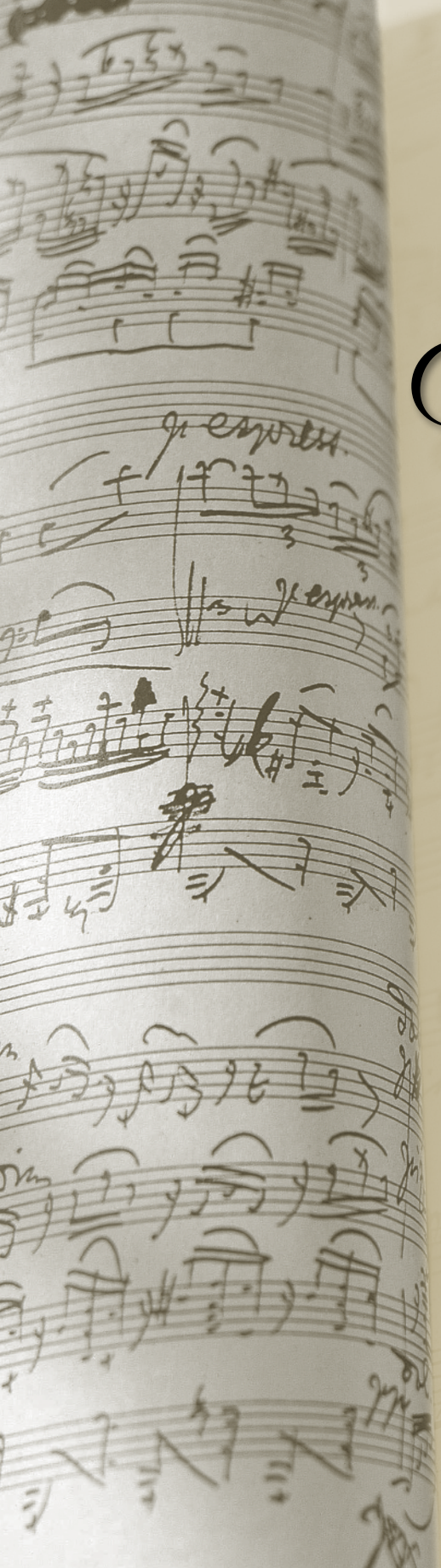


# Algorithms

Sanjoy Dasgupta
Christos Papadimitriou
Umesh Vazirani

# Algorithms

**Sanjoy Dasgupta**
*University of California, San Diego*

**Christos Papadimitriou**
*University of California at Berkeley*

**Umesh Vazirani**
*University of California at Berkeley*


**Higher Education**

Boston Burr Ridge, IL Dubuque, IA New York San Francisco
St. Louis Bangkok Bogotá Caracas Kuala Lumpur Lisbon London
Lisbon London Madrid Mexico City Milan Montreal New Delhi
Santiago Seoul Singapore Sydney Taipei Toronto

The McGraw-Hill Companies

Mc Graw Hill Higher Education

ALGORITHMS



This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9 8 7 6

ISBN 978-0-07-352340-8
MHID 0-07-352340-2

Publisher: *Alan R. Apt*
Executive Marketing Manager: *Michael Weitz*
Project Manager: *Joyce Watters*
Lead Production Supervisor: *Sandy Ludovissy*
Associate Media Producer: *Christina Nelson*
Designer: *John Joran*
Compositor: *Techbooks*
Typeface: *10/12 Slimbach*
Printer: *R. R. Donnelley Crawfordsville, IN*

**Library of Congress Cataloging-in-Publication Data**

Dasgupta Sanjoy.
Algorithms / Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.—1st ed.
p. cm.
Includes index.
ISBN 978-0-07-352340-8 — ISBN 0-07-352340-2
1. Algorithms—Textbooks. 2. Computer algorithms—Textbooks. I. Papadimitriou, Christos H.
II. Vazirani, Umesh Virkumar. III. Title.

QA9.58.D37 2008
518′1—dc22 2006049014
CIP

www.mhhe.com

To our students and teachers,
and our parents.

# Contents

# Boxes

# Preface

This book evolved over the past ten years from a set of lecture notes developed while teaching the undergraduate Algorithms course at Berkeley and U.C. San Diego. Our way of teaching this course evolved tremendously over these years in a number of directions, partly to address our students' background (undeveloped formal skills outside of programming), and partly to reflect the maturing of the field in general, as we have come to see it. The notes increasingly crystallized into a narrative, and we progressively structured the course to emphasize the "story line" implicit in the progression of the material. As a result, the topics were carefully selected and clustered. No attempt was made to be encyclopedic, and this freed us to include topics traditionally de-emphasized or omitted from most Algorithms books.

Playing on the strengths of our students (shared by most of today's undergraduates in Computer Science), instead of dwelling on formal proofs we distilled in each case the crisp mathematical idea that makes the algorithm work. In other words, we emphasized rigor over formalism. We found that our students were much more receptive to mathematical rigor of this form. It is this progression of crisp ideas that helps weave the story.

Once you think about Algorithms in this way, it makes sense to start at the historical beginning of it all, where, in addition, the characters are familiar and the contrasts dramatic: numbers, primality, and factoring. This is the subject of Part I of the book, which also includes the RSA cryptosystem, and divide-and-conquer algorithms for integer multiplication, sorting and median finding, as well as the fast Fourier transform. There are three other parts: Part II, the most traditional section of the book, concentrates on data structures and graphs; the contrast here is between the intricate structure of the underlying problems and the short and crisp pieces of pseudocode that solve them. Instructors wishing to teach a more traditional course can simply start with Part II, which is self-contained (following the prologue), and then cover Part I as required. In Parts I and II we introduced certain techniques (such as greedy and divide-and-conquer) which work for special kinds of problems; Part III deals with the "sledgehammers" of the trade, techniques that are powerful and general: dynamic programming (a novel approach helps clarify this traditional stumbling block for students) and linear programming (a clean and intuitive treatment of the simplex algorithm, duality, and reductions to the basic problem). The final Part IV is about ways of dealing with hard problems: NP-completeness, various heuristics, as well as quantum algorithms, perhaps the most advanced and modern topic. As it happens, we end the story exactly where we started it, with Shor's quantum algorithm for factoring.

The book includes three additional undercurrents, in the form of three series of separate "boxes," strengthening the narrative (and addressing variations in the needs and interests of the students) while keeping the flow intact, pieces that provide historical context; descriptions of how the explained algorithms are used in practice (with emphasis on internet applications); and excursions for the mathematically sophisticated.

Many of our colleagues have made crucial contributions to this book. We are grateful for feedback from Dimitris Achlioptas, Dorit Aharanov, Mike Clancy, Jim Demmel, Monika Henzinger, Mike Jordan, Milena Mihail, Gene Myers, Dana Randall, Satish Rao, Tim Roughgarden, Jonathan Shewchuk, Martha Sideri, Alistair Sinclair, and David Wagner, all of whom beta tested early drafts. Satish Rao, Leonard Schulman, and Vijay Vazirani shaped the exposition of several key sections. Gene Myers, Satish Rao, Luca Trevisan, Vijay Vazirani, and Lofti Zadeh provided exercises. And finally, there are the students of UC Berkeley and, later, UC San Diego, who inspired this project, and who have seen it through its many incarnations.

*Chapter 0*

# Prologue

Look around you. Computers and networks are everywhere, enabling an intricate web of complex human activities: education, commerce, entertainment, research, manufacturing, health management, human communication, even war. Of the two main technological underpinnings of this amazing proliferation, one is obvious: the breathtaking pace with which advances in microelectronics and chip design have been bringing us faster and faster hardware.

This book tells the story of the other intellectual enterprise that is crucially fueling the computer revolution: *efficient algorithms.* It is a fascinating story.

*Gather 'round and listen close.*

## 0.1 Books and algorithms

Two ideas changed the world. In 1448 in the German city of Mainz a goldsmith named Johann Gutenberg discovered a way to print books by putting together movable metallic pieces. Literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened. Many historians say we owe all this to typography. Imagine a world in which only an elite could read these lines! But others insist that the key development was not typography, but *algorithms.*

Johann Gutenberg
1398–1468

Today we are so used to writing numbers in decimal, that it is easy to forget that Gutenberg would write the number 1448 as MCDXLVIII. How do you add two Roman numerals? What is MCDXLVIII + DCCCXII? (And just try to think about multiplying them.) Even a clever man like Gutenberg probably only knew how to add and subtract small numbers using his fingers; for anything more complicated he had to consult an abacus specialist.

The decimal system, invented in India around AD 600, was a revolution in quantitative reasoning: using only 10 symbols, even very large numbers could be written down compactly, and arithmetic could be done efficiently on them by following elementary steps. Nonetheless these ideas took a long time to spread, hindered by traditional barriers of language, distance, and ignorance. The most influential medium of transmission turned out to be a textbook, written in Arabic in the ninth century by a man who lived in Baghdad. Al Khwarizmi laid out the basic methods for adding, multiplying, and dividing numbers—even extracting square roots and calculating digits of $\pi$. These procedures were precise, unambiguous, mechanical, efficient, correct—in short, they were *algorithms*, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.

Since then, this decimal positional system and its numerical algorithms have played an enormous role in Western civilization. They enabled science and technology; they accelerated industry and commerce. And when, much later, the computer was finally designed, it explicitly embodied the positional system in its bits and words and arithmetic unit. Scientists everywhere then got busy developing more and more complex algorithms for all kinds of problems and inventing novel applications—ultimately changing the world.

## 0.2 Enter Fibonacci

Al Khwarizmi's work could not have gained a foothold in the West were it not for the efforts of one man: the 13th century Italian mathematician Leonardo Fibonacci, who saw the potential of the positional system and worked hard to develop it further and propagandize it.

But today Fibonacci is most widely known for his famous sequence of numbers

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

each the sum of its two immediate predecessors. More formally, the Fibonacci numbers $F_n$ are generated by the simple rule

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

No other sequence of numbers has been studied as extensively, or applied to more fields: biology, demography, art, architecture, music, to name just a few. And, together with the powers of 2, it is computer science's favorite sequence.

In fact, the Fibonacci numbers grow *almost* as fast as the powers of 2: for example, $F_{30}$ is over a million, and $F_{100}$ is already 21 digits long! In general, $F_n \approx 2^{0.694n}$ (see Exercise 0.3).

But what is the precise value of $F_{100}$, or of $F_{200}$? Fibonacci himself would surely have wanted to know such things. To answer, we need an algorithm for computing the $n$th Fibonacci number.

Leonardo of Pisa (Fibonacci)
1170–1250

© Corbis

## An exponential algorithm

One idea is to slavishly implement the recursive definition of $F_n$. Here is the resulting algorithm, in the "pseudocode" notation used throughout this book:

```
function fib1(n)
if n = 0: return 0
if n = 1: return 1
return fib1(n − 1) + fib1(n − 2)
```

Whenever we have an algorithm, there are three questions we always ask about it:

1. Is it correct?
2. How much time does it take, as a function of $n$?
3. And can we do better?

The first question is moot here, as this algorithm is precisely Fibonacci's definition of $F_n$. But the second demands an answer. Let $T(n)$ be the number of *computer steps* needed to compute `fib1`$(n)$; what can we say about this function? For starters, if $n$ is less than 2, the procedure halts almost immediately, after just a couple of steps. Therefore,

$$T(n) \leq 2 \quad \text{for } n \leq 1.$$

For larger values of $n$, there are two recursive invocations of `fib1`, taking time $T(n-1)$ and $T(n-2)$, respectively, plus three computer steps (checks on the value of $n$ and a final addition). Therefore,

$$T(n) = T(n-1) + T(n-2) + 3 \quad \text{for } n > 1.$$

Compare this to the recurrence relation for $F_n$: we immediately see that $T(n) \geq F_n$.

This is very bad news: the running time of the algorithm grows as fast as the Fibonacci numbers! $T(n)$ *is exponential in* $n$, which implies that the algorithm is impractically slow except for very small values of $n$.

Let's be a little more concrete about just how bad exponential time is. To compute $F_{200}$, the `fib1` algorithm executes $T(200) \geq F_{200} \geq 2^{138}$ elementary computer steps. How long this actually takes depends, of course, on the computer used. At this time, the fastest computer in the world is the NEC Earth Simulator, which clocks 40 trillion steps per second. Even on this machine, `fib1`$(200)$ would take at least $2^{92}$ seconds. This means that, if we start the computation today, it would still be going long after the sun turns into a red giant star.

But technology is rapidly improving—computer speeds have been doubling roughly every 18 months, a phenomenon sometimes called *Moore's law*. With this extraordinary growth, perhaps `fib1` will run a lot faster on next year's machines. Let's see—the running time of `fib1`$(n)$ is proportional to $2^{0.694n} \approx (1.6)^n$, so it takes 1.6 times longer to compute $F_{n+1}$ than $F_n$. And under Moore's law, computers get roughly 1.6 times faster each year. So if we can reasonably compute $F_{100}$ with this year's technology, then next year we will manage $F_{101}$. And the year after, $F_{102}$. And so on: just one more Fibonacci number every year! Such is the curse of exponential time.

In short, our naive recursive algorithm is correct but hopelessly inefficient. *Can we do better?*

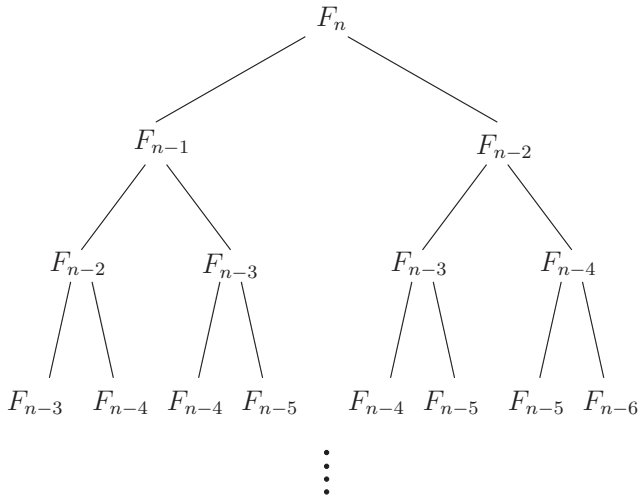
## A polynomial algorithm

Let's try to understand why `fib1` is so slow. Figure 0.1 shows the cascade of recursive invocations triggered by a single call to `fib1`$(n)$. Notice that many computations are repeated!

A more sensible scheme would store the intermediate results—the values $F_0, F_1, \ldots, F_{n-1}$—as soon as they become known.

```
function fib2(n)
if n = 0: return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
  f[i] = f[i-1] + f[i-2]
return f[n]
```

**Figure 0.1** The proliferation of recursive calls in `fib1`.




As with `fib1`, the correctness of this algorithm is self-evident because it directly uses the definition of $F_n$. How long does it take? The inner loop consists of a single computer step and is executed $n - 1$ times. Therefore the number of computer steps used by `fib2` is *linear in* $n$. From exponential we are down to *polynomial*, a huge breakthrough in running time. It is now perfectly reasonable to compute $F_{200}$ or even $F_{200,000}$.[1]

As we will see repeatedly throughout this book, the right algorithm makes all the difference.

## More careful analysis

In our discussion so far, we have been counting the number of *basic computer steps* executed by each algorithm and thinking of these basic steps as taking a constant amount of time. This is a very useful simplification. After all, a processor's instruction set has a variety of basic primitives—branching, storing to memory, comparing numbers, simple arithmetic, and so on—and rather than distinguishing between these elementary operations, it is far more convenient to lump them together into one category.

But looking back at our treatment of Fibonacci algorithms, we have been too liberal with what we consider a basic step. It is reasonable to treat addition as a single computer step if small numbers are being added, 32-bit numbers say. But the $n$th Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as $n$ grows.

[1] To better appreciate the importance of this dichotomy between exponential and polynomial algorithms, the reader may want to peek ahead to *the story of Sissa and Moore* in Chapter 8.

Arithmetic operations on arbitrarily large numbers cannot possibly be performed in a single, constant-time step. We need to audit our earlier running time estimates and make them more honest.

We will see in Chapter 1 that the addition of two $n$-bit numbers takes time roughly proportional to $n$; this is not too hard to understand if you think back to the grade-school procedure for addition, which works on one digit at a time. Thus `fib1`, which performs about $F_n$ additions, actually uses a number of *basic steps* roughly proportional to $nF_n$. Likewise, the number of steps taken by `fib2` is proportional to $n^2$, still polynomial in $n$ and therefore exponentially superior to `fib1`. This correction to the running time analysis does not diminish our breakthrough.

*But can we do even better than* `fib2`*?* Indeed we can: see Exercise 0.4.

## 0.3 Big-*O* notation

We've just seen how sloppiness in the analysis of running times can lead to an unacceptable level of inaccuracy in the result. But the opposite danger is also present: it is possible to be *too* precise. An insightful analysis is based on the right simplifications.

Expressing running time in terms of *basic computer steps* is already a simplification. After all, the time taken by one such step depends crucially on the particular processor and even on details such as caching strategy (as a result of which the running time can differ subtly from one execution to the next). Accounting for these architecture-specific minutiae is a nightmarishly complex task and yields a result that does not generalize from one computer to the next. It therefore makes more sense to seek an uncluttered, machine-independent characterization of an algorithm's efficiency. To this end, we will always express running time by counting the number of basic computer steps, as a function of the size of the input.

And this simplification leads to another. Instead of reporting that an algorithm takes, say, $5n^3 + 4n + 3$ steps on an input of size $n$, it is much simpler to leave out lower-order terms such as $4n$ and 3 (which become insignificant as $n$ grows), and even the detail of the coefficient 5 in the leading term (computers will be five times faster in a few years anyway), and just say that the algorithm takes time $O(n^3)$ (pronounced "big oh of $n^3$").

It is time to define this notation precisely. In what follows, think of $f(n)$ and $g(n)$ as the running times of two algorithms on inputs of size $n$.

> *Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say $f = O(g)$ (which means that "$f$ grows no faster than $g$") if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.*

Saying $f = O(g)$ is a very loose analog of "$f \leq g$." It differs from the usual notion of $\leq$ because of the constant $c$, so that for instance $10n = O(n)$. This constant also allows us to disregard what happens for small values of $n$. For example, suppose we

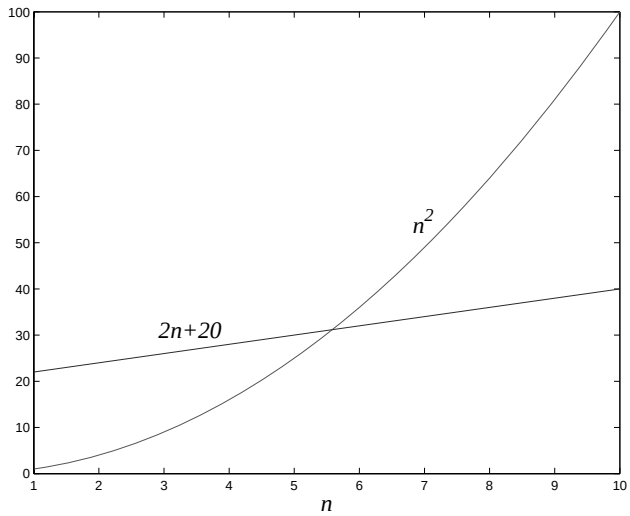
are choosing between two algorithms for a particular computational task. One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps (Figure 0.2). Which is better? Well, this depends on the value of $n$. For $n \leq 5$, $n^2$ is smaller; thereafter, $2n + 20$ is the clear winner. In this case, $f_2$ *scales* much better as $n$ grows, and therefore it is superior.

This superiority is captured by the big-$O$ notation: $f_2 = O(f_1)$, because

$$\frac{f_2(n)}{f_1(n)} = \frac{2n+20}{n^2} \leq 22$$

for all $n$; on the other hand, $f_1 \neq O(f_2)$, since the ratio $f_1(n)/f_2(n) = n^2/(2n+20)$ can get arbitrarily large, and so no constant $c$ will make the definition work.

**Figure 0.2** Which running time is better?




Now another algorithm comes along, one that uses $f_3(n) = n + 1$ steps. Is this better than $f_2$? Certainly, but only by a constant factor. The discrepancy between $2n + 20$ and $n + 1$ is tiny compared to the huge gap between $n^2$ and $2n + 20$. In order to stay focused on the big picture, we treat functions as equivalent if they differ only by multiplicative constants.

Returning to the definition of big-$O$, we see that $f_2 = O(f_3)$:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n+20}{n+1} \leq 20,$$

and of course $f_3 = O(f_2)$, this time with $c = 1$.

Just as $O(\cdot)$ is an analog of $\leq$, we can also define analogs of $\geq$ and $=$ as follows:

$$f = \Omega(g) \text{ means } g = O(f)$$
$$f = \Theta(g) \text{ means } f = O(g) \text{ and } f = \Omega(g).$$

In the preceding example, $f_2 = \Theta(f_3)$ and $f_1 = \Omega(f_3)$.

Big-$O$ notation lets us focus on the big picture. When faced with a complicated function like $3n^2 + 4n + 5$, we just replace it with $O(f(n))$, where $f(n)$ is as simple as possible. In this particular example we'd use $O(n^2)$, because the quadratic portion of the sum dominates the rest. Here are some commonsense rules that help simplify functions by omitting dominated terms:

1. Multiplicative constants can be omitted: $14n^2$ becomes $n^2$.
2. $n^a$ dominates $n^b$ if $a > b$: for instance, $n^2$ dominates $n$.
3. Any exponential dominates any polynomial: $3^n$ dominates $n^5$ (it even dominates $2^n$).
4. Likewise, any polynomial dominates any logarithm: $n$ dominates $(\log n)^3$. This also means, for example, that $n^2$ dominates $n \log n$.

Don't misunderstand this cavalier attitude toward constants. Programmers and algorithm developers are *very* interested in constants and would gladly stay up nights in order to make an algorithm run faster by a factor of 2. But understanding algorithms at the level of this book would be impossible without the simplicity afforded by big-$O$ notation.

## Exercises

0.1. In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case $f = \Theta(g)$).

| | $f(n)$ | $g(n)$ |
|---|---|---|
| (a) | $n - 100$ | $n - 200$ |
| (b) | $n^{1/2}$ | $n^{2/3}$ |
| (c) | $100n + \log n$ | $n + (\log n)^2$ |
| (d) | $n \log n$ | $10n \log 10n$ |
| (e) | $\log 2n$ | $\log 3n$ |
| (f) | $10 \log n$ | $\log(n^2)$ |
| (g) | $n^{1.01}$ | $n \log^2 n$ |
| (h) | $n^2 / \log n$ | $n(\log n)^2$ |
| (i) | $n^{0.1}$ | $(\log n)^{10}$ |
| (j) | $(\log n)^{\log n}$ | $n / \log n$ |
| (k) | $\sqrt{n}$ | $(\log n)^3$ |
| (l) | $n^{1/2}$ | $5^{\log_2 n}$ |
| (m) | $n2^n$ | $3^n$ |

| | | |
|---|---|---|
| (n) | $2^n$ | $2^{n+1}$ |
| (o) | $n!$ | $2^n$ |
| (p) | $(\log n)^{\log n}$ | $2^{(\log_2 n)^2}$ |
| (q) | $\sum_{i=1}^{n} i^k$ | $n^{k+1}$ |

0.2. Show that, if $c$ is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is:

(a) $\Theta(1)$ if $c < 1$.
(b) $\Theta(n)$ if $c = 1$.
(c) $\Theta(c^n)$ if $c > 1$.

The moral: in big-$\Theta$ terms, the sum of a geometric series is simply the first term if the series is strictly decreasing, the last term if the series is strictly increasing, or the number of terms if the series is unchanging.

0.3. The Fibonacci numbers $F_0, F_1, F_2, \ldots,$ are defined by the rule

$$F_0 = 0, \;\; F_1 = 1, \;\; F_n = F_{n-1} + F_{n-2}.$$

In this problem we will confirm that this sequence grows exponentially fast and obtain some bounds on its growth.

(a) Use induction to prove that $F_n \geq 2^{0.5n}$ for $n \geq 6$.
(b) Find a constant $c < 1$ such that $F_n \leq 2^{cn}$ for all $n \geq 0$. Show that your answer is correct.
(c) What is the largest $c$ you can find for which $F_n = \Omega(2^{cn})$?

0.4. Is there a faster way to compute the $n$th Fibonacci number than by `fib2` (page 4)? One idea involves *matrices.*
We start by writing the equations $F_1 = F_1$ and $F_2 = F_0 + F_1$ in matrix notation:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Similarly,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

and in general

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

So, in order to compute $F_n$, it suffices to raise this $2 \times 2$ matrix, call it $X$, to the $n$th power.

(a) Show that two $2 \times 2$ matrices can be multiplied using 4 additions and 8 multiplications.

But how many matrix multiplications does it take to compute $X^n$?

(b) Show that $O(\log n)$ matrix multiplications suffice for computing $X^n$. (*Hint:* Think about computing $X^8$.)

Thus the number of arithmetic operations needed by our matrix-based algorithm, call it `fib3`, is just $O(\log n)$, as compared to $O(n)$ for `fib2`. *Have we broken another exponential barrier?*

The catch is that our new algorithm involves multiplication, not just addition; and multiplications of large numbers are slower than additions. We have already seen that, when the complexity of arithmetic operations is taken into account, the running time of `fib2` becomes $O(n^2)$.

(c) Show that all intermediate results of `fib3` are $O(n)$ bits long.

(d) Let $M(n)$ be the running time of an algorithm for multiplying $n$-bit numbers, and assume that $M(n) = O(n^2)$ (the school method for multiplication, recalled in Chapter 1, achieves this). Prove that the running time of `fib3` is $O(M(n) \log n)$.

(e) Can you prove that the running time of `fib3` is $O(M(n))$? Assume $M(n) = \Theta(n^a)$ for some $1 \leq a \leq 2$. (*Hint:* The lengths of the numbers being multiplied get doubled with every squaring.)

In conclusion, whether `fib3` is faster than `fib2` depends on whether we can multiply $n$-bit integers faster than $O(n^2)$. Do you think this is possible? (The answer is in Chapter 2.)

Finally, there is a formula for the Fibonacci numbers:

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

So, it would appear that we only need to raise a couple of numbers to the $n$th power in order to compute $F_n$. The problem is that these numbers are irrational, and computing them to sufficient accuracy is nontrivial. In fact, our matrix method `fib3` can be seen as a roundabout way of raising these irrational numbers to the $n$th power. If you know your linear algebra, you should see why. (*Hint:* What are the eigenvalues of the matrix $X$?)

*Chapter 1*

# Algorithms with numbers

One of the main themes of this chapter is the dramatic contrast between two ancient problems that at first seem very similar:

> FACTORING: Given a number $N$, express it as a product of its prime factors.
>
> PRIMALITY: Given a number $N$, determine whether it is a prime.

Factoring is hard. Despite centuries of effort by some of the world's smartest mathematicians and computer scientists, the fastest methods for factoring a number $N$ take time exponential in the number of bits of $N$.

On the other hand, we shall soon see that we *can* efficiently test whether $N$ is prime! And (it gets even more interesting) this strange disparity between the two intimately related problems, one very hard and the other very easy, lies at the heart of the technology that enables secure communication in today's global information environment.

En route to these insights, we need to develop algorithms for a variety of computational tasks involving numbers. We begin with basic arithmetic, an especially appropriate starting point because, as we know, the word *algorithms* originally applied only to methods for these problems.

## 1.1 Basic arithmetic

### 1.1.1 Addition

We were so young when we learned the standard technique for addition that we would scarcely have thought to ask *why* it works. But let's go back now and take a closer look.

It is a basic property of decimal numbers that

> *The sum of any three single-digit numbers is at most two digits long.*

Quick check: the sum is at most $9 + 9 + 9 = 27$, two digits long. In fact, this rule holds not just in decimal but in *any* base $b \geq 2$ (Exercise 1.1). In binary, for instance, the maximum possible sum of three single-bit numbers is 3, which is a 2-bit number.

## Bases and logs

Naturally, there is nothing special about the number 10—we just happen to have 10 fingers, and so 10 was an obvious place to pause and take counting to the next level. The Mayans developed a similar positional system based on the number 20 (no shoes, see?). And of course today computers represent numbers in binary.

How many digits are needed to represent the number $N \geq 0$ in base $b$? Let's see—with $k$ digits in base $b$ we can express numbers up to $b^k - 1$; for instance, in decimal, three digits get us all the way up to $999 = 10^3 - 1$. By solving for $k$, we find that $\lceil \log_b(N+1) \rceil$ digits (about $\log_b N$ digits, give or take 1) are needed to write $N$ in base $b$.

How much does the size of a number change when we change bases? Recall the rule for converting logarithms from base $a$ to base $b$: $\log_b N = (\log_a N)/(\log_a b)$. So the size of integer $N$ in base $a$ is the same as its size in base $b$, times a constant factor $\log_a b$. In big-$O$ notation, therefore, the base is irrelevant, and we write the size simply as $O(\log N)$. When we do not specify a base, as we almost never will, we mean $\log_2 N$.

Incidentally, this function $\log N$ appears repeatedly in our subject, in many guises. Here's a sampling:

1. $\log N$ is, of course, the power to which you need to raise 2 in order to obtain $N$.
2. Going backward, it can also be seen as the number of times you must halve $N$ to get down to 1. (More precisely: $\lceil \log N \rceil$.) This is useful when a number is halved at each iteration of an algorithm, as in several examples later in the chapter.
3. It is the number of bits in the binary representation of $N$. (More precisely: $\lceil \log(N+1) \rceil$.)
4. It is also the depth of a complete binary tree with $N$ nodes. (More precisely: $\lfloor \log N \rfloor$.)
5. It is even the sum $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}$, to within a constant factor (Exercise 1.5).

This simple rule gives us a way to add two numbers in any base: align their right-hand ends, and then perform a single right-to-left pass in which the sum is computed digit by digit, maintaining the overflow as a carry. Since we know each individual sum is a two-digit number, *the carry is always a single digit*, and so at any given step, three single-digit numbers are added. Here's an example showing the addition $53 + 35$ in binary.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Carry: | 1 | | | 1 | 1 | 1 | | |
| | | 1 | 1 | 0 | 1 | 0 | 1 | (53) |
| | | 1 | 0 | 0 | 0 | 1 | 1 | (35) |
| | 1 | 0 | 1 | 1 | 0 | 0 | 0 | (88) |

Ordinarily we would spell out the algorithm in pseudocode, but in this case it is so familiar that we do not repeat it. Instead we move straight to analyzing its efficiency.

*Given two binary numbers $x$ and $y$, how long does our algorithm take to add them?* This is the kind of question we shall persistently be asking throughout this book. We want the answer expressed as a function of *the size of the input*: the number of bits of $x$ and $y$, the number of keystrokes needed to type them in.

Suppose $x$ and $y$ are each $n$ bits long; in this chapter we will consistently use the letter $n$ for the sizes of numbers. Then the sum of $x$ and $y$ is $n+1$ bits at most, and each individual bit of this sum gets computed in a fixed amount of time. The total running time for the addition algorithm is therefore of the form $c_0 + c_1 n$, where $c_0$ and $c_1$ are some constants; in other words, it is *linear*. Instead of worrying about the precise values of $c_0$ and $c_1$, we will focus on the big picture and denote the running time as $O(n)$.

Now that we have a working algorithm whose running time we know, our thoughts wander inevitably to the question of whether there is something even better.

*Is there a faster algorithm?* (This is another persistent question.) For addition, the answer is easy: in order to add two $n$-bit numbers we must at least read them and write down the answer, and even that requires $n$ operations. So the addition algorithm is optimal, up to multiplicative constants!

Some readers may be confused at this point: Why $O(n)$ operations? Isn't binary addition something that computers today perform by just one instruction? There are two answers. First, it is certainly true that in a single instruction we can add integers whose size in bits is within the *word length* of today's computers—32 perhaps. But, as will become apparent later in this chapter, it is often useful and necessary to handle numbers much larger than this, perhaps several thousand bits long. Adding and multiplying such large numbers on real computers is very much like performing the operations bit by bit. Second, when we want to understand algorithms, it makes sense to study even the basic algorithms that are encoded in the hardware of today's computers. In doing so, we shall focus on the *bit complexity* of the algorithm, the number of elementary operations on individual bits—because this accounting reflects the amount of hardware, transistors and wires, necessary for implementing the algorithm.

