust to boundary cases and more efficient on highly structured input instances that confound heuristic decisions (such as sorted input to quicksort). It often lends itself to simple algorithms that provide randomized performance guarantees which are otherwise obtainable only using complicated deterministic algorithms.

Proper analysis of randomized algorithms requires some knowledge of probability theory, and is beyond the scope of this book. However, some of the approaches to designing efficient randomized algorithms are readily explainable:

- *Random sampling* – Want to get an idea of the median value of n things but don't have either the time or space to look at them all? Select a small random sample of the input and study those, for the results should be representative.

  This is the idea behind opinion polling. Biases creep in unless you take a truly *random* sample, as opposed to the first x people you happen to see. To avoid bias, actual polling agencies typically dial random phone numbers and hope someone answers.

- *Randomized hashing* – We have claimed that hashing can be used to implement dictionary operations in $O(1)$ "expected-time." However, for any hash

function there is a given worst-case set of keys that all get hashed to the same bucket. But now suppose we randomly select our hash function from a large family of good ones as the first step of our algorithm. We get the same type of improved guarantee that we did with randomized quicksort.

- *Randomized search* – Randomization can also be used to drive search techniques such as simulated annealing, as will be discussed in detail in Section 7.5.3 (page 254).

Stop and Think: Nuts and Bolts

Problem: The *nuts and bolts* problem is defined as follows. You are given a collection of n bolts of different widths, and n corresponding nuts. You can test whether a given nut and bolt fit together, from which you learn whether the nut is too large, too small, or an exact match for the bolt. The differences in size between pairs of nuts or bolts are too small to see by eye, so you cannot compare the sizes of two nuts or two bolts directly. You are to match each bolt to each nut.

Give an $O(n^2)$ algorithm to solve the nuts and bolts problem. Then give a randomized $O(n \log n)$ expected time algorithm for the same problem.

Solution: The brute force algorithm for matching nuts and bolts starts with the first bolt and compares it to each nut until we find a match. In the worst case, this will require n comparisons. Repeating this for each successive bolt on all remaining nuts yields a quadratic-comparison algorithm.

What if we pick a random bolt and try it? On average, we would expect to get about halfway through the set of nuts before we found the match, so this randomized algorithm would do half the work as the worst case. That counts as some kind of improvement, although not an asymptotic one.

Randomized quicksort achieves the desired expected-case running time, so a natural idea is to emulate it on the nuts and bolts problem. Indeed, sorting both the nuts and bolts by size would yield a matching, since the ith largest nut must match the ith largest bolt.

The fundamental step in quicksort is partitioning elements around a pivot. Can we partition nuts and bolts around a randomly selected bolt b? Certainly we can partition the nuts into those of size less than b and greater than b. But decomposing the problem into two halves requires partitioning the bolts as well, and we cannot compare bolt against bolt. But once we find the matching nut to b we can use it to partition the bolts accordingly. In $2n - 2$ comparisons, we partition the nuts and bolts, and the remaining analysis follows directly from randomized quicksort.

What is interesting about this problem is that no simple deterministic algorithm for nut and bolt sorting is known. It illustrates how randomization makes the bad case go away, leaving behind a simple and beautiful algorithm. ∎