### 1.1.2 Multiplication and division

Onward to multiplication! The grade-school algorithm for multiplying two numbers $x$ and $y$ is to create an array of intermediate sums, each representing the product of $x$ by a single digit of $y$. These values are appropriately left-shifted and then added up. Suppose for instance that we want to multiply $13 \times 11$, or in binary notation, $x = 1101$ and $y = 1011$. The multiplication would proceed thus.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 1 | 0 | 1 | |
| | | | × | 1 | 0 | 1 | 1 | |
| | | | | 1 | 1 | 0 | 1 | (1101 times 1) |
| | | | 1 | 1 | 0 | 1 | | (1101 times 1, shifted once) |
| | | 0 | 0 | 0 | 0 | | | (1101 times 0, shifted twice) |
| + | 1 | 1 | 0 | 1 | | | | (1101 times 1, shifted thrice) |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | (binary 143) |

In binary this is particularly easy since each intermediate row is either zero or $x$ itself, left-shifted an appropriate amount of times. Also notice that left-shifting is just a quick way to multiply by the base, which in this case is 2. (Likewise, the effect of a right shift is to divide by the base, rounding down if needed.)

The correctness of this multiplication procedure is the subject of Exercise 1.6; let's move on and figure out how long it takes. If $x$ and $y$ are both $n$ bits, then there are $n$ intermediate rows, with lengths of up to $2n$ bits (taking the shifting into account). The total time taken to add up these rows, doing two numbers at a time, is

$$\underbrace{O(n) + O(n) + \cdots + O(n)}_{n-1 \text{ times}},$$

which is $O(n^2)$, *quadratic* in the size of the inputs: still polynomial but much slower than addition (as we have all suspected since elementary school).

But Al Khwarizmi knew another way to multiply, a method which is used today in some European countries. To multiply two decimal numbers $x$ and $y$, write them next to each other, as in the example below. Then repeat the following: divide the first number by 2, rounding down the result (that is, dropping the .5 if the number was odd), and double the second number. Keep going till the first number gets down to 1. Then strike out all the rows in which the first number is even, and add up whatever remains in the second column.

| | | |
|---|---|---|
| 11 | 13 | |
| 5 | 26 | |
| 2 | 52 | (strike out) |
| 1 | 104 | |
| | 143 | (answer) |

But if we now compare the two algorithms, binary multiplication and multiplication by repeated halvings of the multiplier, we notice that they are doing the same thing! The three numbers added in the second algorithm are precisely the multiples of 13 by powers of 2 that were added in the binary method. Only this time 11 was not given to us explicitly in binary, and so we had to extract its binary representation by looking at the parity of the numbers obtained from it by successive divisions by 2. Al Khwarizmi's second algorithm is a fascinating mixture of decimal and binary!

The same algorithm can thus be repackaged in different ways. For variety we adopt a third formulation, the recursive algorithm of Figure 1.1, which directly implements the rule

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is odd.} \end{cases}$$

*Is this algorithm correct?* The preceding recursive rule is transparently correct; so

**Figure 1.1** Multiplication à la Français.

```
function multiply(x, y)
Input: Two n–bit integers x and y, where y ≥ 0
Output: Their product

if y = 0: return 0
z = multiply(x, ⌊y/2⌋)
if y is even:
   return 2z
else:
   return x + 2z
```

checking the correctness of the algorithm is merely a matter of verifying that it mimics the rule and that it handles the base case ($y = 0$) properly.

*How long does the algorithm take?* It must terminate after $n$ recursive calls, because at each call $y$ is halved—that is, its number of bits is decreased by one. And each recursive call requires these operations: a division by 2 (right shift); a test for odd/even (looking up the last bit); a multiplication by 2 (left shift); and possibly one addition, a total of $O(n)$ bit operations. The total time taken is thus $O(n^2)$, just as before.

*Can we do better?* Intuitively, it seems that multiplication requires adding about $n$ multiples of one of the inputs, and we know that each addition is linear, so it would appear that $n^2$ bit operations are inevitable. Astonishingly, in Chapter 2 we'll see that we *can* do significantly better!

Division is next. To divide an integer $x$ by another integer $y \neq 0$ means to find a quotient $q$ and a remainder $r$, where $x = yq + r$ and $r < y$. We show the recursive version of division in Figure 1.2; like multiplication, it takes quadratic time. The analysis of this algorithm is the subject of Exercise 1.8.

**Figure 1.2** Division.

```
function divide(x, y)
Input: Two n–bit integers x and y, where y ≥ 1
Output: The quotient and remainder of x divided by y

if x = 0: return (q, r) = (0, 0)
(q, r) = divide(⌊x/2⌋, y)
q = 2 · q,  r = 2 · r
if x is odd: r = r + 1
if r ≥ y:  r = r − y,  q = q + 1
return (q, r)
```

## 1.2 Modular arithmetic

With repeated addition or multiplication, numbers can get cumbersomely large. So it is fortunate that we reset the hour to zero whenever it reaches 24, and the month to January after every stretch of 12 months. Similarly, for the built-in arithmetic operations of computer processors, numbers are restricted to some size, 32 bits say, which is considered generous enough for most purposes.

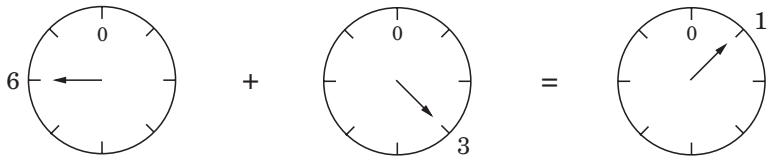
For the applications we are working toward—primality testing and cryptography—it is necessary to deal with numbers that are significantly larger than 32 bits, but whose range is nonetheless limited.

*Modular arithmetic* is a system for dealing with restricted ranges of integers. We define *x modulo N* to be the remainder when $x$ is divided by $N$; that is, if $x = qN + r$ with $0 \leq r < N$, then $x$ modulo $N$ is equal to $r$. This gives an enhanced notion of equivalence between numbers: $x$ and $y$ are *congruent modulo N* if they differ by a multiple of $N$, or in symbols,

$$x \equiv y \pmod{N} \iff N \text{ divides } (x - y).$$

For instance, $253 \equiv 13 \pmod{60}$ because $253 - 13$ is a multiple of 60; more familiarly, 253 minutes is 4 hours and 13 minutes. These numbers can also be negative, as in $59 \equiv -1 \pmod{60}$: when it is 59 minutes past the hour, it is also 1 minute short of the next hour.

**Figure 1.3** Addition modulo 8.




One way to think of modular arithmetic is that it limits numbers to a predefined range $\{0, 1, \ldots, N-1\}$ and wraps around whenever you try to leave this range—like the hand of a clock (Figure 1.3).

Another interpretation is that modular arithmetic deals with all the integers, but divides them into $N$ *equivalence classes*, each of the form $\{i + kN : k \in \mathbb{Z}\}$ for some $i$ between 0 and $N - 1$. For example, there are three equivalence classes modulo 3:

$$\begin{array}{cccccccccc} \cdots & -9 & -6 & -3 & 0 & 3 & 6 & 9 & \cdots \\ \cdots & -8 & -5 & -2 & 1 & 4 & 7 & 10 & \cdots \\ \cdots & -7 & -4 & -1 & 2 & 5 & 8 & 11 & \cdots \end{array}$$

Any member of an equivalence class is substitutable for any other; when viewed modulo 3, the numbers 5 and 11 are no different. Under such substitutions, addition and multiplication remain well-defined:

### Two's complement

Modular arithmetic is nicely illustrated in *two's complement*, the most common format for storing signed integers. It uses $n$ bits to represent numbers in the range $[-2^{n-1}, 2^{n-1} - 1]$ and is usually described as follows:

- Positive integers, in the range 0 to $2^{n-1} - 1$, are stored in regular binary and have a leading bit of 0.
- Negative integers $-x$, with $1 \leq x \leq 2^{n-1}$, are stored by first constructing $x$ in binary, then flipping all the bits, and finally adding 1. The leading bit in this case is 1.

(And the usual description of addition and multiplication in this format is even more arcane!)

Here's a much simpler way to think about it: any number in the range $-2^{n-1}$ to $2^{n-1} - 1$ is stored modulo $2^n$. Negative numbers $-x$ therefore end up as $2^n - x$. Arithmetic operations like addition and subtraction can be performed directly in this format, ignoring any overflow bits that arise.

**Substitution rule** *If* $x \equiv x' \pmod N$ *and* $y \equiv y' \pmod N$, *then:*

$$x + y \equiv x' + y' \pmod N \quad \textit{and} \quad xy \equiv x'y' \pmod N.$$

(See Exercise 1.9.) For instance, suppose you watch an entire season of your favorite television show in one sitting, starting at midnight. There are 25 episodes, each lasting 3 hours. At what time of day are you done? Answer: the hour of completion is $(25 \times 3) \bmod 24$, which (since $25 \equiv 1 \bmod 24$) is $1 \times 3 = 3 \bmod 24$, or three o'clock in the morning.

It is not hard to check that in modular arithmetic, the usual associative, commutative, and distributive properties of addition and multiplication continue to apply, for instance:

| | |
|---|---|
| $x + (y + z) \equiv (x + y) + z \pmod N$ | Associativity |
| $xy \equiv yx \pmod N$ | Commutativity |
| $x(y + z) \equiv xy + yz \pmod N$ | Distributivity |

Taken together with the substitution rule, this implies that while performing a sequence of arithmetic operations, it is legal to reduce intermediate results to their remainders modulo $N$ at any stage. Such simplifications can be a dramatic help in big calculations. Witness, for instance:

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}.$$

## 1.2.1 Modular addition and multiplication

To *add* two numbers $x$ and $y$ modulo $N$, we start with regular addition. Since $x$ and $y$ are each in the range 0 to $N - 1$, their sum is between 0 and $2(N - 1)$. If the sum

exceeds $N-1$, we merely need to subtract off $N$ to bring it back into the required range. The overall computation therefore consists of an addition, and possibly a subtraction, of numbers that never exceed $2N$. Its running time is linear in the sizes of these numbers, in other words $O(n)$, where $n = \lceil \log N \rceil$ is the size of $N$; as a reminder, our convention is to use the letter $n$ to denote input size.

To *multiply* two mod-$N$ numbers $x$ and $y$, we again just start with regular multiplication and then reduce the answer modulo $N$. The product can be as large as $(N-1)^2$, but this is still at most $2n$ bits long since $\log(N-1)^2 = 2\log(N-1) \leq 2n$. To reduce the answer modulo $N$, we compute the remainder upon dividing it by $N$, using our quadratic-time division algorithm. Multiplication thus remains a quadratic operation.

*Division* is not quite so easy. In ordinary arithmetic there is just one tricky case—division by zero. It turns out that in modular arithmetic there are potentially other such cases as well, which we will characterize toward the end of this section. Whenever division is legal, however, it can be managed in cubic time, $O(n^3)$.

To complete the suite of modular arithmetic primitives we need for cryptography, we next turn to *modular exponentiation*, and then to the *greatest common divisor*, which is the key to division. For both tasks, the most obvious procedures take exponentially long, but with some ingenuity polynomial-time solutions can be found. A careful choice of algorithm makes all the difference.

### 1.2.2 Modular exponentiation

In the cryptosystem we are working toward, it is necessary to compute $x^y \bmod N$ for values of $x$, $y$, and $N$ that are several hundred bits long. Can this be done quickly?

The result is some number modulo $N$ and is therefore itself a few hundred bits long. However, the raw value of $x^y$ could be much, much longer than this. Even when $x$ and $y$ are just 20-bit numbers, $x^y$ is at least $(2^{19})^{(2^{19})} = 2^{(19)(524288)}$, about 10 million bits long! Imagine what happens if $y$ is a 500-bit number!

To make sure the numbers we are dealing with never grow too large, we need to perform all intermediate computations modulo $N$. So here's an idea: calculate $x^y \bmod N$ by repeatedly multiplying by $x$ modulo $N$. The resulting sequence of intermediate products,

$$x \bmod N \to x^2 \bmod N \to x^3 \bmod N \to \cdots \to x^y \bmod N,$$

consists of numbers that are smaller than $N$, and so the individual multiplications do not take too long. But there's a problem: if $y$ is 500 bits long, we need to perform $y - 1 \approx 2^{500}$ multiplications! This algorithm is clearly exponential in the size of $y$.

Luckily, we *can* do better: starting with $x$ and *squaring repeatedly* modulo $N$, we get

$$x \bmod N \to x^2 \bmod N \to x^4 \bmod N \to x^8 \bmod N \to \cdots \to x^{2^{\lfloor \log y \rfloor}} \bmod N.$$

Each takes just $O(\log^2 N)$ time to compute, and in this case there are only $\log y$ multiplications. To determine $x^y \bmod N$, we simply multiply together an appropriate subset of these powers, those corresponding to 1's in the binary representation of $y$. For instance,

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^8 \cdot x^1.$$

A polynomial-time algorithm is finally within reach!

**Figure 1.4** Modular exponentiation.

```
function modexp(x, y, N)
Input: Two n–bit integers x and N, an integer exponent y
Output: x^y mod N

if y = 0: return 1
z = modexp(x, ⌊y/2⌋, N)
if y is even:
   return z^2 mod N
else:
   return x · z^2 mod N
```

We can package this idea in a particularly simple form: the recursive algorithm of Figure 1.4, which works by executing, modulo $N$, the self-evident rule

$$x^y = \begin{cases} \left(x^{\lfloor y/2 \rfloor}\right)^2 & \text{if } y \text{ is even} \\ x \cdot \left(x^{\lfloor y/2 \rfloor}\right)^2 & \text{if } y \text{ is odd.} \end{cases}$$

In doing so, it closely parallels our recursive multiplication algorithm (Figure 1.1). For instance, that algorithm would compute the product $x \cdot 25$ by an analogous decomposition to the one we just saw: $x \cdot 25 = x \cdot 16 + x \cdot 8 + x \cdot 1$. And whereas for multiplication the terms $x \cdot 2^i$ come from repeated *doubling*, for exponentiation the corresponding terms $x^{2^i}$ are generated by repeated squaring.

Let $n$ be the size in bits of $x$, $y$, and $N$ (whichever is largest of the three). As with multiplication, the algorithm will halt after at most $n$ recursive calls, and during each call it multiplies $n$-bit numbers (doing computation modulo $N$ saves us here), for a total running time of $O(n^3)$.

### 1.2.3 Euclid's algorithm for greatest common divisor

Our next algorithm was discovered well over 2000 years ago by the mathematician Euclid, in ancient Greece. Given two integers $a$ and $b$, it finds the largest integer that divides both of them, known as their *greatest common divisor* (gcd).

The most obvious approach is to first factor $a$ and $b$, and then multiply together their common factors. For instance, $1035 = 3^2 \cdot 5 \cdot 23$ and $759 = 3 \cdot 11 \cdot 23$, so their

gcd is $3 \cdot 23 = 69$. However, we have no efficient algorithm for factoring. Is there some other way to compute greatest common divisors?

Euclid's algorithm uses the following simple formula.

Euclid of Alexandria
BC 325–265



**Euclid's rule** *If $x$ and $y$ are positive integers with $x \geq y$, then* $\gcd(x, y) = \gcd(x \bmod y, y)$.

*Proof.* It is enough to show the slightly simpler rule $\gcd(x, y) = \gcd(x - y, y)$ from which the one stated can be derived by repeatedly subtracting $y$ from $x$.

Here it goes. Any integer that divides both $x$ and $y$ must also divide $x - y$, so $\gcd(x, y) \leq \gcd(x - y, y)$. Likewise, any integer that divides both $x - y$ and $y$ must also divide both $x$ and $y$, so $\gcd(x, y) \geq \gcd(x - y, y)$. ∎

**Figure 1.5** Euclid's algorithm for finding the greatest common divisor of two numbers.

```
function Euclid(a, b)
Input: Two integers a and b with a ≥ b ≥ 0
Output: gcd(a, b)

if b = 0: return a
return Euclid(b, a mod b)
```

Euclid's rule allows us to write down an elegant recursive algorithm (Figure 1.5), and its correctness follows immediately from the rule. In order to figure out its running time, we need to understand how quickly the arguments $(a, b)$ decrease with each successive recursive call. In a single round, arguments $(a, b)$ become $(b, a \bmod b)$: their order is swapped, and the larger of them, $a$, gets reduced to $a \bmod b$. This is a substantial reduction.

**Lemma** *If $a \geq b$, then $a \bmod b < a/2$.*

*Proof.* Witness that either $b \leq a/2$ or $b > a/2$. These two cases are shown in the following figure. If $b \leq a/2$, then we have $a \bmod b < b \leq a/2$; and if $b > a/2$, then $a \bmod b = a - b < a/2$. ∎


This means that after any two consecutive rounds, both arguments, $a$ and $b$, are at the very least halved in value—the length of each decreases by at least one bit. If they are initially $n$-bit integers, then the base case will be reached within $2n$ recursive calls. And since each call involves a quadratic-time division, the total time is $O(n^3)$.

### 1.2.4 An extension of Euclid's algorithm

A small extension to Euclid's algorithm is the key to dividing in the modular world.

To motivate it, suppose someone claims that $d$ is the greatest common divisor of $a$ and $b$: how can we check this? It is not enough to verify that $d$ divides both $a$ and $b$, because this only shows $d$ to be a common factor, not necessarily the largest one. Here's a test that can be used if $d$ is of a particular form.

**Lemma** *If $d$ divides both $a$ and $b$, and $d = ax + by$ for some integers $x$ and $y$, then necessarily $d = \gcd(a, b)$.*

*Proof.* By the first two conditions, $d$ is a common divisor of $a$ and $b$ and so it cannot exceed the greatest common divisor; that is, $d \leq \gcd(a, b)$. On the other hand, since $\gcd(a, b)$ is a common divisor of $a$ and $b$, it must also divide $ax + by = d$, which implies $\gcd(a, b) \leq d$. Putting these together, $d = \gcd(a, b)$. ∎

So, if we can supply two numbers $x$ and $y$ such that $d = ax + by$, then we can be sure $d = \gcd(a, b)$. For instance, we know $\gcd(13, 4) = 1$ because $13 \cdot 1 + 4 \cdot (-3) = 1$. But when can we find these numbers: under what circumstances can $\gcd(a, b)$ be expressed in this checkable form? It turns out that it *always* can. What is even better, the coefficients $x$ and $y$ can be found by a small extension to Euclid's algorithm; see Figure 1.6.

**Figure 1.6** A simple extension of Euclid's algorithm.

```
function extended-Euclid(a, b)
Input: Two positive integers a and b with a ≥ b ≥ 0
Output: Integers x, y, d such that d = gcd(a, b) and ax + by = d

if b = 0: return (1, 0, a)
(x', y', d) = extended-Euclid(b, a mod b)
return (y', x' − ⌊a/b⌋y', d)
```

**Lemma** *For any positive integers $a$ and $b$, the extended Euclid algorithm returns integers $x$, $y$, and $d$ such that* $\gcd(a, b) = d = ax + by$.

*Proof.* The first thing to confirm is that if you ignore the $x$'s and $y$'s, the extended algorithm is exactly the same as the original. So, at least we compute $d = \gcd(a, b)$.

For the rest, the recursive nature of the algorithm suggests a proof by induction. The recursion ends when $b = 0$, so it is convenient to do induction on the value of $b$.

The base case $b = 0$ is easy enough to check directly. Now pick any larger value of $b$. The algorithm finds $\gcd(a, b)$ by calling $\gcd(b, a \bmod b)$. Since $a \bmod b < b$, we can apply the inductive hypothesis to this recursive call and conclude that the $x'$ and $y'$ it returns are correct:

$$\gcd(b, a \bmod b) = bx' + (a \bmod b)y'.$$

Writing $(a \bmod b)$ as $(a - \lfloor a/b \rfloor b)$, we find

$$\begin{aligned} d &= \gcd(a, b) = \gcd(b, a \bmod b) = bx' + (a \bmod b)y' \\ &= bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y'). \end{aligned}$$

Therefore $d = ax + by$ with $x = y'$ and $y = x' - \lfloor a/b \rfloor y'$, thus validating the algorithm's behavior on input $(a, b)$. ∎

*Example.* To compute $\gcd(25, 11)$, Euclid's algorithm would proceed as follows:

$$\begin{aligned} \underline{25} &= 2 \cdot \underline{11} + 3 \\ \underline{11} &= 3 \cdot \underline{3} + 2 \\ \underline{3} &= 1 \cdot \underline{2} + 1 \\ \underline{2} &= 2 \cdot \underline{1} + 0 \end{aligned}$$

(at each stage, the gcd computation has been reduced to the underlined numbers). Thus $\gcd(25, 11) = \gcd(11, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1$.

To find $x$ and $y$ such that $25x + 11y = 1$, we start by expressing 1 in terms of the last pair $(1, 0)$. Then we work backwards and express it in terms of $(2, 1)$, $(3, 2)$, $(11, 3)$, and finally $(25, 11)$. The first step is:

$$1 = \underline{1} - \underline{0}.$$

To rewrite this in terms of $(2, 1)$, we use the substitution $0 = 2 - 2 \cdot 1$ from the last line of the gcd calculation to get:

$$1 = \underline{1} - (\underline{2} - 2 \cdot \underline{1}) = -1 \cdot \underline{2} + 3 \cdot \underline{1}.$$

The second-last line of the gcd calculation tells us that $1 = 3 - 1 \cdot 2$. Substituting:

$$1 = -1 \cdot \underline{2} + 3(\underline{3} - 1 \cdot \underline{2}) = 3 \cdot \underline{3} - 4 \cdot \underline{2}.$$

Continuing in this same way with substitutions $2 = 11 - 3 \cdot 3$ and $3 = 25 - 2 \cdot 11$ gives:

$$1 = 3 \cdot \underline{3} - 4(\underline{11} - 3 \cdot \underline{3}) = -4 \cdot \underline{11} + 15 \cdot \underline{3} = -4 \cdot \underline{11} + 15(\underline{25} - 2 \cdot \underline{11}) = 15 \cdot \underline{25} - 34 \cdot \underline{11}.$$

We're done: $15 \cdot 25 - 34 \cdot 11 = 1$, so $x = 15$ and $y = -34$.

### 1.2.5 Modular division

In real arithmetic, every number $a \neq 0$ has an inverse, $1/a$, and dividing by $a$ is the same as multiplying by this inverse. In modular arithmetic, we can make a similar definition.

*We say $x$ is the multiplicative inverse of $a$ modulo $N$ if $ax \equiv 1 \pmod{N}$.*

There can be at most one such $x$ modulo $N$ (Exercise 1.23), and we shall denote it by $a^{-1}$. However, this inverse does not always exist! For instance, 2 is not invertible modulo 6: that is, $2x \not\equiv 1 \bmod 6$ for every possible choice of $x$. In this case, $a$ and $N$ are both even and thus then $a \bmod N$ is always even, since $a \bmod N = a - kN$ for some $k$. More generally, we can be certain that $\gcd(a, N)$ divides $ax \bmod N$, because this latter quantity can be written in the form $ax + kN$. So if $\gcd(a, N) > 1$, then $ax \not\equiv 1 \bmod N$, no matter what $x$ might be, and therefore $a$ cannot have a multiplicative inverse modulo $N$.

In fact, this is the only circumstance in which $a$ is not invertible. When $\gcd(a, N) = 1$ (we say $a$ and $N$ are *relatively prime*), the extended Euclid algorithm gives us integers $x$ and $y$ such that $ax + Ny = 1$, which means that $ax \equiv 1 \pmod{N}$. Thus $x$ is $a$'s sought inverse.

*Example.* Continuing with our previous example, suppose we wish to compute $11^{-1} \bmod 25$. Using the extended Euclid algorithm, we find that $15 \cdot 25 - 34 \cdot 11 = 1$. Reducing both sides modulo 25, we have $-34 \cdot 11 \equiv 1 \bmod 25$. So $-34 \equiv 16 \bmod 25$ is the inverse of 11 mod 25.

**Modular division theorem** *For any $a \bmod N$, $a$ has a multiplicative inverse modulo $N$ if and only if it is relatively prime to $N$. When this inverse exists, it can be found in time $O(n^3)$ (where as usual $n$ denotes the number of bits of $N$) by running the extended Euclid algorithm.*

This resolves the issue of modular division: when working modulo $N$, we can divide by numbers relatively prime to $N$—and only by these. And to actually carry out the division, we multiply by the inverse.

## 1.3 Primality testing

Is there some litmus test that will tell us whether a number is prime without actually trying to factor the number? We place our hopes in a theorem from the year 1640.

**Fermat's little theorem** *If $p$ is prime, then for every $1 \leq a < p$,*

$$a^{p-1} \equiv 1 \pmod{p}.$$

*Proof.* Let $S$ be the nonzero integers modulo $p$; that is, $S = \{1, 2, \ldots, p-1\}$. Here's the crucial observation: the effect of multiplying these numbers by $a$ (modulo $p$) is simply to permute them. For instance, here's a picture of the case $a = 3$, $p = 7$:

## Is your social security number a prime?

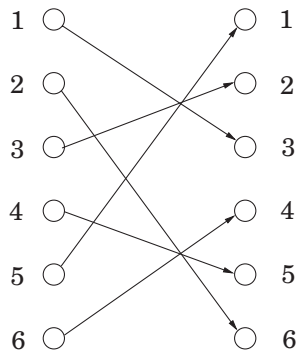
The numbers 7, 17, 19, 71, and 79 are primes, but how about 717-19-7179? Telling whether a reasonably large number is a prime seems tedious because there are far too many candidate factors to try. However, there are some clever tricks to speed up the process. For instance, you can omit even-valued candidates after you have eliminated the number 2. You can actually omit all candidates except those that are themselves primes.

In fact, a little further thought will convince you that you can proclaim $N$ a prime as soon as you have rejected all candidates *up to* $\sqrt{N}$, for if $N$ can indeed be factored as $N = K \cdot L$, then it is impossible for both factors to exceed $\sqrt{N}$.

We seem to be making progress! Perhaps by omitting more and more candidate factors, a truly efficient primality test can be discovered.

Unfortunately, there is no fast primality test down this road. The reason is that we have been trying to tell if a number is a prime *by factoring it.* And factoring is a hard problem!

Modern cryptography, as well as the balance of this chapter, is about the following important idea: *factoring is hard and primality is easy.* We cannot factor large numbers, but we can easily test huge numbers for primality! (Presumably, if a number is composite, such a test will detect this *without finding a factor.*)




Let's carry this example a bit further. From the picture, we can conclude

$$\{1, 2, \ldots, 6\} \;=\; \{3 \cdot 1 \bmod 7,\; 3 \cdot 2 \bmod 7, \ldots,\; 3 \cdot 6 \bmod 7\}.$$

Multiplying all the numbers in each representation then gives $6! \equiv 3^6 \cdot 6! \pmod 7$, and dividing by 6! we get $3^6 \equiv 1 \pmod 7$, exactly the result we wanted in the case $a = 3$, $p = 7$.

Now let's generalize this argument to other values of $a$ and $p$, with $S = \{1, 2, \ldots, p-1\}$. We'll prove that when the elements of $S$ are multiplied by $a$ modulo $p$, the resulting numbers are all distinct and nonzero. And since they lie in the range $[1, p-1]$, they must simply be a permutation of $S$.

The numbers $a \cdot i \bmod p$ are distinct because if $a \cdot i \equiv a \cdot j \pmod{p}$, then dividing both sides by $a$ gives $i \equiv j \pmod{p}$. They are nonzero because $a \cdot i \equiv 0$ similarly implies $i \equiv 0$. (And we *can* divide by $a$, because by assumption it is nonzero and therefore relatively prime to $p$.)

We now have two ways to write set $S$:

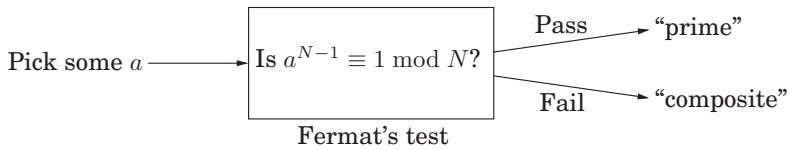
$$S = \{1, 2, \ldots, p-1\} = \{a \cdot 1 \bmod p,\ a \cdot 2 \bmod p, \ldots,\ a \cdot (p-1) \bmod p\}.$$

We can multiply together its elements in each of these representations to get

$$(p-1)! \equiv a^{p-1} \cdot (p-1)! \pmod{p}.$$

Dividing by $(p-1)!$ (which we can do because it is relatively prime to $p$, since $p$ is assumed prime) then gives the theorem. ∎

This theorem suggests a "factorless" test for determining whether a number $N$ is prime:




The problem is that Fermat's theorem is not an if-and-only-if condition; it doesn't say what happens when $N$ is *not* prime, so in these cases the preceding diagram is questionable. In fact, it *is* possible for a composite number $N$ to pass Fermat's test (that is, $a^{N-1} \equiv 1 \bmod N$) for certain choices of $a$. For instance, $341 = 11 \cdot 31$ is not prime, and yet $2^{340} \equiv 1 \bmod 341$. Nonetheless, we might hope that for composite $N$, *most* values of $a$ will fail the test. This is indeed true, in a sense we will shortly make precise, and motivates the algorithm of Figure 1.7: rather than fixing an arbitrary value of $a$ in advance, we should choose it *randomly* from $\{1, \ldots, N-1\}$.

**Figure 1.7** An algorithm for testing primality.

```
function primality(N)
Input: Positive integer N
Output: yes/no

Pick a positive integer a < N at random
if a^(N-1) ≡ 1 (mod N):
   return yes
else:
   return no
```

In analyzing the behavior of this algorithm, we first need to get a minor bad case out of the way. It turns out that certain extremely rare composite numbers $N$, called *Carmichael numbers*, pass Fermat's test for *all* $a$ relatively prime to $N$. On such numbers our algorithm will fail; but they are pathologically rare, and we will later see how to deal with them (page 28), so let's ignore these numbers for the time being.

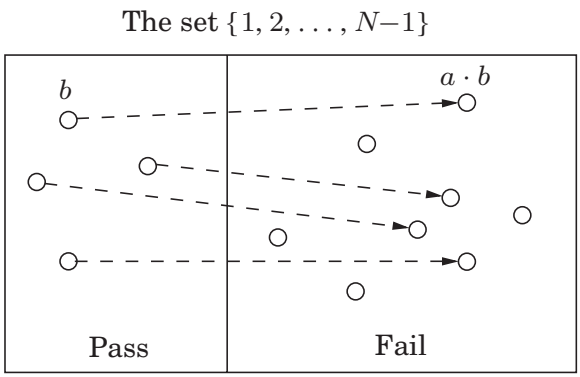
In a Carmichael-free universe, our algorithm works well. Any prime number $N$ will of course pass Fermat's test and produce the right answer. On the other hand, any non-Carmichael composite number $N$ must fail Fermat's test for some value of $a$; and as we will now show, this implies immediately that $N$ fails Fermat's test for *at least half the possible values of* $a$!

**Lemma** *If $a^{N-1} \not\equiv 1 \bmod N$ for some $a$ relatively prime to $N$, then it must hold for at least half the choices of $a < N$.*

*Proof.* Fix some value of $a$ for which $a^{N-1} \not\equiv 1 \bmod N$. The key is to notice that every element $b < N$ that passes Fermat's test with respect to $N$ (that is, $b^{N-1} \equiv 1 \bmod N$) has a twin, $a \cdot b$, that fails the test:

$$(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv a^{N-1} \not\equiv 1 \bmod N.$$

Moreover, all these elements $a \cdot b$, for fixed $a$ but different choices of $b$, are distinct, for the same reason $a \cdot i \not\equiv a \cdot j$ in the proof of Fermat's test: just divide by $a$.




The one-to-one function $b \mapsto a \cdot b$ shows that at least as many elements fail the test as pass it. ∎

We are ignoring Carmichael numbers, so we can now assert

If $N$ is prime, then $a^{N-1} \equiv 1 \bmod N$ for all $a < N$.
If $N$ is not prime, then $a^{N-1} \equiv 1 \bmod N$ for at most half the values of $a < N$.

The algorithm of Figure 1.7 therefore has the following probabilistic behavior.

$$\Pr(\text{Algorithm 1.7 returns } \texttt{yes} \text{ when } N \text{ is prime}) = 1$$
$$\Pr(\text{Algorithm 1.7 returns } \texttt{yes} \text{ when } N \text{ is not prime}) \leq \frac{1}{2}$$

### Hey, that was group theory!

For any integer $N$, the set of all numbers mod $N$ that are relatively prime to $N$ constitute what mathematicians call a *group*:

- There is a multiplication operation defined on this set.
- The set contains a neutral element (namely 1: any number multiplied by this remains unchanged).
- All elements have a well-defined inverse.

This particular group is called the *multiplicative group of* $N$, usually denoted $\mathbb{Z}_N^*$.

Group theory is a very well developed branch of mathematics. One of its key concepts is that a group can contain a *subgroup*—a subset that is a group in and of itself. And an important fact about a subgroup is that its size must divide the size of the whole group.

Consider now the set $B = \{b : b^{N-1} \equiv 1 \bmod N\}$. It is not hard to see that it is a subgroup of $\mathbb{Z}_N^*$ (just check that $B$ is closed under multiplication and inverses). Thus the size of $B$ must divide that of $\mathbb{Z}_N^*$. Which means that if $B$ doesn't contain all of $\mathbb{Z}_N^*$, the next largest size it can have is $|\mathbb{Z}_N^*|/2$.

We can reduce this *one-sided error* by repeating the procedure many times, by randomly picking several values of $a$ and testing them all (Figure 1.8).

$$\Pr(\text{Algorithm 1.8 returns } \texttt{yes} \text{ when } N \text{ is not prime}) \leq \frac{1}{2^k}$$

This probability of error drops exponentially fast, and can be driven arbitrarily low by choosing $k$ large enough. Testing $k = 100$ values of $a$ makes the probability of failure at most $2^{-100}$, which is miniscule: far less, for instance, than the probability that a random cosmic ray will sabotage the computer during the computation!

**Figure 1.8** An algorithm for testing primality, with low error probability.

```
function primality2(N)
Input: Positive integer N
Output: yes/no

Pick positive integers a_1, a_2, ..., a_k < N at random
if a_i^{N-1} ≡ 1 (mod N) for all i = 1, 2, ..., k:
   return yes
else:
   return no
```

### Carmichael numbers

The smallest Carmichael number is 561. It is not a prime: $561 = 3 \cdot 11 \cdot 17$; yet it fools the Fermat test, because $a^{560} \equiv 1 \pmod{561}$ for all values of $a$ relatively prime to 561. For a long time it was thought that there might be only finitely many numbers of this type; now we know they are infinite, but exceedingly rare.

There *is* a way around Carmichael numbers, using a slightly more refined primality test due to Rabin and Miller. Write $N-1$ in the form $2^t u$. As before we'll choose a random base $a$ and check the value of $a^{N-1} \bmod N$. Perform this computation by first determining $a^u \bmod N$ and then repeatedly squaring, to get the sequence:

$$a^u \bmod N,\ a^{2u} \bmod N,\ \ldots,\ a^{2^t u} = a^{N-1} \bmod N.$$

If $a^{N-1} \not\equiv 1 \bmod N$, then $N$ is composite by Fermat's little theorem, and we're done. But if $a^{N-1} \equiv 1 \bmod N$, we conduct a little follow-up test: somewhere in the preceding sequence, we ran into a 1 for the first time. If this happened after the first position (that is, if $a^u \bmod N \neq 1$), and if the preceding value in the list is not $-1 \bmod N$, then we declare $N$ composite.

In the latter case, we have found a *nontrivial square root* of 1 modulo $N$: a number that is not $\pm 1 \bmod N$ but that when squared is equal to $1 \bmod N$. Such a number can only exist if $N$ is composite (Exercise 1.40). It turns out that if we combine this square-root check with our earlier Fermat test, then at least three-fourths of the possible values of $a$ between 1 and $N-1$ will reveal a composite $N$, even if it is a Carmichael number.

### 1.3.1 Generating random primes

We are now close to having all the tools we need for cryptographic applications. The final piece of the puzzle is a fast algorithm for choosing random primes that are a few hundred bits long. What makes this task quite easy is that primes are abundant—a random $n$-bit number has roughly a one-in-$n$ chance of being prime (actually about $1/(\ln 2^n) \approx 1.44/n$). For instance, *about 1 in 20 social security numbers is prime!*

**Lagrange's prime number theorem** *Let $\pi(x)$ be the number of primes $\leq x$. Then $\pi(x) \approx x/(\ln x)$, or more precisely,*

$$\lim_{x \to \infty} \frac{\pi(x)}{(x/\ln x)} = 1.$$

Such abundance makes it simple to generate a random $n$-bit prime:

- Pick a random $n$-bit number $N$.
- Run a primality test on $N$.
- If it passes the test, output $N$; else repeat the process.

How fast is this algorithm? If the randomly chosen $N$ is truly prime, which happens with probability at least $1/n$, then it will certainly pass the test. So on each iteration,

## Randomized algorithms: a virtual chapter

Surprisingly—almost paradoxically—some of the fastest and most clever algorithms we have rely on *chance*: at specified steps they proceed according to the outcomes of random coin tosses. These *randomized algorithms* are often very simple and elegant, and their output is allowed to be incorrect *with small probability*. This bound on the failure probability holds for every input; it only depends on the random choices made by the algorithm itself, and can easily be made as small as one likes.

Instead of devoting a special chapter to this topic, in this book we intersperse randomized algorithms at the chapters and sections where they arise most naturally. Furthermore, no specialized knowledge of probability is necessary to follow what is happening. You just need to be familiar with the concept of probability, expected value, the expected number of times we must flip a coin before getting heads, and the property known as "linearity of expectation."

Here are pointers to the major randomized algorithms in this book: One of the earliest and most dramatic examples of a randomized algorithm is the probabilistic primality test of Figure 1.8. Although a deterministic primality test was recently discovered, the randomized test is much faster and therefore remains the algorithm of choice. Later in this chapter, in Section 1.5 (page 35), we discuss hashing, a general randomized data structure that supports inserts, deletes, and lookups. Again, in practice it leads to faster data access than deterministic schemes like binary search trees.

There are two varieties of randomized algorithms. *Monte Carlo* algorithms always run fast but their output has a small chance of being incorrect; the primality test is an example. *Las Vegas* algorithms, on the other hand, always output the correct answer but guarantee a short running time with high probability. Examples of this are the randomized algorithms for sorting and median finding described in Chapter 2 (on pages 50 and 53, respectively).

The fastest known algorithm for the minimum cut problem is a randomized Monte Carlo algorithm, described in the box on page 139. Randomization plays an important role in heuristics as well; these are described in Section 9.3. And finally the quantum algorithm for factoring (Section 10.7) works very much like a randomized algorithm, its output being correct with high probability—except that it draws its randomness not from coin tosses, but from the superposition principle in quantum mechanics.

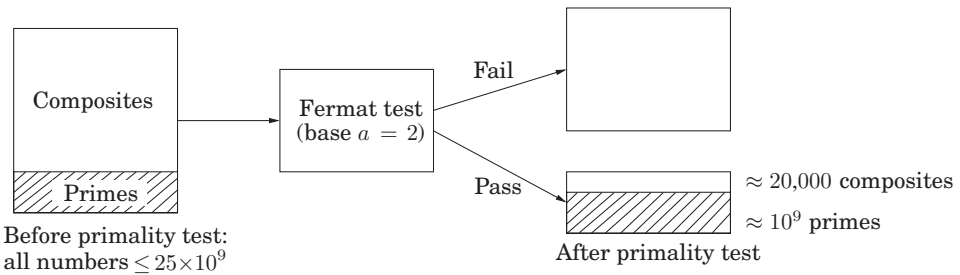
**Virtual exercises:** 1.29, 1.34, 1.46, 2.24, 2.33, 5.35, 9.8, 10.8.

this procedure has at least a $1/n$ chance of halting. Therefore on average it will halt within $O(n)$ rounds (Exercise 1.34).

Next, exactly which primality test should be used? In this application, since the numbers we are testing for primality are chosen at random rather than by an adversary, it is sufficient to perform the Fermat test with base $a = 2$ (or to be really safe, $a = 2, 3, 5$), because for random numbers the Fermat test has a much smaller

failure probability than the worst-case 1/2 bound that we proved earlier. Numbers that pass this test have been jokingly referred to as "industrial grade primes." The resulting algorithm is quite fast, generating primes that are hundreds of bits long in a fraction of a second on a PC.

The important question that remains is: what is the probability that the output of the algorithm is really prime? To answer this we must first understand how discerning the Fermat test is. As a concrete example, suppose we perform the test with base $a = 2$ for all numbers $N \leq 25 \times 10^9$. In this range, there are about $10^9$ primes, and about 20,000 composites that pass the test (see the following figure). Thus the chance of erroneously outputting a composite is approximately $20{,}000/10^9 = 2 \times 10^{-5}$. This chance of error decreases rapidly as the length of the numbers involved is increased (to the few hundred digits we expect in our applications).

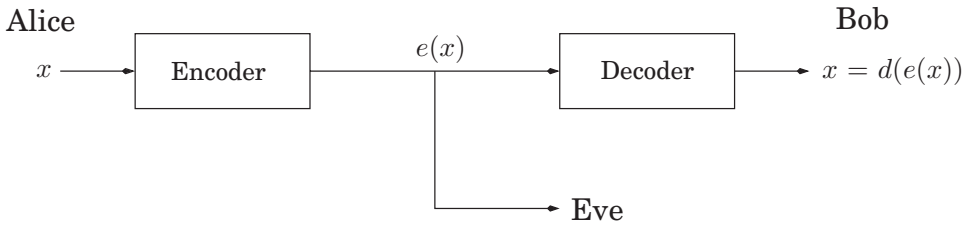



## 1.4 Cryptography

Our next topic, the Rivest-Shamir-Adleman (RSA) cryptosystem, uses all the ideas we have introduced in this chapter! It derives very strong guarantees of security by ingeniously exploiting the wide gulf between the polynomial-time computability of certain number-theoretic tasks (modular exponentiation, greatest common divisor, primality testing) and the intractability of others (factoring).

The typical setting for cryptography can be described via a cast of three characters: Alice and Bob, who wish to communicate in private, and Eve, an eavesdropper who will go to great lengths to find out what they are saying. For concreteness, let's say Alice wants to send a specific message $x$, written in binary (why not), to her friend Bob. She encodes it as $e(x)$, sends it over, and then Bob applies his decryption function $d(\cdot)$ to decode it: $d(e(x)) = x$. Here $e(\cdot)$ and $d(\cdot)$ are appropriate transformations of the messages.

Alice and Bob are worried that the eavesdropper, Eve, will intercept $e(x)$: for instance, she might be a sniffer on the network. But ideally the encryption function




$e(\cdot)$ is so chosen that without knowing $d(\cdot)$, Eve cannot do anything with the information she has picked up. In other words, knowing $e(x)$ tells her little or nothing about what $x$ might be.

For centuries, cryptography was based on what we now call *private-key protocols*. In such a scheme, Alice and Bob meet beforehand and together choose a secret codebook, with which they encrypt all future correspondence between them. Eve's only hope, then, is to collect some encoded messages and use them to at least partially figure out the codebook.

*Public-key* schemes such as RSA are significantly more subtle and tricky: they allow Alice to send a message to Bob without their ever having met before. Bob's encryption function e(·) is publicly available, and Alice can encrypt her message with this function, thereby *digitally locking* it.Only Bob knows the key to quickly unlocking this digital lock: the decryption function d(·). The point is that Alice and Bob need only perform simple calculations to lock and unlock the message respectively—operations that any pocket computing device could handle. By contrast, to unlock the message without the key, Eve must perform operations like factoring large numbers, which requires more computational power than would be afforded by the world's most powerful computers combined. This compelling guarantee enables secure Web commerce, such as sending credit card numbers to companies over the Internet.

### 1.4.1 Private-key schemes: one-time pad and AES

If Alice wants to transmit an important private message to Bob, it would be wise of her to scramble it with an encryption function,

$$e : \langle \text{messages} \rangle \rightarrow \langle \text{encoded messages} \rangle.$$

Of course, this function must be invertible—for decoding to be possible—and is therefore a bijection. Its inverse is the decryption function $d(\cdot)$.

In the *one-time pad*, Alice and Bob meet beforehand and secretly choose a binary string $r$ of the same length—say, $n$ bits—as the important message $x$ that Alice will later send. Alice's encryption function is then a *bitwise exclusive-or*, $e_r(x) = x \oplus r$: each position in the encoded message is the exclusive-or of the corresponding positions in $x$ and $r$. For instance, if $r = 01110010$, then the message 11110000 is scrambled thus:

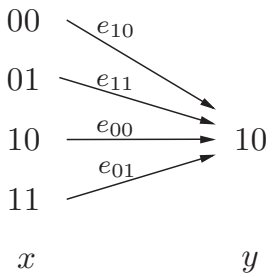
$$e_r(11110000) = 11110000 \oplus 01110010 = 10000010.$$

This function $e_r$ is a bijection from $n$-bit strings to $n$-bit strings, as evidenced by the fact that it is its own inverse!

$$e_r(e_r(x)) \;=\; (x \oplus r) \oplus r \;=\; x \oplus (r \oplus r) \;=\; x \oplus \overline{0} \;=\; x,$$

where $\overline{0}$ is the string of all zeros. Thus Bob can decode Alice's transmission by applying the same encryption function a second time: $d_r(y) = y \oplus r$.

How should Alice and Bob choose $r$ for this scheme to be secure? Simple: they should pick $r$ *at random*, flipping a coin for each bit, so that the resulting string is equally likely to be any element of $\{0, 1\}^n$. This will ensure that if Eve intercepts the encoded message $y = e_r(x)$, she gets no information about $x$. Suppose, for example, that Eve finds out $y = 10$; what can she deduce? She doesn't know $r$, and the possible values it can take all correspond to different original messages $x$:




So given what Eve knows, all possibilities for $x$ are equally likely!

The downside of the one-time pad is that it has to be discarded after use, hence the name. A second message encoded with the same pad would not be secure, because if Eve knew $x \oplus r$ and $z \oplus r$ for two messages $x$ and $z$, then she could take the exclusive-or to get $x \oplus z$, which might be important information—for example, (1) it reveals whether the two messages begin or end the same, and (2) if one message contains a long sequence of zeros (as could easily be the case if the message is an image), then the corresponding part of the other message will be exposed. Therefore the random string that Alice and Bob share has to be the combined length of all the messages they will need to exchange.

The one-time pad is a toy cryptographic scheme whose behavior and theoretical properties are completely clear. At the other end of the spectrum lies the *advanced encryption standard* (AES), a very widely used cryptographic protocol that was approved by the U.S. National Institute of Standards and Technologies in 2001. AES is once again private-key: Alice and Bob have to agree on a shared random string $r$. But this time the string is of a small fixed size, 128 to be precise (variants with 192 or 256 bits also exist), and specifies a bijection $e_r$ from 128-bit strings to 128-bit strings. The crucial difference is that this function can be used repeatedly, so for instance a long message can be encoded by splitting it into segments of 128 bits and applying $e_r$ to each segment.

The security of AES has not been rigorously established, but certainly at present the general public does not know how to break the code—to recover $x$ from $e_r(x)$—except using techniques that are not very much better than the brute-force approach of trying all possibilities for the shared string $r$.

### 1.4.2 RSA

Unlike the previous two protocols, the RSA scheme is an example of *public-key cryptography*: anybody can send a message to anybody else using publicly available information, rather like addresses or phone numbers. Each person has a public key known to the whole world and a secret key known only to him- or herself. When Alice wants to send message $x$ to Bob, she encodes it using his public key. He decrypts it using his secret key, to retrieve $x$. Eve is welcome to see as many encrypted messages for Bob as she likes, but she will not be able to decode them, under certain simple assumptions.

The RSA scheme is based heavily upon number theory. Think of messages from Alice to Bob as numbers modulo $N$; messages larger than $N$ can be broken into smaller pieces. The encryption function will then be a bijection on $\{0, 1, \ldots, N-1\}$, and the decryption function will be its inverse. What values of $N$ are appropriate, and what bijection should be used?

**Property** *Pick any two primes $p$ and $q$ and let $N = pq$. For any $e$ relatively prime to $(p-1)(q-1)$:*

1. *The mapping $x \mapsto x^e \bmod N$ is a bijection on $\{0, 1, \ldots, N-1\}$.*
2. *Moreover, the inverse mapping is easily realized: let $d$ be the inverse of $e$ modulo $(p-1)(q-1)$. Then for all $x \in \{0, \ldots, N-1\}$,*

$$(x^e)^d \equiv x \bmod N.$$

The first property tells us that the mapping $x \mapsto x^e \bmod N$ is a reasonable way to encode messages $x$; no information is lost. So, if Bob publishes $(N, e)$ as his *public key*, everyone else can use it to send him encrypted messages. The second property then tells us how decryption can be achieved. Bob should retain the value $d$ as his *secret key*, with which he can decode all messages that come to him by simply raising them to the $d$th power modulo $N$.

*Example.* Let $N = 55 = 5 \cdot 11$. Choose encryption exponent $e = 3$, which satisfies the condition $\gcd(e, (p-1)(q-1)) = \gcd(3, 40) = 1$. The decryption exponent is then $d = 3^{-1} \bmod 40 = 27$. Now for any message $x$ mod 55, the encryption of $x$ is $y = x^3 \bmod 55$, and the decryption of $y$ is $x = y^{27} \bmod 55$. So, for example, if $x = 13$, then $y = 13^3 = 52 \bmod 55$ and $13 = 52^{27} \bmod 55$.

Let's prove the assertion above and then examine the security of the scheme.

*Proof.* If the mapping $x \mapsto x^e \bmod N$ is invertible, it must be a bijection; hence statement 2 implies statement 1. To prove statement 2, we start by observing that $e$ is invertible modulo $(p-1)(q-1)$ because it is relatively prime to this number. To see that $(x^e)^d \equiv x \bmod N$, we examine the exponent: since $ed \equiv 1 \bmod (p-1)(q-1)$, we can write $ed$ in the form $1 + k(p-1)(q-1)$ for some $k$. Now we need to show that the difference

$$x^{ed} - x \;=\; x^{1+k(p-1)(q-1)} - x$$

is always 0 modulo $N$. The second form of the expression is convenient because it can be simplified using Fermat's little theorem. It is divisible by $p$ (since $x^{p-1} \equiv 1 \bmod p$) and likewise by $q$. Since $p$ and $q$ are primes, this expression must also be divisible by their product $N$. Hence $x^{ed} - x = x^{1+k(p-1)(q-1)} - x \equiv 0 \pmod N$, exactly as we need. ∎

**Figure 1.9** RSA.

Bob chooses his public and secret keys.

- He starts by picking two large ($n$-bit) random primes $p$ and $q$.
- His public key is $(N, e)$ where $N = pq$ and $e$ is a $2n$-bit number relatively prime to $(p-1)(q-1)$. A common choice is $e = 3$ because it permits fast encoding.
- His secret key is $d$, the inverse of $e$ modulo $(p-1)(q-1)$, computed using the extended Euclid algorithm.

Alice wishes to send message $x$ to Bob.

- She looks up his public key $(N, e)$ and sends him $y = (x^e \bmod N)$, computed using an efficient modular exponentiation algorithm.
- He decodes the message by computing $y^d \bmod N$.

The RSA protocol is summarized in Figure 1.9. It is certainly convenient: the computations it requires of Alice and Bob are elementary. But how secure is it against Eve?

The security of RSA hinges upon a simple assumption:

*Given $N$, $e$, and $y = x^e \bmod N$, it is computationally intractable to determine $x$.*

This assumption is quite plausible. How might Eve try to guess $x$? She could experiment with all possible values of $x$, each time checking whether $x^e \equiv y \bmod N$, but this would take exponential time. Or she could try to factor $N$ to retrieve $p$ and $q$, and then figure out $d$ by inverting $e$ modulo $(p-1)(q-1)$, but we believe factoring to be hard. Intractability is normally a source of dismay; the insight of RSA lies in using it to advantage.

## 1.5 Universal hashing

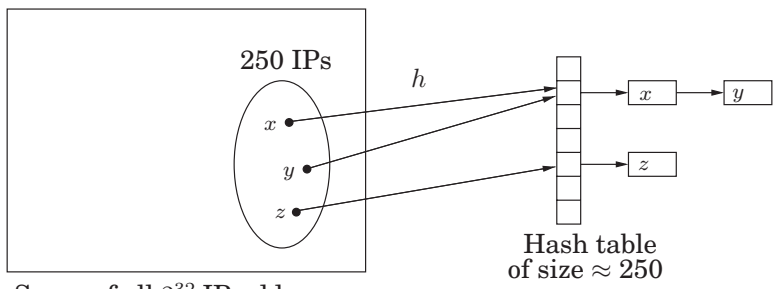
We end this chapter with an application of number theory to the design of *hash functions.* Hashing is a very useful method of storing data items in a table so as to support insertions, deletions, and lookups.

Suppose, for instance, that we need to maintain an ever-changing list of about 250 IP (Internet protocol) addresses, perhaps the addresses of the currently active customers of a Web service. (Recall that an IP address consists of 32 bits encoding the location of a computer on the Internet, usually shown broken down into four 8-bit fields, for example, 128.32.168.80.) We could obtain fast lookup times if we maintained the records in an array indexed by IP address. But this would be very wasteful of memory: the array would have $2^{32} \approx 4 \times 10^9$ entries, the vast majority of them blank. Or alternatively, we could use a linked list of just the 250 records. But then accessing records would be very slow, taking time proportional to 250, the total number of customers. Is there a way to get the best of both worlds, to use an amount of memory that is proportional to the number of customers and yet also achieve fast lookup times? This is exactly where hashing comes in.

### 1.5.1 Hash tables

Here's a high-level view of hashing. We will give a short "nickname" to each of the $2^{32}$ possible IP addresses. You can think of this short name as just a number between 1 and 250 (we will later adjust this range very slightly). Thus many IP addresses will inevitably have the same nickname; however, we hope that most of the 250 IP addresses of our particular customers are assigned distinct names, and we will store their records in an array of size 250 indexed by these names. What if there is more than one record associated with the same name? Easy: each entry of the array points to a linked list containing all records with that name. So the total amount of storage is proportional to 250, the number of customers, and is independent of the total number of possible IP addresses. Moreover, if not too many customer IP addresses are assigned the same name, lookup is fast, because the average size of the linked list we have to scan through is small.

But how do we assign a short name to each IP address? This is the role of a *hash function*: in our example, a function $h$ that maps IP addresses to positions in a table of length about 250 (the expected number of data items). The name assigned to an



IP address $x$ is thus $h(x)$, and the record for $x$ is stored in position $h(x)$ of the table. As described before, each position of the table is in fact a *bucket*, a linked list that contains all current IP addresses that map to it. Hopefully, there will be very few buckets that contain more than a handful of IP addresses.

### 1.5.2 Families of hash functions

Designing hash functions is tricky. A hash function must in some sense be "random" (so that it scatters data items around), but it should also be a function and therefore "consistent" (so that we get the same result every time we apply it). And the statistics of the data items may work against us. In our example, one possible hash function would map an IP address to the 8-bit number that is its last segment: $h(128.32.168.80) = 80$. A table of $n = 256$ buckets would then be required. But is this a good hash function? Not if, for example, the last segment of an IP address tends to be a small (single- or double-digit) number; then low-numbered buckets would be crowded. Taking the first segment of the IP address also invites disaster—for example, if most of our customers come from Asia.

There is nothing inherently wrong with these two functions. If our 250 IP addresses were uniformly drawn from among all $N = 2^{32}$ possibilities, then these functions would behave well. The problem is we have no guarantee that the distribution of IP addresses is uniform.

Conversely, there is no single hash function, no matter how sophisticated, that behaves well on all sets of data. Since a hash function maps $2^{32}$ IP addresses to just 250 names, there must be a collection of at least $2^{32}/250 \approx 2^{24} \approx 16{,}000{,}000$ IP addresses that are assigned the same name (or, in hashing terminology, "collide"). If many of these show up in our customer set, we're in trouble.

Obviously, we need some kind of randomization. Here's an idea: let us pick a hash function *at random* from some class of functions. We will then show that, no matter what set of 250 IP addresses we actually care about, most choices of the hash function will give very few collisions among these addresses.

To this end, we need to define a class of hash functions from which we can pick at random; and this is where we turn to number theory. Let us take the number of buckets to be not 250 but $n = 257$—*a prime number*! And we consider every IP address $x$ as a quadruple $x = (x_1, \ldots, x_4)$ of integers modulo $n$—recall that it is in fact a quadruple of integers between 0 and 255, so there is no harm in this. We can define a function $h$ from IP addresses to a number mod $n$ as follows: fix any four numbers mod $n = 257$, say 87, 23, 125, and 4. Now map the IP address $(x_1, \ldots, x_4)$ to $h(x_1, \ldots, x_4) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \bmod 257$. Indeed, any four numbers mod $n$ define a hash function.

For any four coefficients $a_1, \ldots, a_4 \in \{0, 1, \ldots, n-1\}$, write $a = (a_1, a_2, a_3, a_4)$ and define $h_a$ to be the following hash function:

$$h_a(x_1, \ldots, x_4) = \sum_{i=1}^{4} a_i \cdot x_i \bmod n.$$

We will show that if we pick these coefficients $a$ at random, then $h_a$ is very likely to be good in the following sense.

**Property** *Consider any pair of distinct IP addresses* $x = (x_1, \ldots, x_4)$ *and* $y = (y_1, \ldots, y_4)$. *If the coefficients* $a = (a_1, a_2, a_3, a_4)$ *are chosen uniformly at random from* $\{0, 1, \ldots, n-1\}$, *then*

$$\Pr\{h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)\} = \frac{1}{n}.$$

In other words, the chance that $x$ and $y$ collide under $h_a$ is the same as it would be if each were assigned nicknames randomly and independently. This condition guarantees that the expected lookup time for any item is small. Here's why. If we wish to look up $x$ in our hash table, the time required is dominated by the size of its bucket, that is, the number of items that are assigned the same name as $x$. But there are only 250 items in the hash table, and the probability that any one item gets the same name as $x$ is $1/n = 1/257$. Therefore the expected number of items that are assigned the same name as $x$ by a randomly chosen hash function $h_a$ is $250/257 \approx 1$, which means the expected size of $x$'s bucket is less than 2.[1]

Let us now prove the preceding property.

*Proof.* Since $x = (x_1, \ldots, x_4)$ and $y = (y_1, \ldots, y_4)$ are distinct, these quadruples must differ in some component; without loss of generality let us assume that $x_4 \neq y_4$. We wish to compute the probability $\Pr[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)]$, that is, the probability that $\sum_{i=1}^{4} a_i \cdot x_i \equiv \sum_{i=1}^{4} a_i \cdot y_i \bmod n$. This last equation can be rewritten as

$$\sum_{i=1}^{3} a_i \cdot (x_i - y_i) \equiv a_4 \cdot (y_4 - x_4) \bmod n. \tag{1}$$

Suppose that we draw a random hash function $h_a$ by picking $a = (a_1, a_2, a_3, a_4)$ at random. We start by drawing $a_1$, $a_2$, and $a_3$, and then we pause and think: What is the probability that the last drawn number $a_4$ is such that equation (1) holds? So far the left-hand side of equation (1) evaluates to some number, call it $c$. And since $n$ is prime and $x_4 \neq y_4$, $(y_4 - x_4)$ has a unique inverse modulo $n$. Thus for equation (1) to hold, the last number $a_4$ must be precisely $c \cdot (y_4 - x_4)^{-1} \bmod n$, out of its $n$ possible values. The probability of this happening is $1/n$, and the proof is complete. ∎

Let us step back and see what we just achieved. Since we have no control over the set of data items, we decided instead to select a hash function $h$ uniformly at

[1] When a hash function $h_a$ is chosen at random, let the random variable $Y_i$ (for $i = 1, \ldots, 250$) be 1 if item $i$ gets the same name as $x$ and 0 otherwise. So the expected value of $Y_i$ is $1/n$. Now, $Y = Y_1 + Y_2 + \cdots + Y_{250}$ is the number of items which get the same name as $x$, and by linearity of expectation, the expected value of $Y$ is simply the sum of the expected values of $Y_1$ through $Y_{250}$. It is thus $250/n = 250/257$.

random from among a family $\mathcal{H}$ of hash functions. In our example,

$$\mathcal{H} = \{h_a : a \in \{0, \ldots, n-1\}^4\}.$$

To draw a hash function uniformly at random from this family, we just draw four numbers $a_1, \ldots, a_4$ modulo $n$. (Incidentally, notice that the two simple hash functions we considered earlier, namely, taking the last or the first 8-bit segment, belong to this class. They are $h_{(0,0,0,1)}$ and $h_{(1,0,0,0)}$, respectively.) And we insisted that the family have the following property:

> *For any two distinct data items $x$ and $y$, exactly $|\mathcal{H}|/n$ of all the hash functions in $\mathcal{H}$ map $x$ and $y$ to the same bucket, where $n$ is the number of buckets.*

A family of hash functions with this property is called *universal*. In other words, for any two data items, the probability these items collide is $1/n$ if the hash function is randomly drawn from a universal family. This is also the collision probability if we map $x$ and $y$ to buckets uniformly at random—in some sense the gold standard of hashing. We then showed that this property implies that hash table operations have good performance *in expectation*.

This idea, motivated as it was by the hypothetical IP address application, can of course be applied more generally. Start by choosing the table size $n$ to be some prime number that is a little larger than the number of items expected in the table (there is usually a prime number close to any number we start with; actually, to ensure that hash table operations have good performance, it is better to have the size of the hash table be about twice as large as the number of items). Next assume that the size of the domain of all data items is $N = n^k$, a power of $n$ (if we need to overestimate the true number of data items, so be it). Then each data item can be considered as a $k$-tuple of integers modulo $n$, and $\mathcal{H} = \{h_a : a \in \{0, \ldots, n-1\}^k\}$ is a universal family of hash functions.

## Exercises

1.1. Show that in any base $b \geq 2$, the sum of any three single-digit numbers is at most two digits long.

1.2. Show that any binary integer is at most four times as long as the corresponding decimal integer. For very large numbers, what is the ratio of these two lengths, approximately?

1.3. A $d$-ary tree is a rooted tree in which each node has at most $d$ children. Show that any $d$-ary tree with $n$ nodes must have a depth of $\Omega(\log n / \log d)$. Can you give a precise formula for the minimum depth it could possibly have?

1.4. Show that

$$\log(n!) \quad = \quad \Theta(n \log n).$$

(*Hint:* To show an upper bound, compare $n!$ with $n^n$. To show a lower bound, compare it with $(n/2)^{n/2}$.)

1.5. Unlike a decreasing geometric series, the sum of the *harmonic series* $1, 1/2, 1/3, 1/4, 1/5, \ldots$ diverges; that is,

$$\sum_{i=1}^{\infty} \frac{1}{i} = \infty.$$

It turns out that, for large $n$, the sum of the first $n$ terms of this series can be well approximated as

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + \gamma,$$

where ln is natural logarithm (log base $e = 2.718\ldots$) and $\gamma$ is a particular constant $0.57721\ldots$. Show that

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n).$$

(*Hint:* To show an upper bound, decrease each denominator to the next power of two. For a lower bound, increase each denominator to the next power of 2.)

1.6. Prove that the grade-school multiplication algorithm (page 13), when applied to binary numbers, always gives the right answer.

1.7. How long does the recursive multiplication algorithm (page 15) take to multiply an $n$-bit number by an $m$-bit number? Justify your answer.

1.8. Justify the correctness of the recursive division algorithm given in page 15, and show that it takes time $O(n^2)$ on $n$-bit inputs.

1.9. Starting from the definition of $x \equiv y \bmod N$ (namely, that $N$ divides $x - y$), justify the substitution rule

$$x \equiv x' \bmod N,\ y \equiv y' \bmod N \ \Rightarrow\ x + y \equiv x' + y' \bmod N,$$

and also the corresponding rule for multiplication.

1.10. Show that if $a \equiv b \pmod N$ and if $M$ divides $N$ then $a \equiv b \pmod M$.

1.11. Is $4^{1536} - 9^{4824}$ divisible by 35?

1.12. What is $2^{2^{2006}} \pmod 3$?

1.13. Is the difference of $5^{30,000}$ and $6^{123,456}$ a multiple of 31?

1.14. Suppose you want to compute the $n$th Fibonacci number $F_n$, modulo an integer $p$. Can you find an efficient way to do this? (*Hint:* Recall Exercise 0.4.)

1.15. Determine necessary and sufficient conditions on $x$ and $c$ so that the following holds: for any $a, b$, if $ax \equiv bx \bmod c$, then $a \equiv b \bmod c$.

1.16. The algorithm for computing $a^b \bmod c$ by repeated squaring does not necessarily lead to the minimum number of multiplications. Give an example of $b > 10$ where the exponentiation can be performed using fewer multiplications, by some other method.

1.17. Consider the problem of computing $x^y$ for given integers $x$ and $y$: we want the *whole* answer, not modulo a third integer. We know two algorithms for doing this: the iterative algorithm which performs $y - 1$ multiplications by $x$; and the recursive algorithm based on the binary expansion of $y$.

Compare the time requirements of these two algorithms, assuming that the time to multiply an $n$-bit number by an $m$-bit number is $O(mn)$.

1.18. Compute gcd(210, 588) two different ways: by finding the factorization of each number, and by using Euclid's algorithm.

1.19. The *Fibonacci numbers* $F_0, F_1, \ldots$ are given by the recurrence $F_{n+1} = F_n + F_{n-1}$, $F_0 = 0$, $F_1 = 1$. Show that for any $n \geq 1$, $\gcd(F_{n+1}, F_n) = 1$.

1.20. Find the inverse of: $20 \bmod 79$, $3 \bmod 62$, $21 \bmod 91$, $5 \bmod 23$.

1.21. How many integers modulo $11^3$ have inverses? (Note: $11^3 = 1331$.)

1.22. Prove or disprove: If $a$ has an inverse modulo $b$, then $b$ has an inverse modulo $a$.

1.23. Show that if $a$ has a multiplicative inverse modulo $N$, then this inverse is unique (modulo $N$).

1.24. If $p$ is prime, how many elements of $\{0, 1, \ldots, p^n - 1\}$ have an inverse modulo $p^n$?

1.25. Calculate $2^{125} \bmod 127$ using any method you choose. (*Hint:* 127 is prime.)

1.26. What is the least significant decimal digit of $17^{17^{17}}$? (*Hint:* For distinct primes $p, q$, and any $a$ relatively prime to $pq$, we proved the formula $a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$ in Section 1.4.2.)

1.27. Consider an RSA key set with $p = 17, q = 23, N = 391$, and $e = 3$ (as in Figure 1.9). What value of $d$ should be used for the secret key? What is the encryption of the message $M = 41$?

1.28. In an RSA cryptosystem, $p = 7$ and $q = 11$ (as in Figure 1.9). Find appropriate exponents $d$ and $e$.

1.29. Let $[m]$ denote the set $\{0, 1, \ldots, m - 1\}$. For each of the following families of hash functions, say whether or not it is universal, and determine how many random bits are needed to choose a function from the family.

(a) $H = \{h_{a_1,a_2} : a_1, a_2 \in [m]\}$, where $m$ is a fixed prime and

$$h_{a_1,a_2}(x_1, x_2) = a_1 x_1 + a_2 x_2 \bmod m.$$

Notice that each of these functions has signature $h_{a_1,a_2} : [m]^2 \to [m]$, that is, it maps a pair of integers in $[m]$ to a single integer in $[m]$.

(b) $H$ is as before, except that now $m = 2^k$ is some fixed power of 2.

(c) $H$ is the set of all functions $f : [m] \to [m - 1]$.

1.30. The grade-school algorithm for multiplying two $n$-bit binary numbers $x$ and $y$ consists of adding together $n$ copies of $x$, each appropriately left-shifted. Each copy, when shifted, is at most $2n$ bits long.

In this problem, we will examine a scheme for adding $n$ binary numbers, each $m$ bits long, using a *circuit* or a *parallel architecture*. The main parameter of interest in this question is therefore the depth of the circuit or the longest path from the input to the output of the circuit. This determines the total time taken for computing the function.

To add two $m$-bit binary numbers naively, we must wait for the carry bit from position $i-1$ before we can figure out the $i$th bit of the answer. This leads to a circuit of depth $O(m)$. However carry lookahead circuits (see wikipedia.com if you want to know more about this) can add in $O(\log m)$ depth.

(a) Assuming you have carry lookahead circuits for addition, show how to add $n$ numbers each $m$ bits long using a circuit of depth $O((\log n)(\log m))$.

(b) When adding *three* $m$-bit binary numbers $x+y+z$, there is a trick we can use to parallelize the process. Instead of carrying out the addition completely, we can re-express the result as the sum of just *two* binary numbers $r+s$, such that the $i$th bits of $r$ and $s$ can be computed independently of the other bits. Show how this can be done. (*Hint:* One of the numbers represents carry bits.)

(c) Show how to use the trick from the previous part to design a circuit of depth $O(\log n)$ for multiplying two $n$-bit numbers.

1.31. Consider the problem of computing $N! = 1 \cdot 2 \cdot 3 \cdots N$.

(a) If $N$ is an $n$-bit number, how many bits long is $N!$, approximately (in $\Theta(\cdot)$ form)?

(b) Give an algorithm to compute $N!$ and analyze its running time.

1.32. A positive integer $N$ is a *power* if it is of the form $q^k$, where $q, k$ are positive integers and $k > 1$.

(a) Give an efficient algorithm that takes as input a number $N$ and determines whether it is a square, that is, whether it can be written as $q^2$ for some positive integer $q$. What is the running time of your algorithm?

(b) Show that if $N = q^k$ (with $N$, $q$, and $k$ all positive integers), then either $k \leq \log N$ or $N = 1$.

(c) Give an efficient algorithm for determining whether a positive integer $N$ is a power. Analyze its running time.

1.33. Give an efficient algorithm to compute the *least common multiple* of two $n$-bit numbers $x$ and $y$, that is, the smallest number divisible by both $x$ and $y$. What is the running time of your algorithm as a function of $n$?

1.34. On page 29, we claimed that since about a $1/n$ fraction of $n$-bit numbers are prime, on average it is sufficient to draw $O(n)$ random $n$-bit numbers before hitting a prime. We now justify this rigorously.

Suppose a particular coin has a probability $p$ of coming up heads. How many times must you toss it, on average, before it comes up heads? (*Hint:* Method 1: start by showing that the correct expression is $\sum_{i=1}^{\infty} i(1-p)^{i-1}p$. Method 2: if $E$ is the average number of coin tosses, show that $E = 1 + (1-p)E$.)

1.35. Wilson's theorem says that a number $N$ is prime if and only if

$$(N-1)! \equiv -1 \pmod{N}.$$

(a) If $p$ is prime, then we know every number $1 \leq x < p$ is invertible modulo $p$. Which of these numbers are their own inverse?

(b) By pairing up multiplicative inverses, show that $(p-1)! \equiv -1 \pmod{p}$ for prime $p$.

(c) Show that if $N$ is *not* prime, then $(N-1)! \not\equiv -1 \pmod{N}$. (*Hint:* Consider $d = \gcd(N, (N-1)!)$.)

(d) Unlike Fermat's Little theorem, Wilson's theorem is an if-and-only-if condition for primality. Why can't we immediately base a primality test on this rule?

1.36. *Square roots.* In this problem, we'll see that it is easy to compute square roots modulo a prime $p$ with $p \equiv 3 \pmod{4}$.

(a) Suppose $p \equiv 3 \pmod{4}$. Show that $(p+1)/4$ is an integer.

(b) We say $x$ is a *square root* of $a$ modulo $p$ if $a \equiv x^2 \pmod{p}$. Show that if $p \equiv 3 \pmod{4}$ and if $a$ has a square root modulo $p$, then $a^{(p+1)/4}$ is such a square root.

1.37. *The Chinese remainder theorem.*

(a) Make a table with three columns. The first column is all numbers from 0 to 14. The second is the residues of these numbers modulo 3; the third column is the residues modulo 5.

(b) Prove that if $p$ and $q$ are distinct primes, then for every pair $(j, k)$ with $0 \leq j < p$ and $0 \leq k < q$, there is a unique integer $0 \leq i < pq$ such that $i \equiv j \bmod p$ and $i \equiv k \bmod q$. (*Hint:* Prove that no two different $i$'s in this range can have the same $(j, k)$, and then count.)

(c) In this one-to-one correspondence between integers and pairs, it is easy to go from $i$ to $(j, k)$. Prove that the following formula takes you the other way:

$$i = \{j \cdot q \cdot (q^{-1} \bmod p) + k \cdot p \cdot (p^{-1} \bmod q)\} \bmod pq.$$

(d) Can you generalize parts (b) and (c) to more than two primes?

1.38. To see if a number, say 562437487, is divisible by 3, you just add up the digits of its decimal representation, and see if the result is divisible by 3. ($5+6+2+4+3+7+4+8+7 = 46$, so it is not divisible by 3.)

To see if the same number is divisible by 11, you can do this: subdivide the number into pairs of digits, from the right-hand end $(87, 74, 43, 62, 5)$, add these numbers, and see if the sum is divisible by 11 (if it's too big, repeat).

How about 37? To see if the number is divisible by 37, subdivide it into triples from the end $(487, 437, 562)$ add these up, and see if the sum is divisible by 37.

This is true for any prime $p$ other than 2 and 5. That is, for any prime $p \neq 2, 5$, there is an integer $r$ such that in order to see if $p$ divides a decimal

number $n$, we break $n$ into $r$-tuples of decimal digits (starting from the right-hand end), add up these $r$-tuples, and check if the sum is divisible by $p$.

(a) What is the smallest such $r$ for $p = 13$? For $p = 17$?

(b) Show that $r$ is a divisor of $p - 1$.

1.39. Give a polynomial-time algorithm for computing $a^{b^c} \bmod p$, given $a$, $b$, $c$, and prime $p$.

1.40. Show that if $x$ is a nontrivial square root of 1 modulo $N$, that is, if $x^2 \equiv 1 \bmod N$ but $x \not\equiv \pm 1 \bmod N$, then $N$ must be composite. (For instance, $4^2 \equiv 1 \bmod 15$ but $4 \not\equiv \pm 1 \bmod 15$; thus 4 is a nontrivial square root of 1 modulo 15.)

1.41. *Quadratic residues.* Fix a positive integer $N$. We say that $a$ is a *quadratic residue* modulo $N$ if there exists $x$ such that $a \equiv x^2 \bmod N$.

(a) Let $N$ be an odd prime and $a$ be a non-zero quadratic residue modulo $N$. Show that there are exactly two values in $\{0, 1, \ldots, N-1\}$ satisfying $x^2 \equiv a \bmod N$.

(b) Show that if $N$ is an odd prime, there are exactly $(N+1)/2$ quadratic residues in $\{0, 1, \ldots, N-1\}$.

(c) Give an example of positive integers $a$ and $N$ such that $x^2 \equiv a \bmod N$ has more than two solutions in $\{0, 1, \ldots, N-1\}$.

1.42. Suppose that instead of using a composite $N = pq$ in the RSA cryptosystem (Figure 1.9), we simply use a prime modulus $p$. As in RSA, we would have an encryption exponent $e$, and the encryption of a message $m \bmod p$ would be $m^e \bmod p$. Prove that this new cryptosystem is not secure, by giving an efficient algorithm to decrypt: that is, an algorithm that given $p$, $e$, and $m^e \bmod p$ as input, computes $m \bmod p$. Justify the correctness and analyze the running time of your decryption algorithm.

1.43. In the RSA cryptosystem, Alice's public key $(N, e)$ is available to everyone. Suppose that her private key $d$ is compromised and becomes known to Eve. Show that if $e = 3$ (a common choice) then Eve can efficiently factor $N$.

1.44. Alice and her three friends are all users of the RSA cryptosystem. Her friends have public keys $(N_i, e_i = 3)$, $i = 1, 2, 3$, where as always, $N_i = p_i q_i$ for randomly chosen $n$-bit primes $p_i, q_i$. Show that if Alice sends the same $n$-bit message $M$ (encrypted using RSA) to each of her friends, then anyone who intercepts all three encrypted messages will be able to efficiently recover $M$. (*Hint:* It helps to have solved problem 1.37 first.)

1.45. *RSA and digital signatures.* Recall that in the RSA public-key cryptosystem, each user has a public key $P = (N, e)$ and a secret key $d$. In a *digital signature scheme*, there are two algorithms, `sign` and `verify`. The `sign` procedure takes a message and a secret key, then outputs a signature $\sigma$. The `verify` procedure takes a public key $(N, e)$, a signature $\sigma$, and a message $M$, then returns "true" if $\sigma$ could have been created by `sign` (when called with message $M$ and the secret key corresponding to the public key $(N, e)$); "false" otherwise.

(a) Why would we want digital signatures?

(b) An RSA signature consists of $\texttt{sign}(M, d) = M^d \pmod N$, where $d$ is a secret key and $N$ is part of the public key. Show that anyone who knows the public key $(N, e)$ can perform $\texttt{verify}((N, e), M^d, M)$, i.e., they can check that a signature really was created by the private key. Give an implementation and prove its correctness.

(c) Generate your own RSA modulus $N = pq$, public key $e$, and private key $d$ (you don't need to use a computer). Pick $p$ and $q$ so you have a 4-digit modulus and work by hand. Now sign your name using the private exponent of this RSA modulus. To do this you will need to specify some one-to-one mapping from strings to integers in $[0, N - 1]$. Specify any mapping you like. Give the mapping from your name to numbers $m_1, m_2, \ldots m_k$, then sign the first number by giving the value $m_1^d \pmod N$, and finally show that $(m_1^d)^e = m_1 \pmod N$.

(d) Alice wants to write a message that looks like it was digitally signed by Bob. She notices that Bob's public RSA key is $(17, 391)$. To what exponent should she raise her message?

1.46. *Digital signatures, continued.* Consider the signature scheme of Exercise 1.45.

(a) Signing involves decryption, and is therefore risky. Show that if Bob agrees to sign anything he is asked to, Eve can take advantage of this and decrypt any message sent by Alice to Bob.

(b) Suppose that Bob is more careful, and refuses to sign messages if their signatures look suspiciously like text. (We assume that a randomly chosen message—that is, a random number in the range $\{1, \ldots, N - 1\}$—is very unlikely to look like text.) Describe a way in which Eve can nevertheless still decrypt messages from Alice to Bob, by getting Bob to sign messages whose signatures look random.

# *Chapter 2*
# Divide-and-conquer algorithms

The *divide-and-conquer* strategy solves a problem by:

1. Breaking it into *subproblems* that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

The real work is done piecemeal, in three different places: in the partitioning of problems into subproblems; at the very tail end of the recursion, when the subproblems are so small that they are solved outright; and in the gluing together of partial answers. These are held together and coordinated by the algorithm's core recursive structure.

As an introductory example, we'll see how this technique yields a new algorithm for multiplying numbers, one that is much more efficient than the method we all learned in elementary school!

## 2.1 Multiplication

The mathematician Carl Friedrich Gauss (1777–1855) once noticed that although the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to involve *four* real-number multiplications, it can in fact be done with just *three*: $ac$, $bd$, and $(a + b)(c + d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd.$$

In our big-$O$ way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity. But this modest improvement becomes very significant *when applied recursively*.

Let's move away from complex numbers and see how this helps with regular multiplication. Suppose $x$ and $y$ are two $n$-bit integers, and assume for convenience that $n$ is a power of 2 (the more general case is hardly any different). As a first step toward multiplying $x$ and $y$, split each of them into their left and right halves, which

Carl Friedrich Gauss
1777–1855



are $n/2$ bits long:

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$

For instance, if $x = 10110110_2$ (the subscript 2 means "binary") then $x_L = 1011_2$, $x_R = 0110_2$, and $x = 1011_2 \times 2^4 + 0110_2$. The product of $x$ and $y$ can then be rewritten as

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

We will compute $xy$ via the expression on the right. The additions take linear time, as do the multiplications by powers of 2 (which are merely left-shifts). The significant operations are the four $n/2$-bit multiplications, $x_L y_L$, $x_L y_R$, $x_R y_L$, $x_R y_R$; these we can handle by four recursive calls. Thus our method for multiplying $n$-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers (four subproblems of half the size), and then evaluates the preceding expression in $O(n)$ time. Writing $T(n)$ for the overall running time on $n$-bit inputs, we get the *recurrence relation*

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

We will soon see general strategies for solving such equations. In the meantime, this particular one works out to $O(n^2)$, the same running time as the traditional grade-school multiplication technique. So we have a radically new algorithm, but we haven't yet made any progress in efficiency. How can our method be sped up?

This is where Gauss's trick comes to mind. Although the expression for $xy$ seems to demand four $n/2$-bit multiplications, as before just three will do: $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, since $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$. The resulting

**Figure 2.1** A divide-and-conquer algorithm for integer multiplication.

```
function multiply(x, y)
Input: n-bit positive integers x and y
Output: Their product

if n = 1: return xy

x_L, x_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of x
y_L, y_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of y

P_1 = multiply(x_L, y_L)
P_2 = multiply(x_R, y_R)
P_3 = multiply(x_L + x_R, y_L + y_R)
return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^{n/2} + P_2
```

algorithm, shown in Figure 2.1, has an improved running time of[1]

$$T(n) = 3T(n/2) + O(n).$$

The point is that now the constant factor improvement, from 4 to 3, occurs *at every level of the recursion*, and this compounding effect leads to a dramatically lower time bound of $O(n^{1.59})$.

This running time can be derived by looking at the algorithm's pattern of recursive calls, which form a tree structure, as in Figure 2.2. Let's try to understand the shape of this tree. At each successive level of recursion the subproblems get halved in size. At the $(\log_2 n)^{\text{th}}$ level, the subproblems get down to size 1, and so the recursion ends. Therefore, the height of the tree is $\log_2 n$. The branching factor is 3—each problem recursively produces three smaller ones—with the result that at depth $k$ in the tree there are $3^k$ subproblems, each of size $n/2^k$.

For each subproblem, a linear amount of work is done in identifying further subproblems and combining their answers. Therefore the total time spent at depth $k$ in the tree is

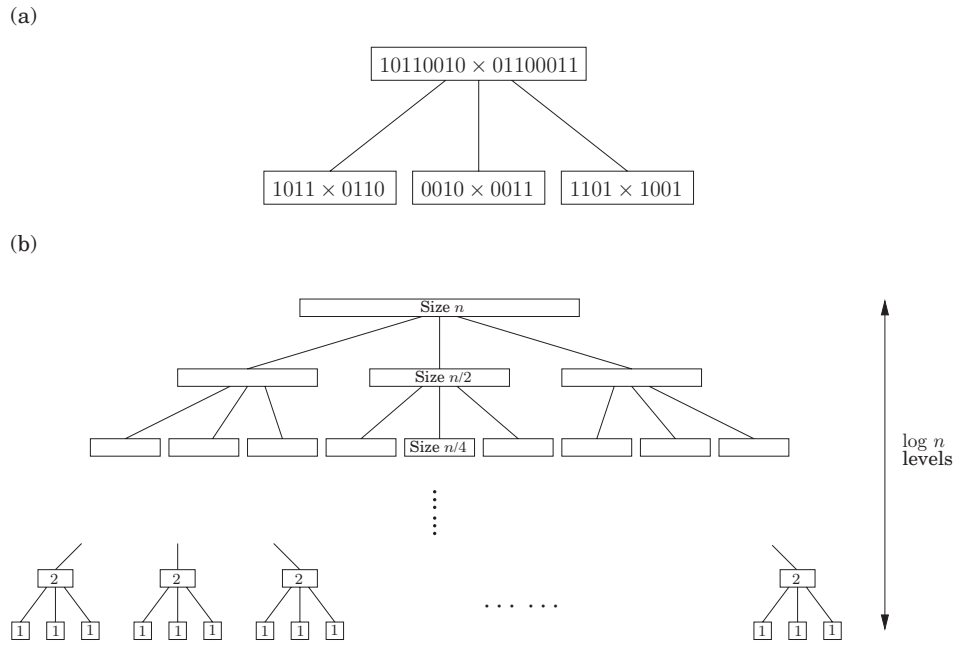
$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n).$$

[1] Actually, the recurrence should read

$$T(n) \leq 3T(n/2 + 1) + O(n)$$

since the numbers $(x_L + x_R)$ and $(y_L + y_R)$ could be $n/2 + 1$ bits long. The one we're using is simpler to deal with and can be seen to imply exactly the same big-$O$ running time.

**Figure 2.2** Divide-and-conquer integer multiplication. (a) Each problem is divided into three subproblems. (b) The levels of recursion.




At the very top level, when $k = 0$, this works out to $O(n)$. At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n})$, which can be rewritten as $O(n^{\log_2 3})$ (do you see why?). Between these two endpoints, the work done increases *geometrically* from $O(n)$ to $O(n^{\log_2 3})$, by a factor of 3/2 per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series: such is the rapidity of the increase (Exercise 0.2). Therefore the overall running time is $O(n^{\log_2 3})$, which is about $O(n^{1.59})$.

In the absence of Gauss's trick, the recursion tree would have the same height, but the branching factor would be 4. There would be $4^{\log_2 n} = n^2$ leaves, and therefore the running time would be at least this much. In divide-and-conquer algorithms, the number of subproblems translates into the branching factor of the recursion tree; small changes in this coefficient can have a big impact on running time.

A practical note: it generally does not make sense to recurse all the way down to 1 bit. For most processors, 16- or 32-bit multiplication is a single operation, so by the time the numbers get into this range they should be handed over to the built-in procedure.

Finally, the eternal question: *Can we do better?* It turns out that even faster algorithms for multiplying numbers exist, based on another important divide-and-conquer algorithm: the fast Fourier transform, to be explained in Section 2.6.

## 2.2 Recurrence relations

Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size $n$ by recursively solving, say, $a$ subproblems of size $n/b$ and then combining these answers in $O(n^d)$ time, for some $a, b, d > 0$ (in the multiplication algorithm, $a = 3$, $b = 2$, and $d = 1$). Their running time can therefore be captured by the equation $T(n) = aT(\lceil n/b \rceil) + O(n^d)$. We next derive a closed-form solution to this general recurrence so that we no longer have to solve it explicitly in each new instance.

**Master theorem**[2] *If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then*

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

This single theorem tells us the running times of most of the divide-and-conquer procedures we are likely to use.

*Proof.* To prove the claim, let's start by assuming for the sake of convenience that $n$ is a power of $b$. This will not influence the final bound in any important way—after all, $n$ is at most a multiplicative factor of $b$ away from some power of $b$ (Exercise 2.2)—and it will allow us to ignore the rounding effect in $\lceil n/b \rceil$.

Next, notice that the size of the subproblems decreases by a factor of $b$ with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree. Its branching factor is $a$, so the $k$th level of the tree is made up of $a^k$ subproblems, each of size $n/b^k$ (Figure 2.3). The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

As $k$ goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio $a/b^d$. Finding the sum of such a series in big-$O$ notation is easy (Exercise 0.2), and comes down to three cases.

1. The ratio is less than 1.
   Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.
2. The ratio is greater than 1.
   The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$:
   $$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$
3. The ratio is exactly 1.
   In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

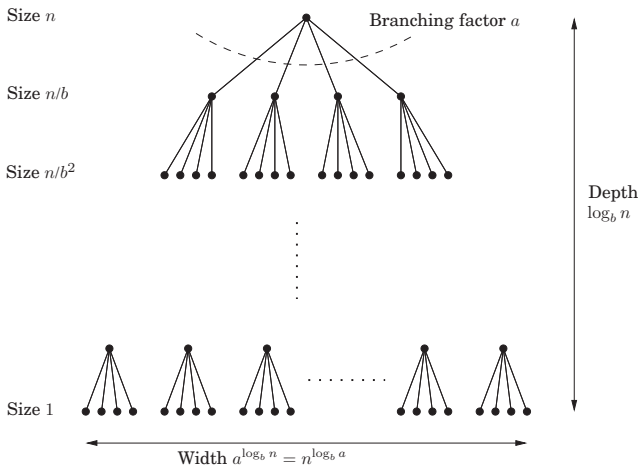
These cases translate directly into the three contingencies in the theorem statement. ∎

[2] There are even more general results of this type, but we will not be needing them.

### Binary search

The ultimate divide-and-conquer algorithm is, of course, *binary search:* to find a key $k$ in a large file containing keys $z[0, 1, \ldots, n-1]$ in sorted order, we first compare $k$ with $z[n/2]$, and depending on the result we recurse either on the first half of the file, $z[0, \ldots, n/2-1]$, or on the second half, $z[n/2, \ldots, n-1]$. The recurrence now is $T(n) = T(\lceil n/2 \rceil) + O(1)$, which is the case $a = 1$, $b = 2$, $d = 0$. Plugging into our master theorem we get the familiar solution: a running time of just $O(\log n)$.

**Figure 2.3** Each problem of size $n$ is divided into $a$ subproblems of size $n/b$.




## 2.3 Mergesort

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then *merge* the two sorted sublists.

```
function mergesort(a[1...n])
Input: An array of numbers a[1...n]
Output: A sorted version of this array

if n > 1:
   return merge(mergesort(a[1...⌊n/2⌋]),
                mergesort(a[⌊n/2⌋ + 1...n]))
else:
   return a
```

The correctness of this algorithm is self-evident, as long as a correct `merge` subroutine is specified. If we are given two sorted arrays $x[1 \ldots k]$ and $y[1 \ldots l]$, how do we efficiently merge them into a single sorted array $z[1 \ldots k+l]$? Well, the very first element of $z$ is either $x[1]$ or $y[1]$, whichever is smaller. The rest of $z[\cdot]$ can then be constructed recursively.

```
function merge(x[1...k], y[1...l])
if k = 0: return y[1...l]
if l = 0: return x[1...k]
if x[1] ≤ y[1]:
   return x[1] ∘ merge(x[2...k], y[1...l])
else:
   return y[1] ∘ merge(x[1...k], y[2...l])
```

Here $\circ$ denotes concatenation. This `merge` procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of $O(k+l)$. Thus `merge`'s are linear, and the overall time taken by `mergesort` is

$$T(n) = 2T(n/2) + O(n),$$

or $O(n \log n)$.

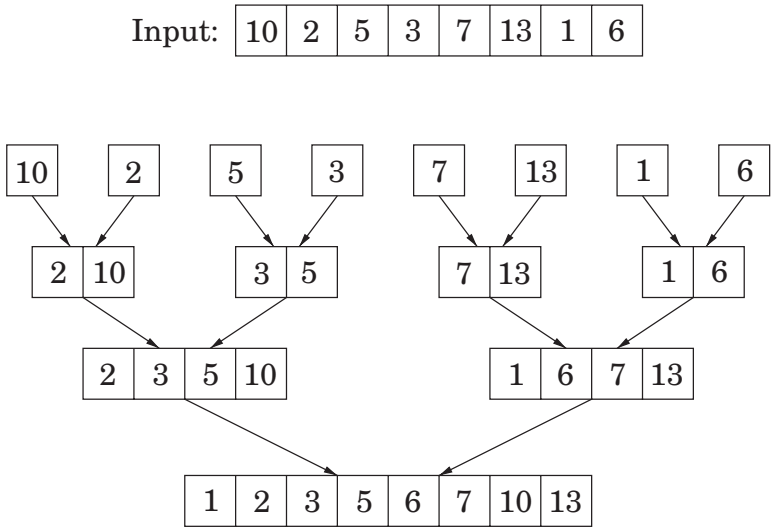
Looking back at the `mergesort` algorithm, we see that all the real work is done in merging, which doesn't start until the recursion gets down to singleton arrays. The singletons are merged in pairs, to yield arrays with two elements. Then pairs of these 2-tuples are merged, producing 4-tuples, and so on. Figure 2.4 shows an example.

This viewpoint also suggests how `mergesort` might be made iterative. At any given moment, there is a set of "active" arrays—initially, the singletons—which are merged in pairs to give the next batch of active arrays. These arrays can be organized in a queue, and processed by repeatedly removing two arrays from the front of the queue, merging them, and putting the result at the end of the queue.

In the following pseudocode, the primitive operation `inject` adds an element to the end of the queue while `eject` removes and returns the element at the front of the queue.

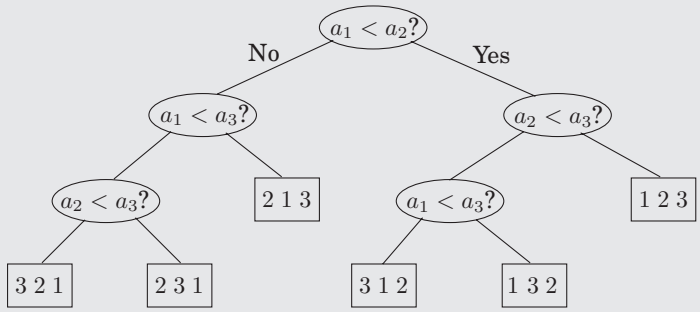
```
function iterative-mergesort(a[1...n])
Input: elements a_1, a_2, ..., a_n to be sorted

Q = [ ] (empty queue)
for i = 1 to n:
   inject(Q, [a_i])
while |Q| > 1:
   inject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
```

**Figure 2.4** The sequence of merge operations in `mergesort`.




## An $n \log n$ lower bound for sorting

Sorting algorithms can be depicted as trees. The one in the following figure sorts an array of three elements, $a_1, a_2, a_3$. It starts by comparing $a_1$ to $a_2$ and, if the first is larger, compares it with $a_3$; otherwise it compares $a_2$ and $a_3$. And so on. Eventually we end up at a leaf, and this leaf is labeled with the true order of the three elements as a permutation of 1, 2, 3. For example, if $a_2 < a_1 < a_3$, we get the leaf labeled "2 1 3."




The *depth* of the tree—the number of comparisons on the longest path from root to leaf, in this case 3—is exactly the worst-case time complexity of the algorithm.

### An $n \log n$ lower bound for sorting (*continued*)

This way of looking at sorting algorithms is useful because it allows one to argue that *mergesort is optimal*, in the sense that $\Omega(n \log n)$ comparisons are necessary for sorting $n$ elements.

Here is the argument: Consider any such tree that sorts an array of $n$ elements. Each of its leaves is labeled by a permutation of $\{1, 2, \ldots, n\}$. In fact, *every* permutation must appear as the label of a leaf. The reason is simple: if a particular permutation is missing, what happens if we feed the algorithm an input ordered according to this same permutation? And since there are $n!$ permutations of $n$ elements, it follows that the tree has at least $n!$ leaves.

We are almost done: This is a binary tree, and we argued that it has at least $n!$ leaves. Recall now that a binary tree of depth $d$ has at most $2^d$ leaves (proof: an easy induction on $d$). So, the depth of our tree—and the complexity of our algorithm—must be at least $\log(n!)$.

And it is well known that $\log(n!) \geq c \cdot n \log n$ for some $c > 0$. There are many ways to see this. The easiest is to notice that $n! \geq (n/2)^{(n/2)}$ because $n! = 1 \cdot 2 \cdot \cdots \cdot n$ contains at least $n/2$ factors larger than $n/2$; and to then take logs of both sides. Another is to recall Stirling's formula

$$n! \approx \sqrt{\pi \left(2n + \frac{1}{3}\right)} \cdot n^n \cdot e^{-n}.$$

Either way, we have established that any comparison tree that sorts $n$ elements must make, in the worst case, $\Omega(n \log n)$ comparisons, and hence mergesort is optimal!
Well, there is some fine print: this neat argument applies only to *algorithms that use comparisons*. Is it conceivable that there are alternative sorting strategies, perhaps using sophisticated numerical manipulations, that work in linear time? The answer is *yes*, under certain exceptional circumstances: the canonical such example is when the elements to be sorted are integers that lie in a small range (Exercise 2.20).

## 2.4 Medians

The *median* of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. For instance, the median of $[45, 1, 10, 30, 25]$ is 25, since this is the middle element when the numbers are arranged in order. If the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value. The *mean*, or average, is also very commonly used for this, but the median is in a sense more typical of the data: it is always one of the data values, unlike the mean, and it is less sensitive to outliers. For instance, the median of a list of a hundred 1's is (rightly) 1, as is the mean. However, if just one of these numbers gets accidentally corrupted to 10,000, the mean shoots up above 100, while the median is unaffected.

Computing the median of $n$ numbers is easy: just sort them. The drawback is that this takes $O(n \log n)$ time, whereas we would ideally like something linear. We have reason to be hopeful, because sorting is doing far more work than we really need—we just want the middle element and don't care about the relative ordering of the rest of them.

When looking for a recursive solution, it is paradoxically often easier to work with a *more general* version of the problem—for the simple reason that this gives a more powerful step to recurse upon. In our case, the generalization we will consider is *selection*.

**Selection**

*Input:* A list of numbers $S$; an integer $k$

*Output:* The $k$th smallest element of $S$

For instance, if $k = 1$, the minimum of $S$ is sought, whereas if $k = \lfloor |S|/2 \rfloor$, it is the median.

**A randomized divide-and-conquer algorithm for selection**

Here's a divide-and-conquer approach to selection. For any number $v$, imagine splitting list $S$ into three categories: elements smaller than $v$, those equal to $v$ (there might be duplicates), and those greater than $v$. Call these $S_L$, $S_v$, and $S_R$ respectively. For instance, if the array

$S$ : | 2 | 36 | 5 | 21 | 8 | 13 | 11 | 20 | 5 | 4 | 1 |

is split on $v = 5$, the three subarrays generated are

$S_L$ : | 2 | 4 | 1 |  $S_v$ : | 5 | 5 |  $S_R$ : | 36 | 21 | 8 | 13 | 11 | 20 | .

The search can instantly be narrowed down to one of these sublists. If we want, say, the *eighth*-smallest element of $S$, we know it must be the *third*-smallest element of $S_R$ since $|S_L| + |S_v| = 5$. That is, $\text{selection}(S, 8) = \text{selection}(S_R, 3)$. More generally, by checking $k$ against the sizes of the subarrays, we can quickly determine which of them holds the desired element:

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

The three sublists $S_L$, $S_v$, and $S_R$ can be computed from $S$ in linear time; in fact, this computation can even be done *in place*, that is, without allocating new memory (Exercise 2.15). We then recurse on the appropriate sublist. The effect of the split is thus to shrink the number of elements from $|S|$ to at most $\max\{|S_L|, |S_R|\}$.

Our divide-and-conquer algorithm for selection is now fully specified, except for the crucial detail of how to choose $v$. It should be picked quickly, and it should shrink the array substantially, the ideal situation being $|S_L|, |S_R| \approx \frac{1}{2}|S|$. If we could always guarantee this situation, we would get a running time of

$$T(n) = T(n/2) + O(n),$$

which is linear as desired. But this requires picking $v$ to be the median, which is our ultimate goal! Instead, we follow a much simpler alternative: *we pick $v$ randomly from $S$*.

**Efficiency analysis**

Naturally, the running time of our algorithm depends on the random choices of $v$. It is possible that due to persistent bad luck we keep picking $v$ to be the largest element of the array (or the smallest element), and thereby shrink the array by only one element each time. In the earlier example, we might first pick $v = 36$, then $v = 21$, and so on. This *worst-case* scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \cdots + \frac{n}{2} = \Theta(n^2)$$

operations (when computing the median), but it is extremely unlikely to occur. Equally unlikely is the *best* possible case we discussed before, in which each randomly chosen $v$ just happens to split the array perfectly in half, resulting in a running time of $O(n)$. Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the *average* running time lie? Fortunately, it lies very close to the best-case time.

To distinguish between lucky and unlucky choices of $v$, we will call $v$ *good* if it lies within the 25th to 75th percentile of the array that it is chosen from. We like these choices of $v$ because they ensure that the sublists $S_L$ and $S_R$ have size at most three-fourths that of $S$ (do you see why?), so that the array shrinks substantially. Fortunately, good $v$'s are abundant: half the elements of any list must fall between the 25th to 75th percentile!

Given that a randomly chosen $v$ has a 50% chance of being good, how many $v$'s do we need to pick on average before getting a good one? Here's a more familiar reformulation (see also Exercise 1.34):

**Lemma** *On average a fair coin needs to be tossed two times before a "heads" is seen.*

*Proof.* Let $E$ be the expected number of tosses before a heads is seen. We certainly need at least one toss, and if it's heads, we're done. If it's tails (which occurs with probability $1/2$), we need to repeat. Hence $E = 1 + \frac{1}{2}E$, which works out to $E = 2$. ∎

Therefore, after two split operations on average, the array will shrink to at most three-fourths of its size. Letting $T(n)$ be the *expected* running time on an array of size $n$, we get

$$T(n) \leq T(3n/4) + O(n).$$

This follows by taking expected values of both sides of the following statement:

Time taken on an array of size $n$

$\leq$ (time taken on an array of size $3n/4$) + (time to reduce array size to $\leq 3n/4$),

and, for the right-hand side, using the familiar property that *the expectation of the sum is the sum of the expectations*.

### The unix `Sort` command

Comparing the algorithms for sorting and median-finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. Mergesort splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call.

*Quicksort* is a sorting algorithm that splits the array in exactly the same way as the median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved (Exercise 2.24) that its average case is $O(n \log n)$; furthermore, empirically it outperforms other sorting algorithms. This has made quicksort a favorite in many applications—for instance, it is the basis of the code by which really enormous files are sorted.

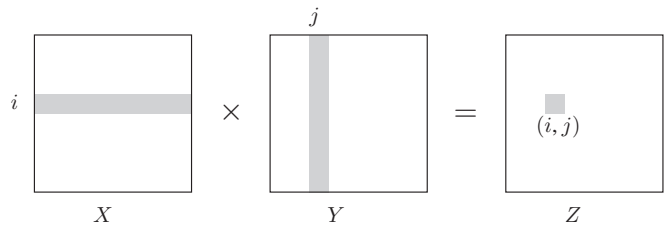
From this recurrence we conclude that $T(n) = O(n)$: on *any* input, our algorithm returns the correct answer after a linear number of steps, on the average.

## 2.5 Matrix multiplication

The product of two $n \times n$ matrices $X$ and $Y$ is a third $n \times n$ matrix $Z = XY$, with $(i, j)$th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

To make it more visual, $Z_{ij}$ is the dot product of the $i$th row of $X$ with the $j$th column of $Y$:




In general, $XY$ is not the same as $YX$; matrix multiplication is not commutative.

The preceding formula implies an $O(n^3)$ algorithm for matrix multiplication: there are $n^2$ entries to be computed, and each takes $O(n)$ time. For quite a while, this was widely believed to be the best running time possible, and it was even proved that in certain models of computation no algorithm could do better. It was therefore a source of great excitement when in 1969, the German mathematician Volker Strassen announced a significantly more efficient algorithm, based upon divide-and-conquer.

Matrix multiplication is particularly easy to break into subproblems, because it can be performed *blockwise*. To see what this means, carve $X$ into four $n/2 \times n/2$ blocks, and also $Y$:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Then their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements (Exercise 2.11).

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

We now have a divide-and-conquer strategy: to compute the size-$n$ product $XY$, recursively compute eight size-$n/2$ products $AE, BG, AF, BH, CE, DG, CF, DH$, and then do a few $O(n^2)$-time additions. The total running time is described by the recurrence relation

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

This comes out to an unimpressive $O(n^3)$, the same as for the default algorithm. But the efficiency *can* be further improved, and as with integer multiplication, the key is some clever algebra. It turns out $XY$ can be computed from just *seven* $n/2 \times n/2$ subproblems, via a decomposition so tricky and intricate that one wonders how Strassen was ever able to discover it!

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

The new running time is

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

which by the master theorem works out to $O(n^{\log_2 7}) \approx O(n^{2.81})$.

## 2.6 The fast Fourier transform

We have so far seen how divide-and-conquer gives fast algorithms for multiplying integers and matrices; our next target is *polynomials*. The product of two degree-$d$ polynomials is a polynomial of degree $2d$, for example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

More generally, if $A(x) = a_0 + a_1x + \cdots + a_dx^d$ and $B(x) = b_0 + b_1x + \cdots + b_dx^d$, their product $C(x) = A(x) \cdot B(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$ has coefficients

$$c_k = a_0b_k + a_1b_{k-1} + \cdots + a_kb_0 = \sum_{i=0}^{k} a_ib_{k-i}$$

(for $i > d$, take $a_i$ and $b_i$ to be zero). Computing $c_k$ from this formula takes $O(k)$ steps, and finding all $2d + 1$ coefficients would therefore seem to require $\Theta(d^2)$ time. *Can we possibly multiply polynomials faster than this?*

The solution we will develop, the fast Fourier transform, has revolutionized—indeed, defined—the field of signal processing (see the following box). Because of its huge importance, and its wealth of insights from different fields of study, we will approach it a little more leisurely than usual. The reader who wants just the core algorithm can skip directly to Section 2.6.4.

### 2.6.1 An alternative representation of polynomials

To arrive at a fast algorithm for polynomial multiplication we take inspiration from an important property of polynomials.

**Fact** *A degree-d polynomial is uniquely characterized by its values at any $d + 1$ distinct points.*

A familiar instance of this is that "any two points determine a line." We will later see why the more general statement is true (page 64), but for the time being it gives us an *alternative representation* of polynomials. Fix any distinct points $x_0, \ldots, x_d$. We can specify a degree-$d$ polynomial $A(x) = a_0 + a_1x + \cdots + a_dx^d$ by either one of the following:

1. Its coefficients $a_0, a_1, \ldots, a_d$
2. The values $A(x_0), A(x_1), \ldots, A(x_d)$

Of these two representations, the second is the more attractive for polynomial multiplication. Since the product $C(x)$ has degree $2d$, it is completely determined by its value at any $2d + 1$ points. And its value at any given point $z$ is easy enough to figure out, just $A(z)$ times $B(z)$. Thus *polynomial multiplication takes linear time in the value representation.*

The problem is that we expect the input polynomials, and also their product, to be specified by coefficients. So we need to first translate from coefficients to values—which is just a matter of *evaluating* the polynomial at the chosen points—then multiply in the value representation, and finally translate back to coefficients, a process called *interpolation*.

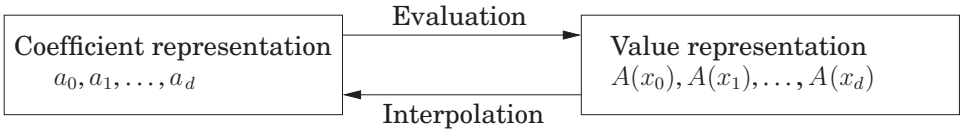



Figure 2.5 presents the resulting algorithm.

## Why multiply polynomials?

For one thing, it turns out that the fastest algorithms we have for multiplying integers rely heavily on polynomial multiplication; after all, polynomials and binary integers are quite similar—just replace the variable $x$ by the base 2, and watch out for carries. But perhaps more importantly, multiplying polynomials is crucial for *signal processing*.

A *signal* is any quantity that is a function of time (as in Figure (a)) or of position. It might, for instance, capture a human voice by measuring fluctuations in air pressure close to the speaker's mouth, or alternatively, the pattern of stars in the night sky, by measuring brightness as a function of angle.


(a) (b) (c)

In order to extract information from a signal, we need to first *digitize* it by sampling (Figure (b))—and, then, to put it through a *system* that will transform it in some way. The output is called the *response* of the system:

signal $\longrightarrow$ SYSTEM $\longrightarrow$ response.

An important class of systems are those that are *linear*—the response to the sum of two signals is just the sum of their individual responses—and *time invariant*—shifting the input signal by time $t$ produces the same output, also shifted by $t$. Any system with these properties is completely characterized by its response to the simplest possible input signal: the *unit impulse* $\delta(t)$, consisting solely of a "jerk" at $t = 0$ (Figure (c)). To see this, first consider the close relative $\delta(t - i)$, a shifted impulse in which the jerk occurs at time $i$. Any signal $a(t)$ can be expressed as a linear combination of these, letting $\delta(t - i)$ pick out its behavior at time $i$,

$$a(t) = \sum_{i=0}^{T-1} a(i)\delta(t - i)$$

(if the signal consists of $T$ samples). By linearity, the system response to input $a(t)$ is determined by the responses to the various $\delta(t - i)$. And by time invariance, these are in turn just shifted copies of the *impulse response* $b(t)$, the response to $\delta(t)$.

In other words, the output of the system at time $k$ is

$$c(k) = \sum_{i=0}^{k} a(i)b(k - i),$$

exactly the formula for polynomial multiplication!

**Figure 2.5** Polynomial multiplication

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree $d$
Output: Their product $C = A \cdot B$

**Selection**
Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d+1$
**Evaluation**
Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$
**Multiplication**
Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$
**Interpolation**
Recover $C(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$

The equivalence of the two polynomial representations makes it clear that this high-level approach is correct, but how efficient is it? Certainly the selection step and the $n$ multiplications are no trouble at all, just linear time.[3] But (leaving aside interpolation, about which we know even less) how about evaluation? Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$ steps (Exercise 2.29), and so the baseline for $n$ points is $\Theta(n^2)$. We'll now see that the fast Fourier transform (FFT) does it in just $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$ in which the computations required by the individual points overlap with one another and can be shared.

### 2.6.2 Evaluation by divide-and-conquer

Here's an idea for how to pick the $n$ points at which to evaluate a polynomial $A(x)$ of degree $\leq n-1$. If we choose them to be positive-negative pairs, that is,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even powers of $x_i$ coincide with those of $-x_i$.

To investigate this, we need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 \;=\; (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4).$$

Notice that the terms in parentheses are polynomials in $x^2$. More generally,

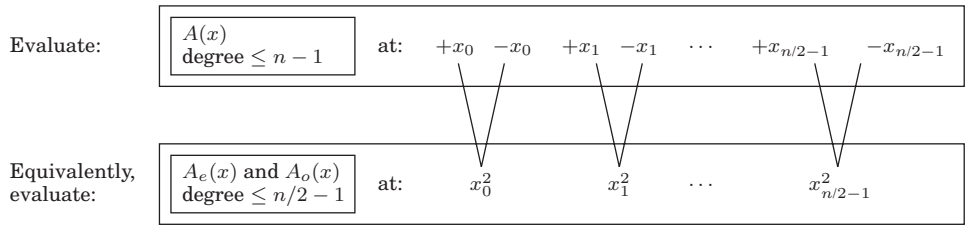
$$A(x) \;=\; A_e(x^2) + x A_o(x^2),$$

[3] In a typical setting for polynomial multiplication, the coefficients of the polynomials are real numbers and, moreover, are small enough that basic arithmetic operations (adding and multiplying) take unit time. We will assume this to be the case without any great loss of generality; in particular, the time bounds we obtain are easily adjustable to situations with larger numbers.

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that $n$ is even). Given *paired* points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2). \end{aligned}$$

In other words, evaluating $A(x)$ at $n$ paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ (which each have half the degree of $A(x)$) at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.




The original problem of size $n$ is in this way recast as two subproblems of size $n/2$, followed by some linear-time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

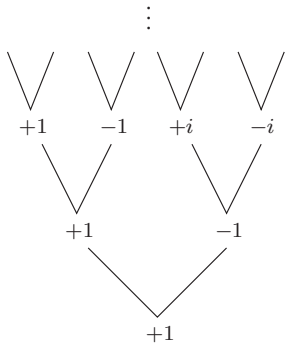
$$T(n) = 2T(n/2) + O(n),$$

which is $O(n \log n)$, exactly what we want.

But we have a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be *themselves* plus-minus pairs. But how can a square be negative? The task seems impossible! *Unless, of course, we use complex numbers*.

Fine, but which complex numbers? To figure this out, let us "reverse engineer" the process. At the very bottom of the recursion, we have a single point. This point might as well be 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

The next level up then has $\pm\sqrt{+1} = \pm 1$ as well as the *complex* numbers $\pm\sqrt{-1} = \pm i$, where $i$ is the imaginary unit. By continuing in this manner, we eventually reach the initial set of $n$ points. Perhaps you have already guessed what




they are: the *complex nth roots of unity*, that is, the $n$ complex solutions to the equation $z^n = 1$.

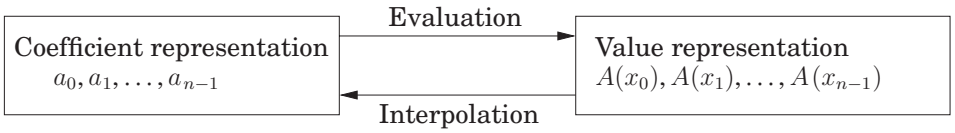
Figure 2.6 is a pictorial review of some basic facts about complex numbers. The third panel of this figure introduces the $n$th roots of unity: the complex numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n}$. If $n$ is even,

1. The $n$th roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$.
2. Squaring them produces the $(n/2)$nd roots of unity.

Therefore, if we start with these numbers for some $n$ that is a power of 2, then at successive levels of recursion we will have the $(n/2^k)$th roots of unity, for $k = 0, 1, 2, 3, \ldots$. All these sets of numbers are plus-minus paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 2.7).

### 2.6.3 Interpolation

Let's take stock of where we are. We first developed a high-level scheme for multiplying polynomials (Figure 2.5), based on the observation that polynomials can be represented in two ways, in terms of their *coefficients* or in terms of their *values* at a selected set of points.

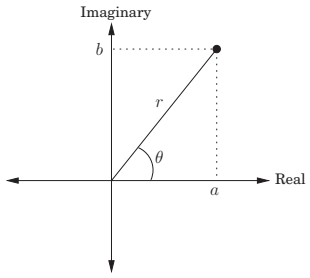



The value representation makes it trivial to multiply polynomials, but we cannot ignore the coefficient representation since it is the form in which the input and output of our overall algorithm are specified.

So we designed the FFT, a way to move from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex $n$th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega).$$

**Figure 2.6** The complex roots of unity are ideal for our divide-and-conquer scheme.




### The complex plane

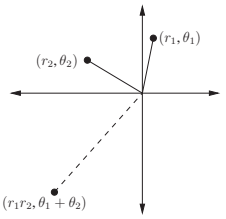
$z = a + bi$ is plotted at position $(a, b)$.

Polar coordinates: rewrite as $z = r(\cos\theta + i\sin\theta) = re^{i\theta}$, denoted $(r, \theta)$.

- *length* $r = \sqrt{a^2 + b^2}$.
- *angle* $\theta \in [0, 2\pi)$: $\cos\theta = a/r$, $\sin\theta = b/r$.
- $\theta$ can always be reduced modulo $2\pi$.

Examples:

| Number | $-1$ | $i$ | $5 + 5i$ |
|---|---|---|---|
| Polar coords | $(1, \pi)$ | $(1, \pi/2)$ | $(5\sqrt{2}, \pi/4)$ |

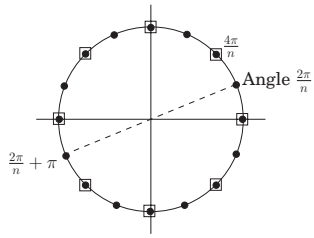



### Multiplying is easy in polar coordinates

Multiply the lengths and add the angles:
$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.

For any $z = (r, \theta)$,

- $-z = (r, \theta + \pi)$ since $-1 = (1, \pi)$.
- If $z$ is on the *unit circle* (i.e., $r = 1$), then $z^n = (1, n\theta)$.




### The $n$th complex roots of unity

Solutions to the equation $z^n = 1$.

By the multiplication rule: solutions are $z = (1, \theta)$, for $\theta$ a multiple of $2\pi/n$ (shown here for $n = 16$).

For even $n$:

- These numbers are *plus-minus paired*: $-(1, \theta) = (1, \theta + \pi)$.
- Their squares are the $(n/2)$nd roots of unity, shown here with boxes around them.

### Divide-and-conquer step

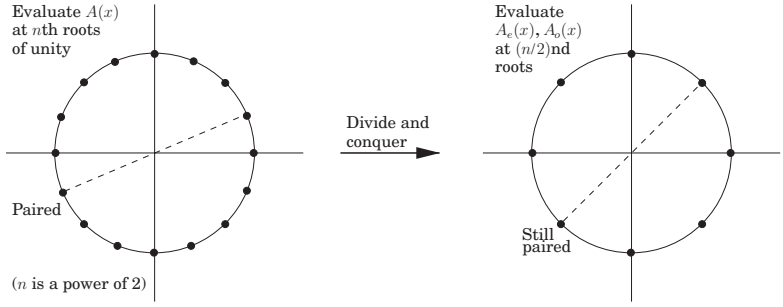


**Figure 2.7** The fast Fourier transform (polynomial formulation)

```
function FFT(A, ω)
Input: Coefficient representation of a polynomial A(x)
       of degree ≤ n−1, where n is a power of 2
       ω, an nth root of unity
Output: Value representation A(ω^0), ..., A(ω^(n−1))

if ω = 1: return A(1)
express A(x) in the form A_e(x^2) + x A_o(x^2)
call FFT(A_e, ω^2) to evaluate A_e at even powers of ω
call FFT(A_o, ω^2) to evaluate A_o at even powers of ω
for j = 0 to n−1:
   compute A(ω^j) = A_e(ω^(2j)) + ω^j A_o(ω^(2j))

return A(ω^0), ..., A(ω^(n−1))
```

The last remaining piece of the puzzle is the inverse operation, interpolation. It will turn out, amazingly, that

$$\langle \text{coefficients} \rangle = \frac{1}{n} \text{FFT}(\langle \text{values} \rangle, \omega^{-1}).$$

Interpolation is thus solved in the most simple and elegant way we could possibly have hoped for—using the same FFT algorithm, but called with $\omega^{-1}$ in place of $\omega$! This might seem like a miraculous coincidence, but it will make a lot more sense when we recast our polynomial operations in the language of linear algebra. Meanwhile, our $O(n \log n)$ polynomial multiplication algorithm (Figure 2.5) is now fully specified.

**A matrix reformulation**

To get a clearer view of interpolation, let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$. They are both vectors of $n$ numbers, and one is a linear transformation of the other:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

Call the matrix in the middle $M$. Its specialized format—a *Vandermonde* matrix—gives it many remarkable properties, of which the following is particularly relevant to us.

*If $x_0, \ldots, x_{n-1}$ are distinct numbers, then $M$ is invertible.*

The existence of $M^{-1}$ allows us to invert the preceding matrix equation so as to express coefficients in terms of values. In brief,

*Evaluation is multiplication by $M$, while interpolation is multiplication by $M^{-1}$.*

This reformulation of our polynomial operations reveals their essential nature more clearly. Among other things, it finally justifies an assumption we have been making throughout, that $A(x)$ is uniquely characterized by its values at any $n$ points—in fact, we now have an explicit formula that will give us the coefficients of $A(x)$ in this situation. Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$. However, using this for interpolation would still not be fast enough for us, so once again we turn to our special choice of points—the complex roots of unity.

**Interpolation resolved**

In linear algebra terms, the FFT multiplies an arbitrary $n$-dimensional vector—which we have been calling the *coefficient representation*—by the $n \times n$ matrix

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{array}{l} \\ \\ \\ \\ \longleftarrow \\ \\ \longleftarrow \end{array} \begin{array}{l} \\ \\ \\ \\ \text{row for } \omega^j \\ \\ \omega^{n-1} \end{array}$$

where $\omega$ is a complex $n$th root of unity, and $n$ is a power of 2. Notice how simple this matrix is to describe: its $(j, k)$th entry (starting row- and column-count at zero) is $\omega^{jk}$.

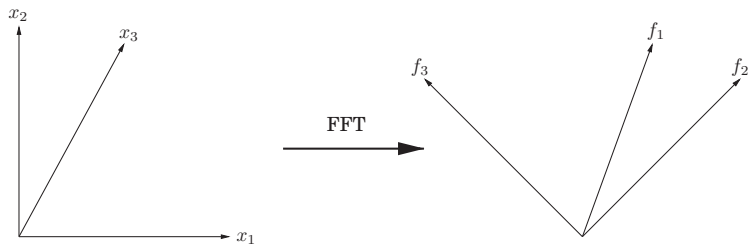
Multiplication by $M = M_n(\omega)$ maps the $k$th coordinate axis (the vector with all zeros except for a 1 at position $k$) onto the $k$th column of $M$. Now here's the crucial observation, which we'll prove shortly: *the columns of $M$ are orthogonal (at right angles) to each other*. Therefore they can be thought of as the axes of an alternative coordinate system, which is often called the *Fourier basis*. The effect of multiplying a vector by $M$ is to rotate it from the standard basis, with the usual set of axes, into the Fourier basis, which is defined by the columns of $M$ (Figure 2.8). The FFT is thus a change of basis, a *rigid rotation*. The inverse of $M$ is the opposite rotation, from the Fourier basis back into the standard basis. When we write out the orthogonality condition precisely, we will be able to read off this inverse transformation with ease:

**Inversion formula** $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

But $\omega^{-1}$ is also an $n$th root of unity, and so interpolation—or equivalently, multiplication by $M_n(\omega)^{-1}$—is itself just an FFT operation, but with $\omega$ replaced by $\omega^{-1}$.

Now let's get into the details. Take $\omega$ to be $e^{2\pi i/n}$ for convenience, and think of the columns of $M$ as vectors in $\mathbb{C}^n$. Recall that the *angle* between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v = (v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their

**Figure 2.8** The FFT takes points in the standard coordinate system, whose axes are shown here as $x_1, x_2, x_3$, and rotates them into the Fourier basis, whose axes are the columns of $M_n(\omega)$, shown here as $f_1$, $f_2$, $f_3$. For instance, points in direction $x_1$ get mapped into direction $f_1$.




*inner product*

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \cdots + u_{n-1} v_{n-1}^*,$$

where $z^*$ denotes the complex conjugate[4] of $z$. This quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

The fundamental observation we need is the following.

**Lemma** *The columns of matrix $M$ are orthogonal to each other.*

*Proof.* Take the inner product of any columns $j$ and $k$ of matrix $M$,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}.$$

This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio $\omega^{(j-k)}$. Therefore it evaluates to $(1 - \omega^{n(j-k)})/(1 - \omega^{(j-k)})$, which is 0—except when $j = k$, in which case all terms are 1 and the sum is $n$. ∎

The orthogonality property can be summarized in the single equation

$$MM^* = nI,$$

since $(MM^*)_{ij}$ is the inner product of the $i$th and $j$th columns of $M$ (do you see why?). This immediately implies $M^{-1} = (1/n)M^*$: we have an inversion formula! But is it the same formula we earlier claimed? Let's see—the $(j, k)$th entry of $M^*$ is the complex conjugate of the corresponding entry of $M$, in other words $\omega^{-jk}$. Whereupon $M^* = M_n(\omega^{-1})$, and we're done.

And now we can finally step back and view the whole affair geometrically. The task we need to perform, polynomial multiplication, is a lot easier in the Fourier

[4]The *complex conjugate* of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$. The complex conjugate of a vector (or matrix) is obtained by taking the complex conjugates of all its entries.

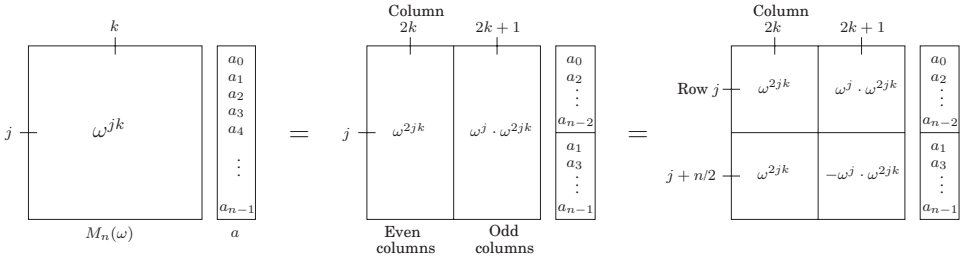
basis than in the standard basis. Therefore, we first rotate vectors into the Fourier basis (*evaluation*), then perform the task (*multiplication*), and finally rotate back (*interpolation*). The initial vectors are *coefficient representations*, while their rotated counterparts are *value representations*. To efficiently switch between these, back and forth, is the province of the FFT.

### 2.6.4 A closer look at the fast Fourier transform

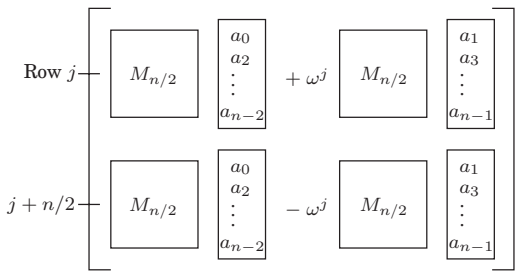
Now that our efficient scheme for polynomial multiplication is fully realized, let's hone in more closely on the core subroutine that makes it all possible, the fast Fourier transform.

**The definitive FFT algorithm**

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number $\omega$ whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex $n$th roots of unity. It multiplies vector $a$ by the $n \times n$ matrix $M_n(\omega)$, which has $(j, k)$th entry (starting row- and column-count at zero) $\omega^{jk}$. The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when $M$'s columns are segregated into evens and odds:




In the second step, we have simplified entries in the bottom half of the matrix using $\omega^{n/2} = -1$ and $\omega^n = 1$. Notice that the top left $n/2 \times n/2$ submatrix is $M_{n/2}(\omega^2)$, as is the one on the bottom left. And the top and bottom right submatrices are almost the same as $M_{n/2}(\omega^2)$, but with their $j$th rows multiplied through by $\omega^j$ and $-\omega^j$, respectively. Therefore the final product is the vector.




In short, the product of $M_n(\omega)$ with vector $(a_0, \ldots, a_{n-1})$, a size-$n$ problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$. This divide-and-conquer strategy

leads to the definitive FFT algorithm of Figure 2.9, whose running time is $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

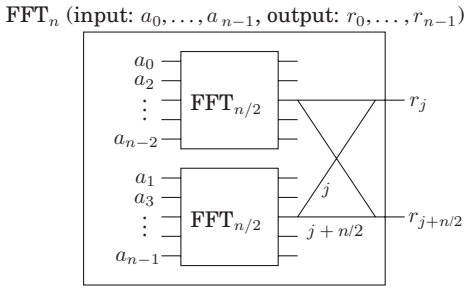
**Figure 2.9** The fast Fourier transform

```
function FFT(a, ω)
Input: An array a = (a_0, a_1, ..., a_{n-1}), for n a power of 2
   A primitive nth root of unity, ω
Output: M_n(ω) a

if ω = 1: return a
(s_0, s_1, ..., s_{n/2-1}) = FFT((a_0, a_2, ..., a_{n-2}), ω^2)
(s'_0, s'_1, ..., s'_{n/2-1}) = FFT((a_1, a_3, ..., a_{n-1}), ω^2)
for j = 0 to n/2 - 1:
    r_j = s_j + ω^j s'_j
    r_{j+n/2} = s_j - ω^j s'_j
return (r_0, r_1, ..., r_{n-1})
```

**The fast Fourier transform unraveled**

Throughout all our discussions so far, the fast Fourier transform has remained tightly cocooned within a divide-and-conquer formalism. To fully expose its structure, we now unravel the recursion.

The divide-and-conquer step of the FFT can be drawn as a very simple circuit. Here is how a problem of size $n$ is reduced to two subproblems of size $n/2$ (for clarity, one pair of outputs $(j, j + n/2)$ is singled out):




We're using a particular shorthand: the edges are wires carrying complex numbers from left to right. A weight of $j$ means "multiply the number on this wire by $\omega^j$." And when two wires come into a junction from the left, the numbers they are carrying get added up. So the two outputs depicted are executing the commands

$$r_j = s_j + \omega^j s'_j$$
$$r_{j+n/2} = s_j - \omega^j s'_j$$

from the FFT algorithm (Figure 2.9), via a pattern of wires known as a *butterfly*: ⧖.

Unraveling the FFT circuit completely for $n = 8$ elements, we get Figure 2.10. Notice the following.

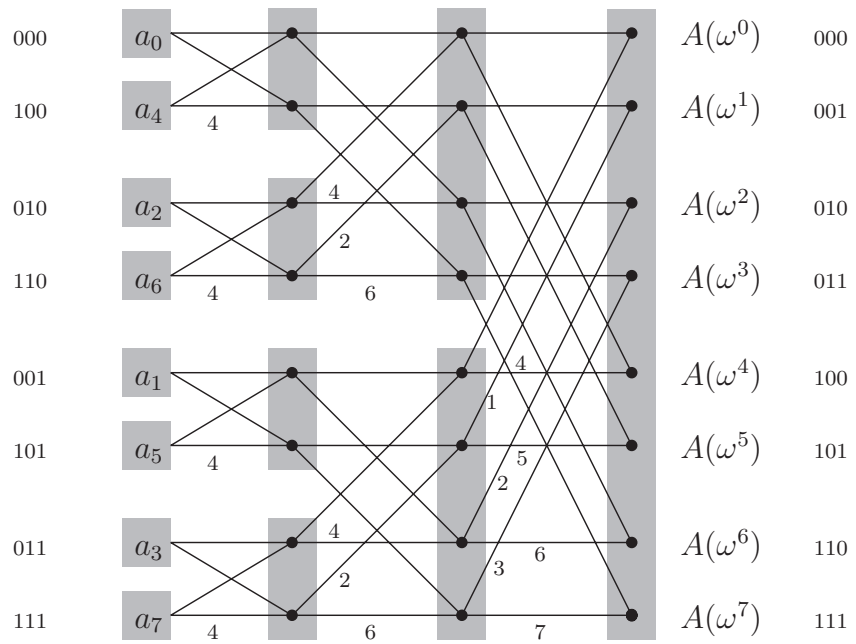
1. For $n$ inputs there are $\log_2 n$ levels, each with $n$ nodes, for a total of $n \log n$ operations.
2. The inputs are arranged in a peculiar order: 0, 4, 2, 6, 1, 5, 3, 7.

Why? Recall that at the top level of recursion, we first bring up the even coefficients of the input and then move on to the odd ones. Then at the next level, the even coefficients of this first group (which therefore are multiples of 4, or equivalently, have zero as their two least significant bits) are brought up, and so on. To put it otherwise, the inputs are arranged by increasing *last* bit of the binary representation of their index, resolving ties by looking at the next more significant bit(s). The resulting order in binary, 000, 100, 010, 110, 001, 101, 011, 111, is the same as the natural one, 000, 001, 010, 011, 100, 101, 110, 111 *except the bits are mirrored!*

3. There is a unique path between each input $a_j$ and each output $A(\omega^k)$.

This path is most easily described using the binary representations of $j$ and $k$ (shown in Figure 2.10). There are two edges out of each node, one going up (the 0-edge) and

**Figure 2.10** The fast Fourier transform circuit.



one going down (the 1-edge). To get to $A(\omega^k)$ from any input node, simply follow the edges specified in the bit representation of $k$, starting from the rightmost bit. (Can you similarly specify the path in the reverse direction?)

4. On the path between $a_j$ and $A(\omega^k)$, the labels add up to $jk \bmod 8$.

Since $\omega^8 = 1$, this means that the contribution of input $a_j$ to output $A(\omega^k)$ is $a_j\omega^{jk}$, and therefore the circuit computes correctly the values of polynomial $A(x)$.

5. And finally, notice that the FFT circuit is a natural for parallel computation and direct implementation in hardware.

### The slow spread of a fast algorithm

In 1963, during a meeting of President Kennedy's scientific advisors, John Tukey, a mathematician from Princeton, explained to IBM's Dick Garwin a fast method for computing Fourier transforms. Garwin listened carefully, because he was at the time working on ways to detect nuclear explosions from seismographic data, and Fourier transforms were the bottleneck of his method. When he went back to IBM, he asked John Cooley to implement Tukey's algorithm; they decided that a paper should be published so that the idea could not be patented.

Tukey was not very keen to write a paper on the subject, so Cooley took the initiative. And this is how one of the most famous and most cited scientific papers was published in 1965, co-authored by Cooley and Tukey. The reason Tukey was reluctant to publish the FFT was not secretiveness or pursuit of profit via patents. He just felt that this was a simple observation that was probably already known. This was typical of the period: back then (and for some time later) algorithms were considered second-class mathematical objects, devoid of depth and elegance, and unworthy of serious attention.

But Tukey was right about one thing: it was later discovered that British engineers had used the FFT for hand calculations during the late 1930s. And—to end this chapter with the same great mathematician who started it—a paper by Gauss in the early 1800s on (what else?) interpolation contained essentially the same idea in it! Gauss's paper had remained a secret for so long because it was protected by an old-fashioned cryptographic technique: like most scientific papers of its era, it was written in Latin.

## Exercises

2.1. Use the divide-and-conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

2.2. Show that for any positive integers $n$ and any base $b$, there must be some power of $b$ lying in the range $[n, bn]$.

2.3. Section 2.2 describes a method for solving recurrence relations which is based on analyzing the recursion tree and deriving a formula for the work done at each level. Another (closely related) method is to expand out the recurrence a few times, until a pattern emerges. For instance, let's start with the familiar $T(n) = 2T(n/2) + O(n)$. Think of $O(n)$ as being $\leq cn$ for some constant $c$, so: $T(n) \leq 2T(n/2) + cn$. By repeatedly applying this rule, we can bound $T(n)$ in terms of $T(n/2)$, then $T(n/4)$, then $T(n/8)$, and so on, at each step getting closer to the value of $T(\cdot)$ we do know, namely $T(1) = O(1)$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2[2T(n/4) + cn/2] + cn = 4T(n/4) + 2cn \\ &\leq 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn \\ &\leq 8[2T(n/16) + cn/8] + 3cn = 16T(n/16) + 4cn \\ &\vdots \end{aligned}$$

A pattern is emerging... the general term is

$$T(n) \leq 2^k T(n/2^k) + kcn.$$

Plugging in $k = \log_2 n$, we get $T(n) \leq nT(1) + cn\log_2 n = O(n \log n)$.

(a) Do the same thing for the recurrence $T(n) = 3T(n/2) + O(n)$. What is the general $k$th term in this case? And what value of $k$ should be plugged in to get the answer?

(b) Now try the recurrence $T(n) = T(n-1) + O(1)$, a case which is not covered by the master theorem. Can you solve this too?

2.4. Suppose you are choosing between the following three algorithms:

- Algorithm $A$ solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm $B$ solves problems of size $n$ by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
- Algorithm $C$ solves problems of size $n$ by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-$O$ notation), and which would you choose?

2.5. Solve the following recurrence relations and give a $\Theta$ bound for each of them.

(a) $T(n) = 2T(n/3) + 1$

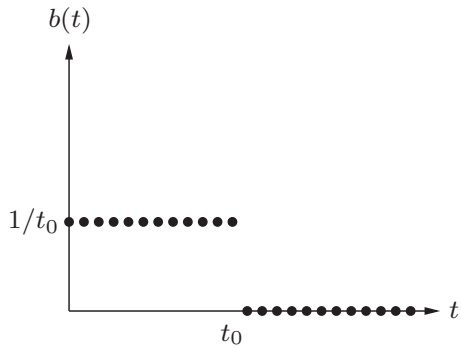
(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 8T(n/2) + n^3$
(f) $T(n) = 49T(n/25) + n^{3/2} \log n$
(g) $T(n) = T(n-1) + 2$
(h) $T(n) = T(n-1) + n^c$, where $c \geq 1$ is a constant
(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant
(j) $T(n) = 2T(n-1) + 1$
(k) $T(n) = T(\sqrt{n}) + 1$

2.6. A linear, time-invariant system has the following impulse response:




(a) Describe in words the effect of this system.
(b) What is the corresponding polynomial?

2.7. What is the sum of the $n$th roots of unity? What is their product if $n$ is odd? If $n$ is even?

2.8. Practice with the fast Fourier transform.

(a) What is the FFT of $(1, 0, 0, 0)$? What is the appropriate value of $\omega$ in this case? And of which sequence is $(1, 0, 0, 0)$ the FFT?
(b) Repeat for $(1, 0, 1, -1)$.

2.9. Practice with polynomial multiplication by FFT.

(a) Suppose that you want to multiply the two polynomials $x + 1$ and $x^2 + 1$ using the FFT. Choose an appropriate power of two, find the FFT of the two sequences, multiply the results componentwise, and compute the inverse FFT to get the final result.
(b) Repeat for the pair of polynomials $1 + x + 2x^2$ and $2 + 3x$.

2.10. Find the unique polynomial of degree 4 that takes on values $p(1) = 2$, $p(2) = 1$, $p(3) = 0$, $p(4) = 4$, and $p(5) = 0$. Write your answer in the coefficient representation.

2.11. In justifying our matrix multiplication algorithm (Section 2.5), we claimed the following blockwise property: if $X$ and $Y$ are $n \times n$ matrices, and

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

where $A$, $B$, $C$, $D$, $E$, $F$, $G$, and $H$ are $n/2 \times n/2$ submatrices, then the product $XY$ can be expressed in terms of these blocks:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Prove this property.

2.12. How many lines, as a function of $n$ (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume $n$ is a power of 2.

```
function f(n)
  if n > 1:
    print_line("still going")
    f(n/2)
    f(n/2)
```

2.13. A binary tree is *full* if all of its vertices have either zero or two children. Let $B_n$ denote the number of full binary trees with $n$ vertices.

(a) By drawing out all full binary trees with 3, 5, or 7 vertices, determine the exact values of $B_3$, $B_5$, and $B_7$. Why have we left out even numbers of vertices, like $B_4$?

(b) For general $n$, derive a recurrence relation for $B_n$.

(c) Show by induction that $B_n$ is $2^{\Omega(n)}$.

2.14. You are given an array of $n$ elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

2.15. In our median-finding algorithm (Section 2.4), a basic primitive is the `split` operation, which takes as input an array $S$ and a value $v$ and then divides $S$ into three sets: the elements less than $v$, the elements equal to $v$, and the elements greater than $v$. Show how to implement this `split` operation *in place*, that is, without allocating new memory.

2.16. You are given an infinite array $A[\cdot]$ in which the first $n$ cells contain integers in sorted order and the rest of the cells are filled with $\infty$. You are *not* given the value of $n$. Describe an algorithm that takes an integer $x$ as input and finds a position in the array containing $x$, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array $A$ has infinite length, assume instead that it is of length $n$, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message $\infty$ whenever elements $A[i]$ with $i > n$ are accessed.)

2.17. Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index $i$ for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

2.18. Consider the task of searching a sorted array $A[1 \ldots n]$ for a given element $x$: a task we usually perform by binary search in time $O(\log n)$. Show that any algorithm that accesses the array only via comparisons (that is, by asking questions of the form "is $A[i] \leq z$?"), must take $\Omega(\log n)$ steps.

2.19. *A k-way merge operation.* Suppose you have $k$ sorted arrays, each with $n$ elements, and you want to combine them into a single sorted array of $kn$ elements.

(a) Here's one strategy: Using the `merge` procedure from Section 2.3, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of $k$ and $n$?

(b) Give a more efficient solution to this problem, using divide-and-conquer.

2.20. Show that any array of integers $x[1 \ldots n]$ can be sorted in $O(n + M)$ time, where

$$M = \max_i x_i - \min_i x_i.$$

For small $M$, this is linear time: why doesn't the $\Omega(n \log n)$ lower bound apply in this case?

2.21. *Mean and median.* One of the most basic tasks in statistics is to summarize a set of observations $\{x_1, x_2, \ldots, x_n\} \subseteq \mathbb{R}$ by a single number. Two popular choices for this summary statistic are:

- The median, which we'll call $\mu_1$
- The mean, which we'll call $\mu_2$

(a) Show that the median is the value of $\mu$ that minimizes the function

$$\sum_i |x_i - \mu|.$$

You can assume for simplicity that $n$ is odd. (*Hint:* Show that for any $\mu \neq \mu_1$, the function decreases if you move $\mu$ either slightly to the left or slightly to the right.)

(b) Show that the mean is the value of $\mu$ that minimizes the function

$$\sum_i (x_i - \mu)^2.$$

One way to do this is by calculus. Another method is to prove that for any $\mu \in \mathbb{R}$,

$$\sum_i (x_i - \mu)^2 = \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2.$$

Notice how the function for $\mu_2$ penalizes points that are far from $\mu$ much more heavily than the function for $\mu_1$. Thus $\mu_2$ tries much harder to be close to *all* the observations. This might sound like a good thing at some level, but it is statistically undesirable because just a few outliers can severely throw off the estimate of $\mu_2$. It is therefore sometimes said that $\mu_1$ is a more robust estimator than $\mu_2$. Worse than either of them, however, is $\mu_\infty$, the value of $\mu$ that minimizes the
function

$$\max_i |x_i - \mu|.$$

(c) Show that $\mu_\infty$ can be computed in $O(n)$ time (assuming the numbers $x_i$ are small enough that basic arithmetic operations on them take unit time).

2.22. You are given two sorted lists of size $m$ and $n$. Give an $O(\log m + \log n)$ time algorithm for computing the $k$th smallest element in the union of the two lists.

2.23. An array $A[1 \ldots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time.

(a) Show how to solve this problem in $O(n \log n)$ time. (*Hint:* Split the array $A$ into two arrays $A_1$ and $A_2$ of half the size. Does knowing the majority elements of $A_1$ and $A_2$ help you figure out the majority element of $A$? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (*Hint:* Here's another divide-and-conquer approach:
- Pair up the elements of $A$ arbitrarily, to get $n/2$ pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if $A$ does.)

2.24. On page 56 there is a high-level description of the quicksort algorithm.

(a) Write down the pseudocode for quicksort.

(b) Show that its *worst-case* running time on an array of size $n$ is $\Theta(n^2)$.

(c) Show that its *expected* running time satisfies the recurrence relation

$$T(n) \leq O(n) + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i)).$$

Then, show that the solution to this recurrence is $O(n \log n)$.

2.25. In Section 2.1 we described an algorithm that multiplies two $n$-bit binary integers $x$ and $y$ in time $n^a$, where $a = \log_2 3$. Call this procedure `fastmultiply` $(x, y)$.

(a) We want to convert the decimal integer $10^n$ (a 1 followed by $n$ zeros) into binary. Here is the algorithm (assume $n$ is a power of 2):

```
function pwr2bin(n)
  if n = 1: return 1010₂
  else:
    z = ???
    return fastmultiply(z, z)
```

Fill in the missing details. Then give a recurrence relation for the running time of the algorithm, and solve the recurrence.

(b) Next, we want to convert any decimal integer $x$ with $n$ digits (where $n$ is a power of 2) into binary. The algorithm is the following:

```
function dec2bin(x)
  if n = 1: return binary[x]
  else:
    split x into two decimal numbers x_L, x_R with n/2
      digits each
    return ???
```

Here `binary`$[\cdot]$ is a vector that contains the binary representation of all one-digit integers. That is, `binary`$[0] = 0_2$, `binary`$[1] = 1_2$, up to `binary`$[9] = 1001_2$. Assume that a lookup in `binary` takes $O(1)$ time. Fill in the missing details. Once again, give a recurrence for the running time of the algorithm, and solve it.

2.26. Professor F. Lake tells his class that it is asymptotically faster to square an $n$-bit integer than to multiply two $n$-bit integers. Should they believe him?

2.27. The *square* of a matrix $A$ is its product with itself, $AA$.

(a) Show that five multiplications are sufficient to compute the square of a $2 \times 2$ matrix.

(b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

> "Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time $O(n^{\log_2 5})$."

(c) In fact, squaring matrices is no easier than matrix multiplication. Show that if $n \times n$ matrices can be squared in time $O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.

2.28. The *Hadamard matrices* $H_0, H_1, H_2, \ldots$ are defined as follows:

- $H_0$ is the $1 \times 1$ matrix $\begin{bmatrix} 1 \end{bmatrix}$

- For $k > 0$, $H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right].$$

Show that if $v$ is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

2.29. Suppose we want to evaluate the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ at point $x$.

(a) Show that the following simple routine, known as *Horner's rule*, does the job and leaves the answer in $z$.

$z = a_n$
`for` $i = n-1$ `downto 0:`
    $z = zx + a_i$

(b) How many additions and multiplications does this routine use, as a function of $n$? Can you find a polynomial for which an alternative method is substantially better?

2.30. This problem illustrates how to do the Fourier Transform (FT) in modular arithmetic, for example, modulo 7.

(a) There is a number $\omega$ such that all the powers $\omega, \omega^2, \ldots, \omega^6$ are distinct (modulo 7). Find this $\omega$, and show that $\omega + \omega^2 + \cdots + \omega^6 = 0$. (Interestingly, for any prime modulus there is such a number.)

(b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 1, 1, 5, 2)$ modulo 7; that is, multiply this vector by the matrix $M_6(\omega)$, for the value of $\omega$ you found earlier. In the matrix multiplication, all calculations should be performed modulo 7.

(c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 7.)

(d) Now show how to multiply the polynomials $x^2 + x + 1$ and $x^3 + 2x - 1$ using the FT modulo 7.

2.31. In Section 1.2.3, we studied Euclid's algorithm for computing the *greatest common divisor* (gcd) of two positive integers: the largest integer which divides them both. Here we will look at an alternative algorithm based on divide-and-conquer.

(a) Show that the following rule is true.

$$\gcd(a, b) = \begin{cases} 2\gcd(a/2, b/2) & \text{if } a, b \text{ are even} \\ \gcd(a, b/2) & \text{if } a \text{ is odd, } b \text{ is even} \\ \gcd((a-b)/2, b) & \text{if } a, b \text{ are odd} \end{cases}$$

(b) Give an efficient divide-and-conquer algorithm for greatest common divisor.

(c) How does the efficiency of your algorithm compare to Euclid's algorithm if $a$ and $b$ are $n$-bit integers? (In particular, since $n$ might be large you cannot assume that basic arithmetic operations like addition take constant time.)

2.32. In this problem we will develop a divide-and-conquer algorithm for the following geometric task.

CLOSEST PAIR
*Input:* A set of points in the plane, $\{p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)\}$
*Output:* The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between $p_i$ and $p_j$, that is,

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

is minimized.

For simplicity, assume that $n$ is a power of two, and that all the $x$-coordinates $x_i$ are distinct, as are the $y$-coordinates.

Here's a high-level overview of the algorithm:

- Find a value $x$ for which exactly half the points have $x_i < x$, and half have $x_i > x$. On this basis, split the points into two groups, $L$ and $R$.
- Recursively find the closest pair in $L$ and in $R$. Say these pairs are $p_L, q_L \in L$ and $p_R, q_R \in R$, with distances $d_L$ and $d_R$ respectively. Let $d$ be the smaller of these two distances.
- It remains to be seen whether there is a point in $L$ and a point in $R$ that are less than distance $d$ apart from each other. To this end, discard all points with $x_i < x - d$ or $x_i > x + d$ and sort the remaining points by $y$-coordinate.
- Now, go through this sorted list, and for each point, compute its distance to the *seven* subsequent points in the list. Let $p_M, q_M$ be the closest pair found in this way.
- The answer is one of the three pairs $\{p_L, q_L\}$, $\{p_R, q_R\}$, $\{p_M, q_M\}$, whichever is closest.

(a) In order to prove the correctness of this algorithm, start by showing the following property: any square of size $d \times d$ in the plane contains at most four points of $L$.

(b) Now show that the algorithm is correct. The only case which needs careful consideration is when the closest pair is split between $L$ and $R$.

(c) Write down the pseudocode for the algorithm, and show that its running time is given by the recurrence:

$$T(n) = 2T(n/2) + O(n \log n).$$

Show that the solution to this recurrence is $O(n \log^2 n)$.

(d) Can you bring the running time down to $O(n \log n)$?

2.33. Suppose you are given $n \times n$ matrices $A, B, C$ and you wish to check whether $AB = C$. You can do this in $O(n^{\log_2 7})$ steps using Strassen's algorithm. In this question we will explore a much faster, $O(n^2)$ randomized test.

(a) Let $\mathbf{v}$ be an $n$-dimensional vector whose entries are randomly and independently chosen to be 0 or 1 (each with probablity 1/2). Prove that if $M$ is a non-zero $n \times n$ matrix, then $\Pr[M\mathbf{v} = 0] \leq 1/2$.

(b) Show that $\Pr[AB\mathbf{v} = C\mathbf{v}] \leq 1/2$ if $AB \neq C$. Why does this give an $O(n^2)$ randomized test for checking whether $AB = C$?

2.34. *Linear 3SAT*. The 3SAT problem is defined in Section 8.1. Briefly, the input is a Boolean formula—expressed as a set of clauses—over some set of variables and the goal is to determine whether there is an assignment (of `true/false` values) to these variables that makes the entire formula evaluate to `true`.

Consider a 3SAT instance with the following special locality property. Suppose there are $n$ variables in the Boolean formula, and that they are numbered $1, 2, \ldots, n$ in such a way that each clause involves variables whose numbers are within $\pm 10$ of each other. Give a linear-time algorithm for solving such an instance of 3SAT.

*Chapter 3*

# Decompositions of graphs

## 3.1 Why graphs?

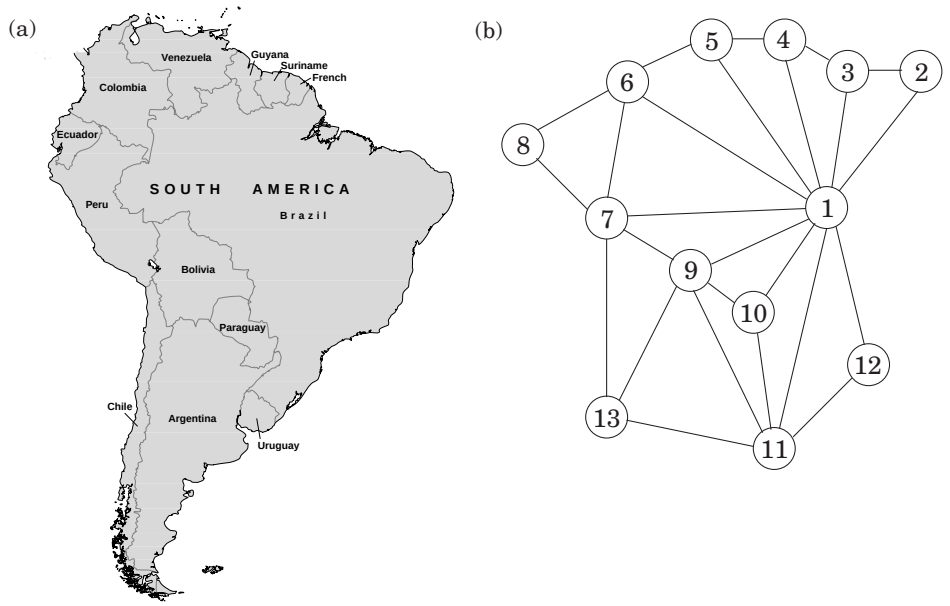
A wide range of problems can be expressed with clarity and precision in the concise pictorial language of graphs. For instance, consider the task of coloring a political map. What is the minimum number of colors needed, with the obvious restriction that neighboring countries should have different colors? One of the difficulties in attacking this problem is that the map itself, even a stripped-down version like Figure 3.1(a), is usually cluttered with irrelevant information: intricate boundaries, border posts where three or more countries meet, open seas, and meandering rivers. Such distractions are absent from the mathematical object of Figure 3.1(b), a graph with one *vertex* for each country (1 is Brazil, 11 is Argentina) and *edges* between neighbors. It contains exactly the information needed for coloring, and nothing more. The precise goal is now to assign a color to each vertex so that no edge has endpoints of the same color.

Graph coloring is not the exclusive domain of map designers. Suppose a university needs to schedule examinations for all its classes and wants to use the fewest time slots possible. The only constraint is that two exams cannot be scheduled concurrently if some student will be taking both of them. To express this problem as a graph, use one vertex for each exam and put an edge between two vertices if there is a conflict, that is, if there is somebody taking both endpoint exams. Think of each time slot as having its own color. Then, assigning time slots is exactly the same as coloring this graph!

Some basic operations on graphs arise with such frequency, and in such a diversity of contexts, that a lot of effort has gone into finding efficient procedures for them. This chapter is devoted to some of the most fundamental of these algorithms—those that uncover the basic connectivity structure of a graph.

Formally, a graph is specified by a set of vertices (also called *nodes*) $V$ and by edges $E$ between select pairs of vertices. In the map example, $V = \{1, 2, 3, \ldots, 13\}$ and $E$ includes, among many other edges, $\{1, 2\}$, $\{9, 11\}$, and $\{7, 13\}$. Here an edge between $x$ and $y$ specifically means "$x$ shares a border with $y$." This is a symmetric

**Figure 3.1** (a) A map and (b) its graph.




relation—it implies also that $y$ shares a border with $x$—and we denote it using set notation, $e = \{x, y\}$. Such edges are *undirected* and are part of an *undirected graph*.

Sometimes graphs depict relations that do not have this reciprocity, in which case it is necessary to use edges with directions on them. There can be *directed edges e from x to y* (written $e = (x, y)$), or from $y$ to $x$ (written $(y, x)$), or both. A particularly enormous example of a *directed graph* is the graph of all links in the World Wide Web. It has a vertex for each site on the Internet, and a directed edge $(u, v)$ whenever site $u$ has a link to site $v$: in total, billions of nodes and edges! Understanding even the most basic connectivity properties of the Web is of great economic and social interest. Although the size of this problem is daunting, we will soon see that a lot of valuable information about the structure of a graph can, happily, be determined in just linear time.

### 3.1.1 How is a graph represented?

We can represent a graph by an *adjacency matrix*; if there are $n = |V|$ vertices $v_1, \ldots, v_n$, this is an $n \times n$ array whose $(i, j)$th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

For undirected graphs, the matrix is symmetric since an edge $\{u, v\}$ can be taken in either direction.

## How big is your graph?

Which of the two representations, adjacency matrix or adjacency list, is better? Well, it depends on the relationship between $|V|$, the number of nodes in the graph, and $|E|$, the number of edges. $|E|$ can be as small as $|V|$ (if it gets much smaller, then the graph degenerates— for example, has isolated vertices), or as large as $|V|^2$ (when all possible edges are present). When $|E|$ is close to the upper limit of this range, we call the graph *dense*. At the other extreme, if $|E|$ is close to $|V|$, the graph is *sparse*. As we shall see in this chapter and the next two chapters, *exactly where $|E|$ lies in this range is usually a crucial factor in selecting the right graph algorithm*.

Or, for that matter, in selecting the graph representation. If it is the World Wide Web graph that we wish to store in computer memory, we should think twice before using an adjacency matrix: at the time of writing, search engines know of about eight billion vertices of this graph, and hence the adjacency matrix would take up *dozens of millions of terabits*. Again at the time we write these lines, it is not clear that there is enough computer memory in the whole world to achieve this. (And waiting a few years until there *is* enough memory is unwise: the Web will grow too and will probably grow faster.)

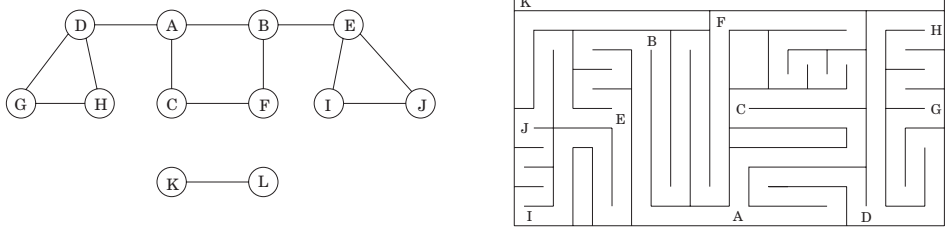
With adjacency lists, representing the World Wide Web becomes feasible: there are only a few dozen billion hyperlinks in the Web, and each will occupy a few bytes in the adjacency list. You can carry a device that stores the result, a terabyte or two, in your pocket (it may soon fit in your earring, but by that time the Web will have grown too).

The reason why adjacency lists are so much more effective in the case of the World Wide Web is that the Web is very sparse: the average Web page has hyperlinks to only about half a dozen other pages, out of the billions of possibilities.

The biggest convenience of this format is that the presence of a particular edge can be checked in constant time, with just one memory access. On the other hand the matrix takes up $O(n^2)$ space, which is wasteful if the graph does not have very many edges.

An alternative representation, with size proportional to the number of edges, is the *adjacency list*. It consists of $|V|$ linked lists, one per vertex. The linked list for vertex $u$ holds the names of vertices to which $u$ has an outgoing edge—that is, vertices $v$ for which $(u, v) \in E$. Therefore, each edge appears in exactly one of the linked lists if the graph is directed or two of the lists if the graph is undirected. Either way, the total size of the data structure is $O(|E|)$. Checking for a particular edge $(u, v)$ is no longer constant time, because it requires sifting through $u$'s adjacency list. But it is easy to iterate through all neighbors of a vertex (by running down the corresponding linked list), and, as we shall soon see, this turns out to be a very useful operation in graph algorithms. Again, for undirected graphs, this representation has a symmetry of sorts: $v$ is in $u$'s adjacency list if and only if $u$ is in $v$'s adjacency list.

**Figure 3.2** Exploring a graph is rather like navigating a maze.




## 3.2 Depth-first search in undirected graphs

### 3.2.1 Exploring mazes

*Depth-first search* is a surprisingly versatile linear-time procedure that reveals a wealth of information about a graph. The most basic question it addresses is,

*What parts of the graph are reachable from a given vertex?*

To understand this task, try putting yourself in the position of a computer that has just been given a new graph, say in the form of an adjacency list. This representation offers just one basic operation: finding the neighbors of a vertex. With only this primitive, the reachability problem is rather like exploring a labyrinth (Figure 3.2). You start walking from a fixed place and whenever you arrive at any junction (vertex) there are a variety of passages (edges) you can follow. A careless choice of passages might lead you around in circles or might cause you to overlook some accessible part of the maze. Clearly, you need to record some intermediate information during exploration.

This classic challenge has amused people for centuries. Everybody knows that all you need to explore a labyrinth is a ball of string and a piece of chalk. The chalk prevents looping, by marking the junctions you have already visited. The string always takes you back to the starting place, enabling you to return to passages that you previously saw but did not yet investigate.

How can we simulate these two primitives, chalk and string, on a computer? The chalk marks are easy: for each vertex, maintain a Boolean variable indicating whether it has been visited already. As for the ball of string, the correct cyber-analog is a *stack*. After all, the exact role of the string is to offer two primitive operations—*unwind* to get to a new junction (the stack equivalent is to *push* the new vertex) and *rewind* to return to the previous junction (*pop* the stack).

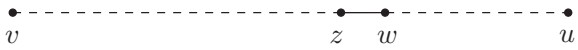
Instead of explicitly maintaining a stack, we will do so implicitly via recursion (which is implemented using a stack of activation records). The resulting algorithm

**Figure 3.3** Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable
   from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u): explore(u)
postvisit(v)
```

is shown in Figure 3.3.[1] The `previsit` and `postvisit` procedures are optional, meant for performing operations on a vertex when it is first discovered and also when it is being left for the last time. We will soon see some creative uses for them.

More immediately, we need to confirm that `explore` always works correctly. It certainly does not venture too far, because it only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from $v$. But does it find *all* vertices reachable from $v$? Well, if there is some $u$ that it misses, choose any path from $v$ to $u$, and look at the last vertex on that path that the procedure actually visited. Call this node $z$, and let $w$ be the node immediately after it on the same path.



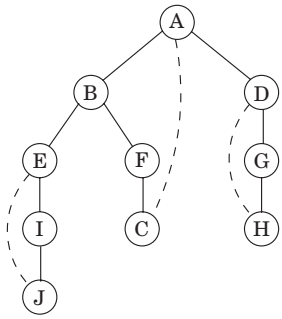

So $z$ was visited but $w$ was not. This is a contradiction: while the `explore` procedure was at node $z$, it would have noticed $w$ and moved on to it.

Incidentally, this pattern of reasoning arises often in the study of graphs and is in essence a streamlined induction. A more formal inductive proof would start by framing a hypothesis, such as "for any $k \geq 0$, all nodes within $k$ hops from $v$ get visited." The base case is as usual trivial, since $v$ is certainly visited. And the general case—showing that if all nodes $k$ hops away are visited, then so are all nodes $k+1$ hops away—is precisely the same point we just argued.

Figure 3.4 shows the result of running `explore` on our earlier example graph, starting at node $A$, and breaking ties in alphabetical order whenever there is a choice of nodes to visit. The solid edges are those that were actually traversed, each of which was elicited by a call to `explore` and led to the discovery of a new vertex.

[1] As with many of our graph algorithms, this one applies to both undirected and directed graphs. In such cases, we adopt the *directed* notation for edges, $(x, y)$. If the graph is undirected, then each of its edges should be thought of as existing in both directions: $(x, y)$ and $(y, x)$.

**Figure 3.4** The result of explore($A$) on the graph of Figure 3.2.




For instance, while $B$ was being visited, the edge $B - E$ was noticed and, since $E$ was as yet unknown, was traversed via a call to explore($E$). These solid edges form a tree (a connected graph with no cycles) and are therefore called *tree edges*. The dotted edges were ignored because they led back to familiar terrain, to vertices previously visited. They are called *back edges*.

### 3.2.2 Depth-first search

The explore procedure visits only the portion of the graph reachable from its starting point. To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited. The algorithm of Figure 3.5, called *depth-first search* (DFS), does this repeatedly until the entire graph has been traversed.

**Figure 3.5** Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v): explore(v)
```

The first step in analyzing the running time of DFS is to observe that each vertex is explore'd just once, thanks to the visited array (the chalk marks). During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work—marking the spot as visited, and the pre/postvisit.

2. A loop in which adjacent edges are scanned, to see if they lead somewhere new.

This loop takes a different amount of time for each vertex, so let's consider all vertices together. The total work done in step 1 is then $O(|V|)$. In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly *twice*, once during `explore(`$x$`)` and once during `explore(`$y$`)`. The overall time for step 2 is therefore $O(|E|)$ and so the depth-first search has a running time of $O(|V| + |E|)$, linear in the size of its input. This is as efficient as we could possibly hope for, since it takes this long even just to read the adjacency list.

**Figure 3.6** (a) A 12-node graph. (b) DFS search forest.

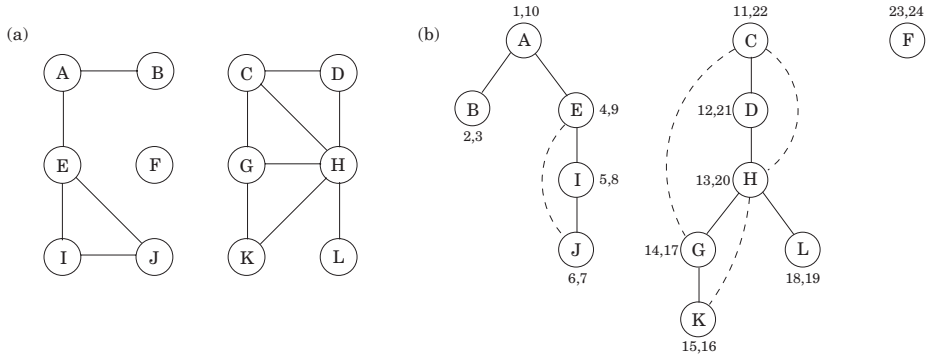



Figure 3.6 shows the outcome of depth-first search on a 12-node graph, once again breaking ties alphabetically (ignore the pairs of numbers for the time being). The outer loop of DFS calls `explore` three times, on $A$, $C$, and finally $F$. As a result, there are three trees, each rooted at one of these starting points. Together they constitute a *forest*.

### 3.2.3 Connectivity in undirected graphs

An undirected graph is *connected* if there is a path between any pair of vertices. The graph of Figure 3.6 is *not* connected because, for instance, there is no path from $A$ to $K$. However, it does have three disjoint connected regions, corresponding to the following sets of vertices:

$$\{A, B, E, I, J\} \qquad \{C, D, G, H, K, L\} \qquad \{F\}.$$

These regions are called *connected components*: each of them is a subgraph that is internally connected but has no edges to the remaining vertices. When `explore` is started at a particular vertex, it identifies precisely the connected component containing that vertex. And each time the DFS outer loop calls `explore`, a new connected component is picked out.

Thus depth-first search is trivially adapted to check if a graph is connected and, more generally, to assign each node $v$ an integer `ccnum`[$v$] identifying the connected component to which it belongs. All it takes is

```
procedure previsit(v)
ccnum[v] = cc
```

where `cc` needs to be initialized to zero and to be incremented each time the DFS procedure calls `explore`.

### 3.2.4 Previsit and postvisit orderings

We have seen how depth-first search—a few unassuming lines of code—is able to uncover the connectivity structure of an undirected graph in just linear time. But it is far more versatile than this. In order to stretch it further, we will collect a little more information during the exploration process: for each node, we will note down the times of two important events, the moment of first discovery (corresponding to `previsit`) and that of final departure (`postvisit`). Figure 3.6 shows these numbers for our earlier example, in which there are 24 events. The fifth event is the discovery of $I$. The 21st event consists of leaving $D$ behind for good.

One way to generate arrays `pre` and `post` with these numbers is to define a simple counter `clock`, initially set to 1, which gets updated as follows.

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

These timings will soon take on larger significance. Meanwhile, you might have noticed from Figure 3.4 that:

**Property** *For any nodes $u$ and $v$, the two intervals* $[pre(u), post(u)]$ *and* $[pre(v), post(v)]$ *are either disjoint or one is contained within the other.*

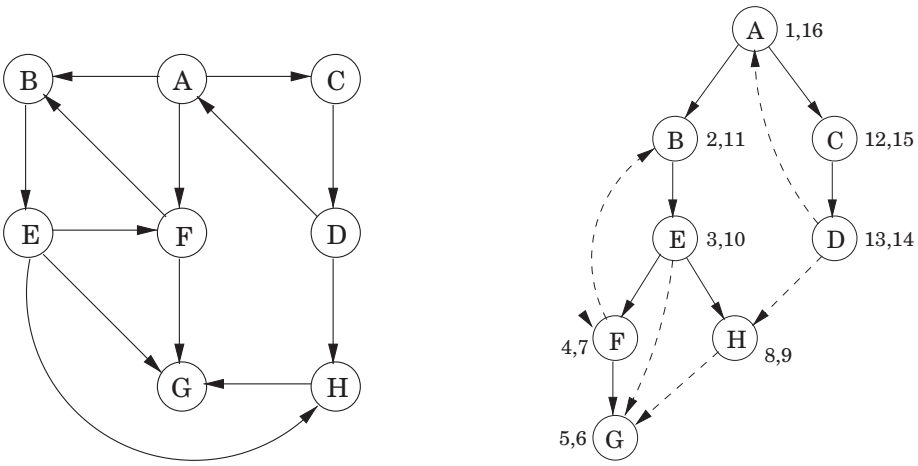
Why? Because $[\text{pre}(u), \text{post}(u)]$ is essentially the time during which vertex $u$ was on the stack. The last-in, first-out behavior of a stack explains the rest.

## 3.3 Depth-first search in directed graphs

### 3.3.1 Types of edges

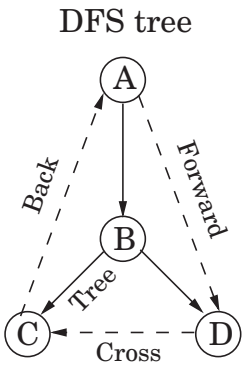
Our depth-first search algorithm can be run verbatim on directed graphs, taking care to traverse edges only in their prescribed directions. Figure 3.7 shows an example and the search tree that results when vertices are considered in lexicographic order.

**Figure 3.7** DFS on a directed graph.




In further analyzing the directed case, it helps to have terminology for important relationships between nodes of a tree. $A$ is the *root* of the search tree; everything else is its *descendant*. Similarly, $E$ has descendants $F$, $G$, and $H$, and conversely, is an *ancestor* of these three nodes. The family analogy is carried further: $C$ is the *parent* of $D$, which is its *child*.

For undirected graphs we distinguished between tree edges and nontree edges. In the directed case, there is a slightly more elaborate taxonomy:




*Tree edges* are actually part of the DFS forest.

*Forward edges* lead from a node to a *nonchild* descendant in the DFS tree.

*Back edges* lead to an ancestor in the DFS tree.

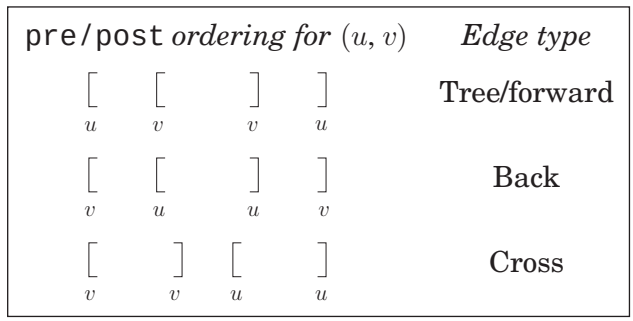
*Cross edges* lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

Figure 3.7 has two forward edges, two back edges, and two cross edges. Can you spot them?

Ancestor and descendant relationships, as well as edge types, can be read off directly from `pre` and `post` numbers. Because of the depth-first exploration strategy, vertex $u$ is an ancestor of vertex $v$ exactly in those cases where $u$ is discovered first and

$v$ is discovered during `explore`($u$). This is to say `pre`(u) < `pre`(v) < `post`(v) < `post`(u), which we can depict pictorially as two nested intervals:


The case of descendants is symmetric, since $u$ is a descendant of $v$ if and only if $v$ is an ancestor of $u$. And since edge categories are based entirely on ancestor-descendant relationships, it follows that they, too, can be read off from `pre` and `post` numbers. Here is a summary of the various possibilities for an edge $(u, v)$:




You can confirm each of these characterizations by consulting the diagram of edge types. Do you see why no other orderings are possible?

### 3.3.2 Directed acyclic graphs

A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$. Figure 3.7 has quite a few of them, for example, $B \to E \to F \to B$. A graph without cycles is *acyclic*. It turns out we can test for acyclicity in linear time, with a single depth-first search.

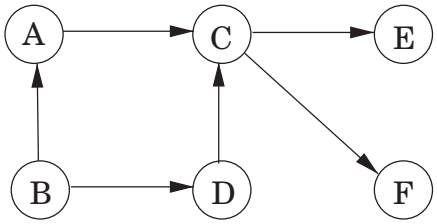
**Property** *A directed graph has a cycle if and only if its depth-first search reveals a back edge.*

*Proof.* One direction is quite easy: if $(u, v)$ is a back edge, then there is a cycle consisting of this edge together with the path from $v$ to $u$ in the search tree.

Conversely, if the graph has a cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$, look at the *first* node on this cycle to be discovered (the node with the lowest `pre` number). Suppose it is $v_i$. All the other $v_j$ on the cycle are reachable from it and will therefore be its descendants in the search tree. In particular, the edge $v_{i-1} \to v_i$ (or $v_k \to v_0$ if $i = 0$) leads from a node to its ancestor and is thus by definition a back edge. ∎

*Directed acyclic graphs*, or *dags* for short, come up all the time. They are good for modeling relations like causalities, hierarchies, and temporal dependencies. For example, suppose that you need to perform many tasks, but some of them cannot begin until certain others are completed (you have to wake up before you can get

**Figure 3.8** A directed acyclic graph with one source, two sinks, and four possible linearizations.




out of bed; you have to be out of bed, but not yet dressed, to take a shower; and so on). The question then is, what is a valid order in which to perform the tasks?

Such constraints are conveniently represented by a directed graph in which each task is a node, and there is an edge from $u$ to $v$ if $u$ is a precondition for $v$. In other words, before performing a task, all the tasks pointing to it must be completed. If this graph has a cycle, there is no hope: no ordering can possibly work. If on the other hand the graph is a dag, we would like if possible to *linearize* (or *topologically sort*) it, to order the vertices one after the other in such a way that each edge goes from an earlier vertex to a later vertex, so that all precedence constraints are satisfied. In Figure 3.8, for instance, one valid ordering is $B, A, D, C, E, F$. (Can you spot the other three?)

What types of dags can be linearized? Simple: *All of them*. And once again depth-first search tells us exactly how to do it: simply perform tasks in *decreasing* order of their `post` numbers. After all, the only edges $(u, v)$ in a graph for which `post`$(u)$ <`post`$(v)$ are back edges (recall the table of edge types on page 88)—and we have seen that a dag cannot have back edges. Therefore:

**Property** *In a dag, every edge leads to a vertex with a lower* `post` *number.*

This gives us a linear-time algorithm for ordering the nodes of a dag. And, together with our earlier observations, it tells us that three rather different-sounding properties—acyclicity, linearizability, and the absence of back edges during a depth-first search—are in fact one and the same thing.

Since a dag is linearized by decreasing `post` numbers, the vertex with the smallest `post` number comes last in this linearization, and it must be a *sink*—no outgoing edges. Symmetrically, the one with the highest `post` is a *source*, a node with no incoming edges.

**Property** *Every dag has at least one source and at least one sink.*

The guaranteed existence of a source suggests an alternative approach to linearization:

*Find a source, output it, and delete it from the graph.*

*Repeat until the graph is empty.*

Can you see why this generates a valid linearization for any dag? What happens if the graph has cycles? And, how can this algorithm be implemented in linear time? (Exercise 3.14.)

## 3.4 Strongly connected components

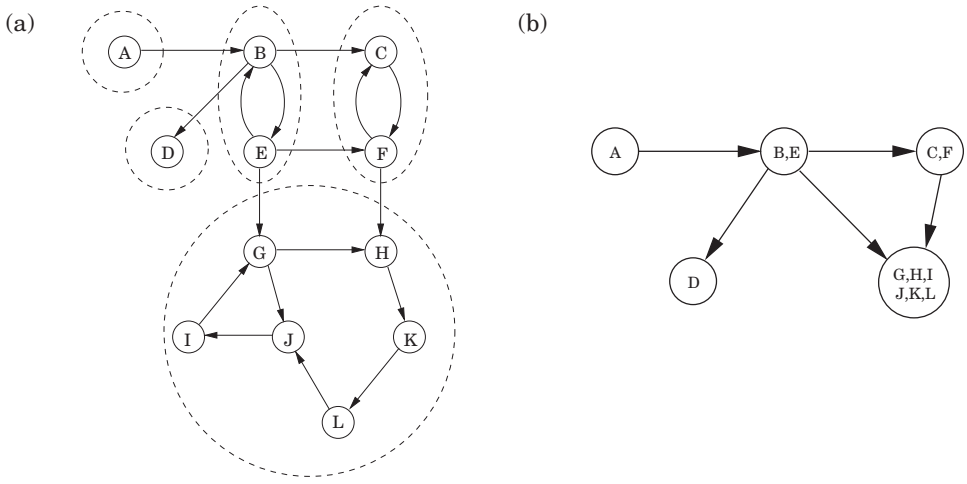
### 3.4.1 Defining connectivity for directed graphs

Connectivity in undirected graphs is pretty straightforward: a graph that is not connected can be decomposed in a natural and obvious manner into several connected components (Figure 3.6 is a case in point). As we saw in Section 3.2.3, depth-first search does this handily, with each restart marking a new connected component.

In directed graphs, connectivity is more subtle. In some primitive sense, the directed graph of Figure 3.9(a) is "connected"—it can't be "pulled apart," so to speak, without breaking edges. But this notion is hardly interesting or informative. The graph cannot be considered connected, because for instance there is no path from $G$ to $B$ or from $F$ to $A$. The right way to define connectivity for directed graphs is this:

*Two nodes u and v of a directed graph are connected if there is a path from u to v and a path from v to u.*

**Figure 3.9** (a) A directed graph and its strongly connected components. (b) The meta-graph.



This relation partitions $V$ into disjoint sets (Exercise 3.30) that we call *strongly connected components*. The graph of Figure 3.9(a) has five of them.

Now shrink each strongly connected component down to a single meta-node, and draw an edge from one meta-node to another if there is an edge (in the same direction) between their respective components (Figure 3.9(b)). The resulting *meta-graph* must be a dag. The reason is simple: a cycle containing several strongly connected components would merge them all into a single, strongly connected component. Restated,

**Property** *Every directed graph is a dag of its strongly connected components.*

This tells us something important: The connectivity structure of a directed graph is two-tiered. At the top level we have a dag, which is a rather simple structure—for instance, it can be linearized. If we want finer detail, we can look inside one of the nodes of this dag and examine the full-fledged strongly connected component within.

### 3.4.2 An efficient algorithm

The decomposition of a directed graph into its strongly connected components is very informative and useful. It turns out, fortunately, that it can be found in linear time by making further use of depth-first search. The algorithm is based on some properties we have already seen but which we will now pinpoint more closely.

**Property 1** *If the* `explore` *subroutine is started at node $u$, then it will terminate precisely when all nodes reachable from $u$ have been visited.*

Therefore, if we call `explore` on a node that lies somewhere in a *sink* strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component. Figure 3.9 has two sink strongly connected components. Starting `explore` at node $K$, for instance, will completely traverse the larger of them and then stop.

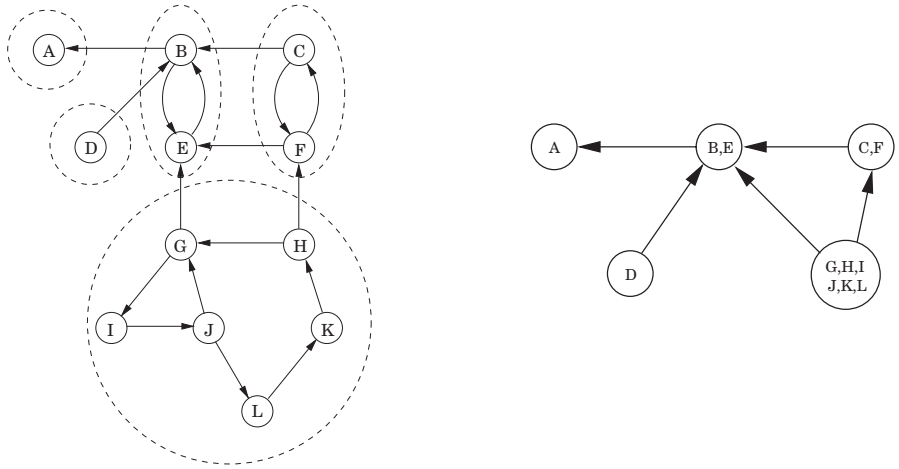
This suggests a way of finding one strongly connected component, but still leaves open two major problems: (A) how do we find a node that we know for sure lies in a sink strongly connected component and (B) how do we continue once this first component has been discovered?

Let's start with problem (A). There is not an easy, direct way to pick out a node that is guaranteed to lie in a sink strongly connected component. But there is a way to get a node in a *source* strongly connected component.

**Property 2** *The node that receives the highest* `post` *number in a depth-first search must lie in a source strongly connected component.*

This follows from the following more general property.

**Figure 3.10** The *reverse* of the graph from Figure 3.9.




**Property 3** *If $C$ and $C'$ are strongly connected components, and there is an edge from a node in $C$ to a node in $C'$, then the highest* `post` *number in $C$ is bigger than the highest* `post` *number in $C'$.*

*Proof.* In proving Property 3, there are two cases to consider. If the depth-first search visits component $C$ before component $C'$, then clearly all of $C$ and $C'$ will be traversed before the procedure gets stuck (see Property 1). Therefore the first node visited in $C$ will have a higher `post` number than any node of $C'$. On the other hand, if $C'$ gets visited first, then the depth-first search will get stuck after seeing all of $C'$ but before seeing any of $C$, in which case the property follows immediately. ∎

Property 3 can be restated as saying that *the strongly connected components can be linearized by arranging them in decreasing order of their highest* `post` *numbers*. This is a generalization of our earlier algorithm for linearizing dags; in a dag, each node is a singleton strongly connected component.

Property 2 helps us find a node in the source strongly connected component of $G$. However, what we need is a node in the *sink* component. Our means seem to be the opposite of our needs! But consider the *reverse* graph $G^R$, the same as $G$ but with all edges reversed (Figure 3.10). $G^R$ has exactly the same strongly connected components as $G$ (why?). So, if we do a depth-first search of $G^R$, the node with the highest `post` number will come from a source strongly connected component in $G^R$, which is to say a sink strongly connected component in $G$. We have solved problem (A)!

Onward to problem (B). How do we continue after the first sink component is identified? The solution is also provided by Property 3. Once we have found the first strongly connected component and deleted it from the graph, the node with

## Crawling fast

All this assumes that the graph is neatly given to us, with vertices numbered 1 to $n$ and edges tucked in adjacency lists. The realities of the World Wide Web are very different. The nodes of the Web graph are not known in advance, and they have to be discovered one by one during the process of search. And, of course, recursion is out of the question.

Still, crawling the Web is done by algorithms very similar to depth-first search. An explicit stack is maintained, containing all nodes that have been discovered (as endpoints of hyperlinks) but not yet explored. In fact, this "stack" is not exactly a last-in, first-out list. It gives highest priority not to the nodes that were inserted most recently (nor the ones that were inserted earliest, that would be a *breadth-first search*, see Chapter 2), but to the ones that look most "interesting"—a heuristic criterion whose purpose is to keep the stack from overflowing and, in the worst case, to leave unexplored only nodes that are very unlikely to lead to vast new expanses.

In fact, crawling is typically done by many computers running `explore` simultaneously: each one takes the next node to be explored from the top of the stack, downloads the http file (the kind of Web files that point to each other), and scans it for hyperlinks. But when a new http document is found at the end of a hyperlink, no recursive calls are made: instead, the new vertex is inserted in the central stack.

But one question remains: When we see a "new" document, how do we know that it is indeed new, that we have not seen it before in our crawl? And how do we give it a *name*, so it can be inserted in the stack and recorded as "already seen"? The answer is *by hashing*.

Incidentally, researchers have run the strongly connected components algorithm on the Web and have discovered some very interesting structure.

the highest `post` number among those remaining will belong to a sink strongly connected component of whatever remains of $G$. Therefore we can keep using the `post` numbering from our initial depth-first search on $G^R$ to successively output the second strongly connected component, the third strongly connected component, and so on. The resulting algorithm is this.

1. Run depth-first search on $G^R$.
2. Run the undirected connected components algorithm (from Section 3.2.3) on $G$, and during the depth-first search, process the vertices in decreasing order of their `post` numbers from step 1.

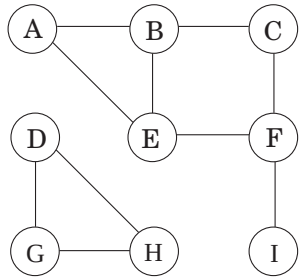
This algorithm is linear-time, only the constant in the linear term is about twice that of straight depth-first search. (Question: How does one construct an adjacency list representation of $G^R$ in linear time? And how, in linear time, does one order the vertices of $G$ by decreasing `post` values?)

Let's run this algorithm on the graph of Figure 3.9. If step 1 considers vertices in lexicographic order, then the ordering it sets up for the second step (namely,

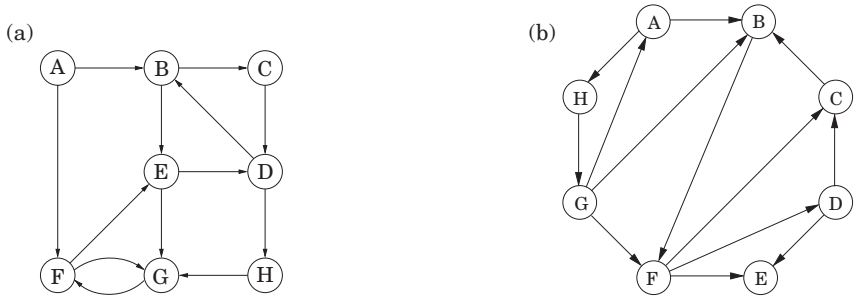
decreasing `post` numbers in the depth-first search of $G^R$) is: $G, I, J, L, K, H, D, C, F, B, E, A$. Then step 2 peels off components in the following sequence: $\{G, H, I, J, K, L\}, \{D\}, \{C, F\}, \{B, E\}, \{A\}$.

## Exercises

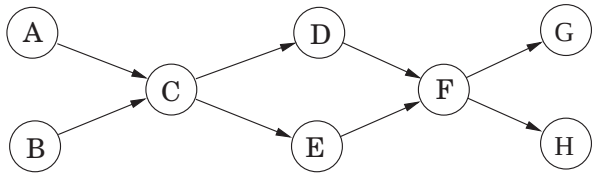
3.1. Perform a depth-first search on the following graph; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge or back edge, and give the `pre` and `post` number of each vertex.




3.2. Perform depth-first search on each of the following graphs; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge, or cross edge, and give the `pre` and `post` number of each vertex.




3.3. Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.




(a) Indicate the `pre` and `post` numbers of the nodes.

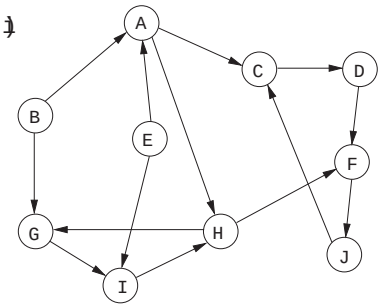
(b) What are the sources and sinks of the graph?

(c) What topological ordering is found by the algorithm?

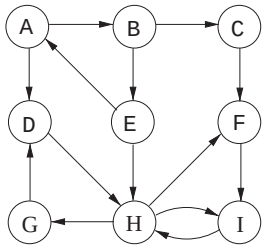
(d) How many topological orderings does this graph have?

3.4. Run the strongly connected components algorithm on the following directed graphs $G$. When doing DFS on $G^R$: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

(i)




(ii)




In each case answer the following questions.

(a) In what order are the strongly connected components (SCCs) found?

(b) Which are source SCCs and which are sink SCCs?

(c) Draw the "metagraph" (each meta-node is an SCC of $G$).

(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

3.5. The *reverse* of a directed graph $G = (V, E)$ is another directed graph $G^R = (V, E^R)$ on the same vertex set, but with all edges reversed; that is, $E^R = \{(v, u) : (u, v) \in E\}$.

Give a linear-time algorithm for computing the reverse of a graph in adjacency list format.

3.6. In an undirected graph, the *degree* $d(u)$ of a vertex $u$ is the number of neighbors $u$ has, or equivalently, the number of edges incident upon it. In a directed graph, we distinguish between the *indegree* $d_{in}(u)$, which is the number of edges into $u$, and the *outdegree* $d_{out}(u)$, the number of edges leaving $u$.

(a) Show that in an undirected graph, $\sum_{u \in V} d(u) = 2|E|$.

(b) Use part (a) to show that in an undirected graph, there must be an even number of vertices whose degree is odd.

(c) Does a similar statement hold for the number of vertices with odd indegree in a directed graph?

3.7. A *bipartite graph* is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$, then there is no edge between $u$ and $v$).

(a) Give a linear-time algorithm to determine whether an undirected graph is bipartite.

(b) There are many other ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors.

Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.

(c) At most how many colors are needed to color in an undirected graph with exactly *one* odd-length cycle?

3.8. *Pouring water.* We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

(a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

(b) What algorithm should be applied to solve the problem?

3.9. For each node $u$ in an undirected graph, let `twodegree`$[u]$ be the sum of the degrees of $u$'s neighbors. Show how to compute the entire array of `twodegree`$[\cdot]$ values in linear time, given a graph in adjacency list format.

3.10. Rewrite the `explore` procedure (Figure 3.3) so that it is non-recursive (that is, explicitly use a stack). The calls to `previsit` and `postvisit` should be positioned so that they have the same effect as in the recursive procedure.

3.11. Design a linear-time algorithm which, given an undirected graph $G$ and a particular edge $e$ in it, determines whether $G$ has a cycle containing $e$.

3.12. Either prove or give a counterexample: if $\{u, v\}$ is an edge in an undirected graph, and during depth-first search `post`$(u) <$ `post`$(v)$, then $v$ is an ancestor of $u$ in the DFS tree.

3.13. *Undirected vs. directed connectivity.*

(a) Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ whose removal leaves $G$ connected. (*Hint:* Consider the DFS search tree for $G$.)

(b) Give an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing $v$ from $G$ leaves a directed graph that is not strongly connected.

(c) In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

3.14. The chapter suggests an alternative algorithm for linearization (topological sorting), which repeatedly removes source nodes from the graph (page 90). Show that this algorithm can be implemented in linear time.

3.15. The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

(a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.

(b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

3.16. Suppose a CS curriculum consists of $n$ courses, all of them mandatory. The prerequisite graph $G$ has a node for each course, and an edge from course $v$ to course $w$ if and only if $v$ is a prerequisite for $w$. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). The running time of your algorithm should be linear.

3.17. *Infinite paths.* Let $G = (V, E)$ be a directed graph with a designated "start vertex" $s \in V$, a set $V_G \subseteq V$ of "good" vertices, and a set $V_B \subseteq V$ of "bad" vertices. An *infinite* trace $p$ of $G$ is an infinite sequence $v_0 v_1 v_2 \cdots$ of vertices $v_i \in V$ such that (1) $v_0 = s$, and (2) for all $i \geq 0$, $(v_i, v_{i+1}) \in E$. That is, $p$ is an infinite path in $G$ starting at vertex $s$. Since the set $V$ of vertices is finite, every infinite trace of $G$ must visit some vertices infinitely often.

(a) If $p$ is an infinite trace, let $Inf(p) \subseteq V$ be the set of vertices that occur infinitely often in $p$. Show that $Inf(p)$ is a subset of a strongly connected component of $G$.

(b) Describe an algorithm that determines if $G$ has an infinite trace.

(c) Describe an algorithm that determines if $G$ has an infinite trace that visits some good vertex in $V_G$ infinitely often.

(d) Describe an algorithm that determines if $G$ has an infinite trace that visits some good vertex in $V_G$ infinitely often, but visits no bad vertex in $V_B$ infinitely often.

3.18. You are given a binary tree $T = (V, E)$ (in adjacency list format), along with a designated root node $r \in V$. Recall that $u$ is said to be an *ancestor* of $v$ in the rooted tree, if the path from $r$ to $v$ in $T$ passes through $u$.

You wish to preprocess the tree so that queries of the form "is $u$ an ancestor of $v$?" can be answered in constant time. The preprocessing itself should take linear time. How can this be done?

3.19. As in the previous problem, you are given a binary tree $T = (V, E)$ with designated root node. In addition, there is an array $x[\cdot]$ with a value for each node in $V$. Define a new array $z[\cdot]$ as follows: for each $u \in V$,

$z[u]$ = *the maximum of the x-values associated with u's descendants.*

Give a linear-time algorithm which calculates the entire $z$-array.

3.20. You are given a tree $T = (V, E)$ along with a designated root node $r \in V$. The *parent* of any node $v \neq r$, denoted $p(v)$, is defined to be the node adjacent to $v$ in the path from $r$ to $v$. By convention, $p(r) = r$. For $k > 1$, define $p^k(v) = p^{k-1}(p(v))$ and $p^1(v) = p(v)$ (so $p^k(v)$ is the $k$th ancestor of $v$).

Each vertex $v$ of the tree has an associated non-negative integer label $l(v)$. Give a linear-time algorithm to update the labels of all the vertices in $T$ according to the following rule: $l_{new}(v) = l(p^{l(v)}(v))$.

3.21. Give a linear-time algorithm to find an odd-length cycle in a *directed* graph. (*Hint:* First solve this problem under the assumption that the graph is strongly connected.)

3.22. Give an efficient algorithm which takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

3.23. Give an efficient algorithm that takes as input a directed acyclic graph $G = (V, E)$, and two vertices $s, t \in V$, and outputs the number of different directed paths from $s$ to $t$ in $G$.

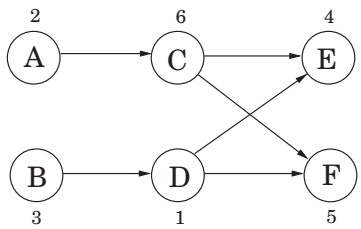
3.24. Give a linear-time algorithm for the following task.

*Input:* A directed acyclic graph $G$
*Question:* Does $G$ contain a directed path that touches every vertex exactly once?

3.25. You are given a directed graph in which each node $u \in V$ has an associated *price* $p_u$ which is a positive integer. Define the array `cost` as follows: for each $u \in V$,

`cost`$[u]$ = price of the cheapest node reachable from $u$ (including $u$ itself).

For instance, in the graph below (with prices shown for each vertex), the `cost` values of the nodes $A, B, C, D, E, F$ are 2, 1, 4, 1, 4, 5, respectively.



Your goal is to design an algorithm that fills in the *entire* `cost` array (i.e., for all vertices).

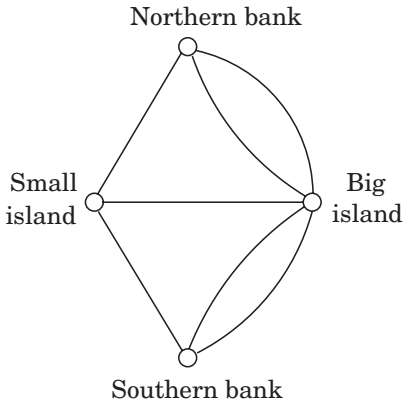
(a) Give a linear-time algorithm that works for directed *acyclic* graphs. (*Hint:* Handle the vertices in a particular *order*.)

(b) Extend this to a linear-time algorithm that works for all directed graphs. (*Hint:* Recall the "two-tiered" structure of directed graphs.)

3.26. An *Eulerian tour* in an undirected graph is a cycle that is allowed to pass through each vertex multiple times, but must use each edge exactly once.

This simple concept was used by Euler in 1736 to solve the famous Konigsberg bridge problem, which launched the field of graph theory. The city of Konigsberg (now called Kaliningrad, in western Russia) is the meeting point of two rivers with a small island in the middle. There are seven bridges across the rivers, and a popular recreational question of the time was to determine whether it is possible to perform a tour in which each bridge is crossed *exactly once*.

Euler formulated the relevant information as a graph with four nodes (denoting land masses) and seven edges (denoting bridges), as shown here.




Notice an unusual feature of this problem: multiple edges between certain pairs of nodes.

(a) Show that an undirected graph has an Eulerian tour if and only if all its vertices have even degree. Conclude that there is no Eulerian tour of the Konigsberg bridges.

(b) An *Eulerian path* is a path which uses each edge exactly once. Can you give a similar if-and-only-if characterization of which undirected graphs have Eulerian paths?

(c) Can you give an analog of part (a) for *directed* graphs?

3.27. Two paths in a graph are called *edge-disjoint* if they have no edges in common. Show that in any undirected graph, it is possible to pair up the vertices of odd degree and find paths between each such pair so that all these paths are edge-disjoint.

3.28. In the 2SAT problem, you are given a set of *clauses*, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value `true` or `false` to each of the variables so that *all* clauses are satisfied—that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x}_2) \ \wedge \ (\overline{x}_1 \vee \overline{x}_3) \ \wedge \ (x_1 \vee x_2) \ \wedge \ (\overline{x}_3 \vee x_4) \ \wedge \ (\overline{x}_1 \vee x_4).$$

This instance has a satisfying assignment: set $x_1$, $x_2$, $x_3$, and $x_4$ to `true`, `false`, `false`, and `true`, respectively.

(a) Are there other satisfying truth assignments of this 2SAT formula? If so, find them all.

(b) Give an instance of 2SAT with four variables, and with no satisfying assignment.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance $I$ of 2SAT with $n$ variables and $m$ clauses, construct a directed graph $G_I = (V, E)$ as follows.

- $G_I$ has $2n$ nodes, one for each variable and its negation.
- $G_I$ has $2m$ edges: for each clause $(\alpha \vee \beta)$ of $I$ (where $\alpha, \beta$ are literals), $G_I$ has an edge from the negation of $\alpha$ to $\beta$, and one from the negation of $\beta$ to $\alpha$.

Note that the clause $(\alpha \vee \beta)$ is equivalent to either of the implications $\overline{\alpha} \Rightarrow \beta$ or $\overline{\beta} \Rightarrow \alpha$. In this sense, $G_I$ records all implications in $I$.

(c) Carry out this construction for the instance of 2SAT given above, and for the instance you constructed in (b).

(d) Show that if $G_I$ has a strongly connected component containing both $x$ and $\overline{x}$ for some variable $x$, then $I$ has no satisfying assignment.

(e) Now show the converse of (d): namely, that if none of $G_I$'s strongly connected components contain both a literal and its negation, then the instance $I$ must be satisfiable. (*Hint:* Assign values to the variables as follows: repeatedly pick a sink strongly connected component of $G_I$. Assign value `true` to all literals in the sink, assign `false` to their negations, and delete all of these. Show that this ends up discovering a satisfying assignment.)

(f) Conclude that there is a linear-time algorithm for solving 2SAT.

3.29. Let $S$ be a finite set. A binary *relation* on $S$ is simply a collection $R$ of ordered pairs $(x, y) \in S \times S$. For instance, $S$ might be a set of people, and each such pair $(x, y) \in R$ might mean "$x$ knows $y$."

An *equivalence relation* is a binary relation which satisfies three properties:

- Reflexivity: $(x, x) \in R$ for all $x \in S$
- Symmetry: if $(x, y) \in R$ then $(y, x) \in R$
- Transitivity: if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$

For instance, the binary relation "has the same birthday as" is an equivalence relation, whereas "is the father of" is not, since it violates all three properties.

Show that an equivalence relation partitions set $S$ into disjoint groups $S_1, S_2, \ldots, S_k$ (in other words, $S = S_1 \cup S_2 \cup \cdots \cup S_k$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$) such that:

- Any two members of a group are related, that is, $(x, y) \in R$ for any $x, y \in S_i$, for any $i$.
- Members of different groups are not related, that is, for all $i \neq j$, for all $x \in S_i$ and $y \in S_j$, we have $(x, y) \notin R$.

(*Hint:* Represent an equivalence relation by an undirected graph.)

3.30. On page 91, we defined the binary relation "connected" on the set of vertices of a *directed* graph. Show that this is an equivalence relation (see Exercise 3.29), and conclude that it partitions the vertices into disjoint strongly connected components.

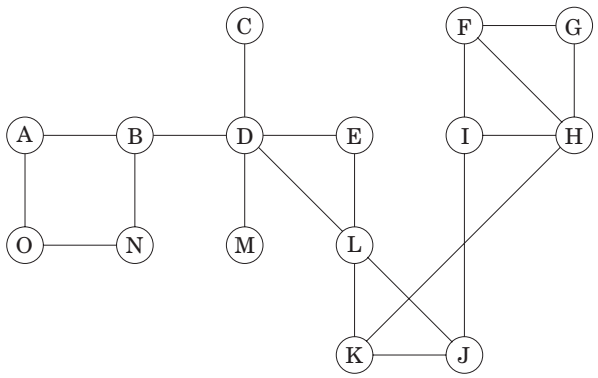
3.31. *Biconnected components*. Let $G = (V, E)$ be an undirected graph. For any two edges $e, e' \in E$, we'll say $e \sim e'$ if either $e = e'$ or there is a (simple) cycle containing both $e$ and $e'$.

(a) Show that $\sim$ is an equivalence relation (recall Exercise 3.29) on the edges.

The equivalence classes into which this relation partitions the edges are called the *biconnected components* of $G$. A *bridge* is an edge which is in a biconnected component all by itself.

A *separating vertex* is a vertex whose removal disconnects the graph.

(b) Partition the edges of the graph below into biconnected components, and identify the bridges and separating vertices.




Not only do biconnected components partition the edges of the graph, they also *almost* partition the vertices in the following sense.

(c) Associate with each biconnected component all the vertices that are endpoints of its edges. Show that the vertices corresponding to two

different biconnected components are either disjoint or intersect in a single separating vertex.

(d) Collapse each biconnected component into a single meta-node, and retain individual nodes for each separating vertex. (So there are edges between each component-node and its separating vertices.) Show that the resulting graph is a tree.

DFS can be used to identify the biconnected components, bridges, and separating vertices of a graph in linear time.

(e) Show that the root of the DFS tree is a separating vertex if and only if it has more than one child in the tree.

(f) Show that a non-root vertex $v$ of the DFS tree is a separating vertex if and only if it has a child $v'$ none of whose descendants (including itself) has a backedge to a proper ancestor of $v$.

(g) For each vertex $u$ define:

$$\text{low}(u) = \min \begin{cases} \texttt{pre}(u) \\ \texttt{pre}(w) \text{ where } (v, w) \text{ is a backedge for} \\ \qquad\qquad \text{some descendant } v \text{ of } u \end{cases}$$

Show that the entire array of `low` values can be computed in linear time.

(h) Show how to compute all separating vertices, bridges, and biconnected components of a graph in linear time. (*Hint:* Use `low` to identify separating vertices, and run another DFS with an extra stack of edges to remove biconnected components one at a time.)

*Chapter 4*

# Paths in graphs

## 4.1 Distances

Depth-first search readily identifies all the vertices of a graph that can be reached from a designated starting point. It also finds explicit paths to these vertices, summarized in its search tree (Figure 4.1). However, these paths might not be the most economical ones possible. In the figure, vertex $C$ is reachable from $S$ by traversing just one edge, while the DFS tree shows a path of length 3. This chapter is about algorithms for finding *shortest paths* in graphs.

Path lengths allow us to talk quantitatively about the extent to which different vertices of a graph are separated from each other:

*The* distance *between two nodes is the length of the shortest path between them.*

To get a concrete feel for this notion, consider a physical realization of a graph that has a ball for each vertex and a piece of string for each edge. If you lift the ball for vertex $s$ high enough, the other balls that get pulled up along with it are precisely the vertices reachable from $s$. And to find their distances from $s$, you need only measure how far below $s$ they hang.

In Figure 4.2, for example, vertex $B$ is at distance 2 from $S$, and there are two shortest paths to it. When $S$ is held up, the strings along each of these paths become taut.

**Figure 4.1** (a) A simple graph and (b) its depth-first search tree.

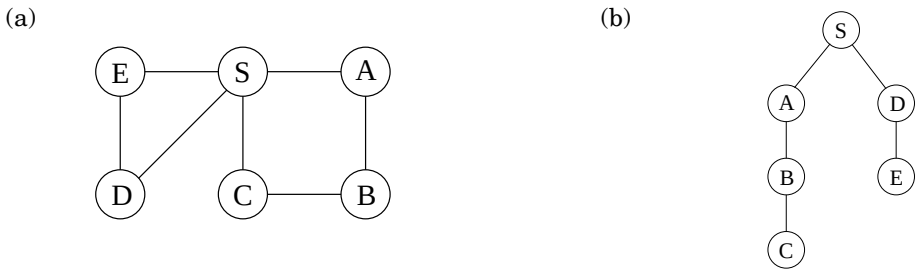
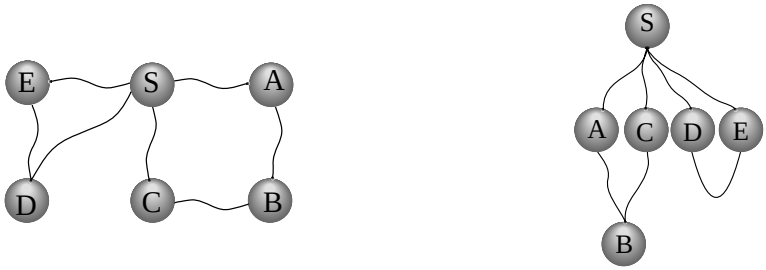


**Figure 4.2** A physical model of a graph.




On the other hand, edge $(D, E)$ plays no role in any shortest path and therefore remains slack.

## 4.2 Breadth-first search

In Figure 4.2, the lifting of $s$ partitions the graph into layers: $s$ itself, the nodes at distance 1 from it, the nodes at distance 2 from it, and so on. A convenient way to compute distances from $s$ to the other vertices is to proceed layer by layer. Once we have picked out the nodes at distance $0, 1, 2, \ldots, d$, the ones at $d + 1$ are easily determined: they are precisely the as-yet-unseen nodes that are adjacent to the layer at distance $d$. This suggests an iterative algorithm in which two layers are active at any given time: some layer $d$, which has been fully identified, and $d + 1$, which is being discovered by scanning the neighbors of layer $d$.

Breadth-first search (BFS) directly implements this simple reasoning (Figure 4.3). Initially the queue $Q$ consists only of $s$, the one node at distance 0. And for each subsequent distance $d = 1, 2, 3, \ldots$, there is a point in time at which $Q$ contains all the nodes at distance $d$ and nothing else. As these nodes are processed (ejected off the front of the queue), their as-yet-unseen neighbors are injected into the end of the queue.

Let's try out this algorithm on our earlier example (Figure 4.1) to confirm that it does the right thing. If $S$ is the starting point and the nodes are ordered alphabetically, they get visited in the sequence shown in Figure 4.4. The breadth-first search tree, on the right, contains the edges through which each node is initially discovered. Unlike the DFS tree we saw earlier, it has the property that all its paths from $S$ are the shortest possible. It is therefore a *shortest-path tree*.

**Correctness and efficiency**

We have developed the basic intuition behind breadth-first search. In order to check that the algorithm works correctly, we need to make sure that it faithfully executes this intuition. What we expect, precisely, is that

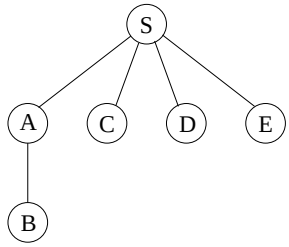
> *For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from $s$ have their distances correctly set; (2) all other nodes have their distances set to $\infty$; and (3) the queue contains exactly the nodes at distance $d$.*

**Figure 4.3** Breadth-first search.

```
procedure bfs(G, s)
Input: Graph G = (V, E), directed or undirected; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
```

**Figure 4.4** The result of breadth-first search on the graph of Figure 4.1.

| Order of visitation | Queue contents after processing node |
|---|---|
| | $[S]$ |
| $S$ | $[A\ C\ D\ E]$ |
| $A$ | $[C\ D\ E\ B]$ |
| $C$ | $[D\ E\ B]$ |
| $D$ | $[E\ B]$ |
| $E$ | $[B]$ |
| $B$ | $[\,]$ |




This has been phrased with an inductive argument in mind. We have already discussed both the base case and the inductive step. Can you fill in the details?

The overall running time of this algorithm is linear, $O(|V| + |E|)$, for exactly the same reasons as depth-first search. Each vertex is put on the queue exactly once, when it is first encountered, so there are $2|V|$ queue operations. The rest of the work is done in the algorithm's innermost loop. Over the course of execution, this loop looks at each edge once (in directed graphs) or twice (in undirected graphs), and therefore takes $O(|E|)$ time.

Now that we have both BFS and DFS before us: how do their exploration styles compare? Depth-first search makes deep incursions into a graph, retreating only when it runs out of new nodes to visit. This strategy gives it the wonderful, subtle,

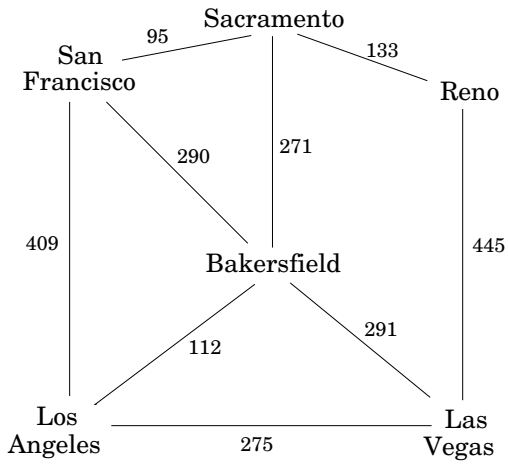
and extremely useful properties we saw in Chapter 3. But it also means that DFS can end up taking a long and convoluted route to a vertex that is actually very close by, as in Figure 4.1. Breadth-first search makes sure to visit vertices in increasing order of their distance from the starting point. This is a broader, shallower search, rather like the propagation of a wave upon water. And it is achieved using almost exactly the same code as DFS—but with a queue in place of a stack.

Also notice one stylistic difference from DFS: since we are only interested in distances from $s$, we do not restart the search in other connected components. Nodes not reachable from $s$ are simply ignored.

## 4.3 Lengths on edges

Breadth-first search treats all edges as having the same length. This is rarely true in applications where shortest paths are to be found. For instance, suppose you are driving from San Francisco to Las Vegas, and want to find the quickest route. Figure 4.5 shows the major highways you might conceivably use. Picking the right combination of them is a shortest-path problem in which the length of each edge (each stretch of highway) is important. For the remainder of this chapter, we will deal with this more general scenario, annotating every edge $e \in E$ with a length $l_e$. If $e = (u, v)$, we will sometimes also write $l(u, v)$ or $l_{uv}$.

**Figure 4.5** Edge lengths often matter.




These $l_e$'s do not have to correspond to physical lengths. They could denote time (driving time between cities) or money (cost of taking a bus), or any other quantity that we would like to conserve. In fact, there are cases in which we need to use negative lengths, but we will briefly overlook this particular complication.

# 4.4 Dijkstra's algorithm

## 4.4.1 An adaptation of breadth-first search

Breadth-first search finds shortest paths in any graph whose edges have unit length. Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths $l_e$ are *positive integers*?

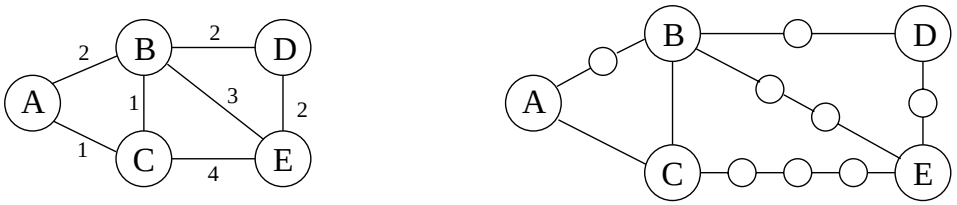
### A more convenient graph

Here is a simple trick for converting $G$ into something BFS can handle: break $G$'s long edges into unit-length pieces by introducing "dummy" nodes. Figure 4.6 shows an example of this transformation. To construct the new graph $G'$,

> *For any edge $e = (u, v)$ of $E$, replace it by $l_e$ edges of length 1, by adding $l_e - 1$ dummy nodes between $u$ and $v$.*

Graph $G'$ contains all the vertices $V$ that interest us, and the distances between them are exactly the same as in $G$. Most importantly, the edges of $G'$ all have unit length. Therefore, we can compute distances in $G$ by running BFS on $G'$.

**Figure 4.6** Breaking edges into unit-length pieces.



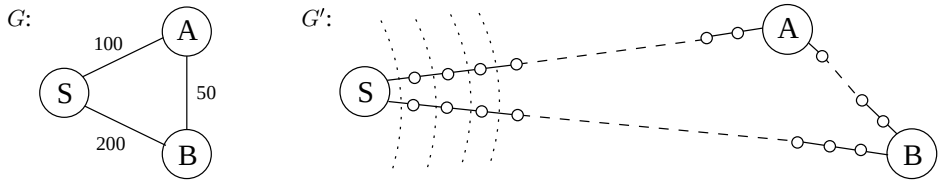

### Alarm clocks

If efficiency were not an issue, we could stop here. But when $G$ has very long edges, the $G'$ it engenders is thickly populated with dummy nodes, and the BFS spends most of its time diligently computing distances to these nodes that we don't care about at all.

To see this more concretely, consider the graphs $G$ and $G'$ of Figure 4.7, and imagine that the BFS, started at node $s$ of $G'$, advances by one unit of distance per minute. For the first 99 minutes it tediously progresses along $S - A$ and $S - B$, an endless desert of dummy nodes. Is there some way we can snooze through these boring phases and have an alarm wake us up whenever something *interesting* is happening—specifically, whenever one of the real nodes (from the original graph $G$) is reached?

We do this by setting two alarms at the outset, one for node $A$, set to go off at time $T = 100$, and one for $B$, at time $T = 200$. These are *estimated times of arrival*, based upon the edges currently being traversed. We doze off and awake at $T = 100$ to find $A$ has been discovered. At this point, the estimated time of arrival for $B$ is adjusted to $T = 150$ and we change its alarm accordingly.

**Figure 4.7** BFS on $G'$ is mostly uneventful. The dotted lines show some early "wavefronts."




More generally, at any given moment the breadth-first search is advancing along certain edges of $G$, and there is an alarm for every endpoint node toward which it is moving, set to go off at the estimated time of arrival at that node. Some of these might be overestimates because BFS may later find shortcuts, as a result of future arrivals elsewhere. In the preceding example, a quicker route to $B$ was revealed upon arrival at $A$. However, *nothing interesting can possibly happen before an alarm goes off*. The sounding of the next alarm must therefore signal the arrival of the wavefront to a real node $u \in V$ by BFS. At that point, BFS might also start advancing along some new edges out of $u$, and alarms need to be set for their endpoints.

The following "alarm clock algorithm" faithfully simulates the execution of BFS on $G'$.

- Set an alarm clock for node $s$ at time 0.
- Repeat until there are no more alarms:

  Say the next alarm goes off at time $T$, for node $u$. Then:
  - The distance from $s$ to $u$ is $T$.
  - For each neighbor $v$ of $u$ in $G$:
    - If there is no alarm yet for $v$, set one for time $T + l(u, v)$.
    - If $v$'s alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.

**Dijkstra's algorithm**

The alarm clock algorithm computes distances in any graph with positive integral edge lengths. It is almost ready for use, except that we need to somehow implement the system of alarms. The right data structure for this job is a *priority queue* (usually implemented via a *heap*), which maintains a set of elements (nodes) with associated numeric key values (alarm times) and supports the following operations:

*Insert.* Add a new element to the set.

*Decrease-key.* Accommodate the decrease in key value of a particular element.[1]

[1]The name *decrease-key* is standard but is a little misleading: the priority queue typically does not itself change key values. What this procedure really does is to notify the queue that a certain key value has been decreased.

*Delete-min.* Return the element with the smallest key, and remove it from the set.

*Make-queue.* Build a priority queue out of the given elements with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The first two let us set alarms, and the third tells us which alarm is next to go off. Putting this all together, we get Dijkstra's algorithm (Figure 4.8).

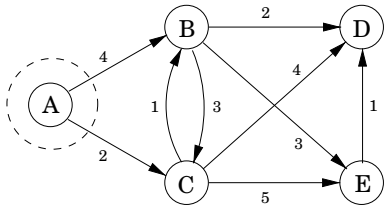
**Figure 4.8** Dijkstra's shortest-path algorithm.

```
procedure dijkstra(G, l, s)
Input:  Graph G = (V, E), directed or undirected;
        positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0


H = makequeue(V)  (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
       if dist(v) > dist(u) + l(u, v):
          dist(v) = dist(u) + l(u, v)
          prev(v) = u
          decreasekey(H, v)
```

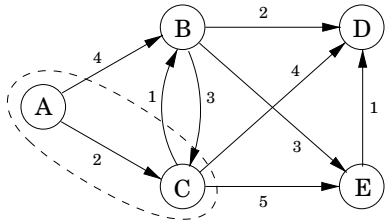
In the code, `dist`$(u)$ refers to the current alarm clock setting for node $u$. A value of $\infty$ means the alarm hasn't so far been set. There is also a special array, `prev`, that holds one crucial piece of information for each node $u$: the identity of the node immediately before it on the shortest path from $s$ to $u$. By following these back-pointers, we can easily reconstruct shortest paths, and so this array is a compact summary of all the paths found. A full example of the algorithm's operation, along with the final shortest-path tree, is shown in Figure 4.9.

In summary, we can think of Dijkstra's algorithm as just BFS, except it uses a priority queue instead of a regular queue, so as to prioritize nodes in a way that takes edge lengths into account. This viewpoint gives a concrete appreciation of how and why the algorithm works, but there is a more direct, more abstract derivation that

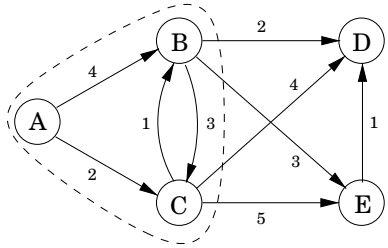
**Figure 4.9** A complete run of Dijkstra's algorithm, with node *A* as the starting point. Also shown are the associated `dist` values and the final shortest-path tree.



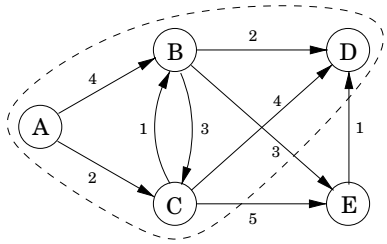

| | |
|---|---|
| A: 0 | D: $\infty$ |
| B: 4 | E: $\infty$ |
| C: 2 | |



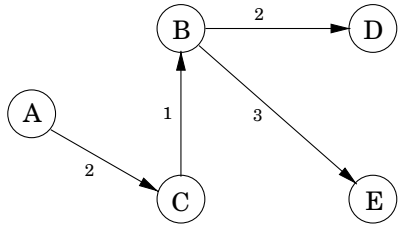

| | |
|---|---|
| A: 0 | D: 6 |
| B: 3 | E: 7 |
| C: 2 | |




| | |
|---|---|
| A: 0 | D: 5 |
| B: 3 | E: 6 |
| C: 2 | |




| | |
|---|---|
| A: 0 | D: 5 |
| B: 3 | E: 6 |
| C: 2 | |

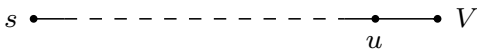


doesn't depend upon BFS at all. We now start from scratch with this complementary interpretation.

### 4.4.2 An alternative derivation

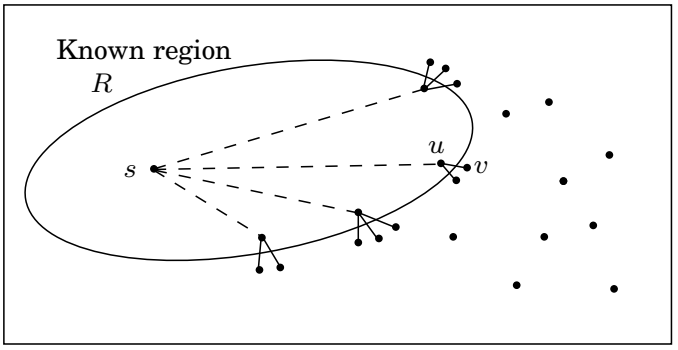
Here's a plan for computing shortest paths: expand outward from the starting point $s$, steadily growing the region of the graph to which distances and shortest paths are known. This growth should be orderly, first incorporating the closest nodes and then moving on to those further away. More precisely, when the "known region" is some subset of vertices $R$ that includes $s$, the next addition to it should be *the node outside $R$ that is closest to $s$*. Let us call this node $v$; the question is: how do we identify it?

To answer, consider $u$, the node just before $v$ in the shortest path from $s$ to $v$:




*Since we are assuming that all edge lengths are positive*, $u$ must be closer to $s$ than $v$ is. This means that $u$ is in $R$—otherwise it would contradict $v$'s status as the closest node to $s$ outside $R$. So, the shortest path from $s$ to $v$ is simply *a known shortest path extended by a single edge.*

**Figure 4.10** Single-edge extensions of known shortest paths.




But there will typically be many single-edge extensions of the currently known shortest paths (Figure 4.10); which of these identifies $v$? The answer is, *the shortest of these extended paths*. Because, if an even shorter single-edge-extended path existed, this would once more contradict $v$'s status as the node outside $R$ closest to $s$. So, it's easy to find $v$: it is the node outside $R$ for which the smallest value of distance$(s, u) + l(u, v)$ is attained, as $u$ ranges over $R$. In other words, *try all single-edge extensions of the currently known shortest paths, find the shortest such extended path, and proclaim its endpoint to be the next node of $R$.*

We now have an algorithm for growing $R$ by looking at extensions of the current set of shortest paths. Some extra efficiency comes from noticing that on any given

iteration, the only *new* extensions are those involving the node most recently added to region $R$. All other extensions will have been assessed previously and do not need to be recomputed. In the following pseudocode, `dist`$(v)$ is the length of the currently shortest single-edge-extended path leading to $v$; it is $\infty$ for nodes not adjacent to $R$.

```
Initialize dist(s) to 0, other dist(·) values to ∞
R = { } (the "known region")
while R ≠ V:
   Pick the node v ∉ R with smallest dist(·)
   Add v to R
   for all edges (v, z) ∈ E:
       if dist(z) > dist(v) + l(v, z):
          dist(z) = dist(v) + l(v, z)
```

Incorporating priority queue operations gives us back Dijkstra's algorithm (Figure 4.8).

To justify this algorithm formally, we would use a proof by induction, as with breadth-first search. Here's an appropriate inductive hypothesis.

> *At the end of each iteration of the while loop, the following conditions hold: (1) there is a value $d$ such that all nodes in $R$ are at distance $\leq d$ from $s$ and all nodes outside $R$ are at distance $\geq d$ from $s$, and (2) for every node $u$, the value* `dist`$(u)$ *is the length of the shortest path from $s$ to $u$ whose intermediate nodes are constrained to be in $R$ (if no such path exists, the value is $\infty$).*

The base case is straightforward (with $d = 0$), and the details of the inductive step can be filled in from the preceding discussion.

### 4.4.3 Running time

At the level of abstraction of Figure 4.8, Dijkstra's algorithm is structurally identical to breadth-first search. However, it is slower because the priority queue primitives are computationally more demanding than the constant-time `eject`'s and `inject`'s of BFS. Since `makequeue` takes at most as long as $|V|$ `insert` operations, we get a total of $|V|$ `deletemin` and $|V| + |E|$ `insert/decreasekey` operations. The time needed for these varies by implementation; for instance, a binary heap gives an overall running time of $O((|V| + |E|) \log |V|)$.

## 4.5 Priority queue implementations

### 4.5.1 Array

The simplest implementation of a priority queue is as an unordered array of key values for all potential elements (the vertices of the graph, in the case of Dijkstra's algorithm). Initially, these values are set to $\infty$.

### Which heap is best?

The running time of Dijkstra's algorithm depends heavily on the priority queue implementation used. Here are the typical choices.

| Implementation | `deletemin` | `insert/` `decreasekey` | $\lvert V\rvert \times$ `deletemin` + $(\lvert V\rvert + \lvert E\rvert) \times$ `insert` |
|---|---|---|---|
| Array | $O(\lvert V\rvert)$ | $O(1)$ | $O(\lvert V\rvert^2)$ |
| Binary heap | $O(\log \lvert V\rvert)$ | $O(\log \lvert V\rvert)$ | $O((\lvert V\rvert + \lvert E\rvert)\log \lvert V\rvert)$ |
| $d$-ary heap | $O\left(\frac{d \log \lvert V\rvert}{\log d}\right)$ | $O\left(\frac{\log \lvert V\rvert}{\log d}\right)$ | $O\left((\lvert V\rvert \cdot d + \lvert E\rvert)\frac{\log \lvert V\rvert}{\log d}\right)$ |
| Fibonacci heap | $O(\log \lvert V\rvert)$ | $O(1)$ (amortized) | $O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$ |

So for instance, even a naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|)\log |V|)$. Which is preferable?

This depends on whether the graph is *sparse* (has few edges) or *dense* (has lots of them). For all graphs, $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then clearly the array implementation is the faster. On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $|V|^2/\log |V|$.

The $d$-ary heap is a generalization of the binary heap (which corresponds to $d = 2$) and leads to a running time that is a function of $d$. The optimal choice is $d \approx |E|/|V|$; in other words, to optimize we must set the degree of the heap to be equal to the *average degree* of the graph. This works well for both sparse and dense graphs. For very sparse graphs, in which $|E| = O(|V|)$, the running time is $O(|V|\log |V|)$, as good as with a binary heap. For dense graphs, $|E| = \Omega(|V|^2)$ and the running time is $O(|V|^2)$, as good as with a linked list. Finally, for graphs with intermediate density $|E| = |V|^{1+\delta}$, the running time is $O(|E|)$, linear!

The last line in the table gives running times using a sophisticated data structure called a *Fibonacci heap*. Although its efficiency is impressive, this data structure requires considerably more work to implement than the others, and this tends to dampen its appeal in practice. We will say little about it except to mention a curious feature of its time bounds. Its *insert* operations take varying amounts of time but are guaranteed to *average* $O(1)$ over the course of the algorithm. In such situations (one of which we shall encounter in Chapter 5) we say that the *amortized* cost of heap *insert*'s is $O(1)$.

An `insert` or `decreasekey` is fast, because it just involves adjusting a key value, an $O(1)$ operation. To `deletemin`, on the other hand, requires a linear-time scan of the list.

## 4.5.2 Binary heap

Here elements are stored in a *complete* binary tree, namely, a binary tree in which each level is filled in from left to right, and must be full before the next level

is started. In addition, a special ordering constraint is enforced: *the key value of any node of the tree is less than or equal to that of its children*. In particular, therefore, the root always contains the smallest element. See Figure 4.11(a) for an example.

To `insert`, place the new element at the bottom of the tree (in the first available position), and let it "bubble up." That is, if it is smaller than its parent, swap the two and repeat (Figure 4.11(b)–(d)). The number of swaps is at most the height of the tree, which is $\lfloor \log_2 n \rfloor$ when there are $n$ elements. A `decreasekey` is similar, except that the element is already in the tree, so we let it bubble up from its current position.

To `deletemin`, return the root value. To then remove this element from the heap, take the last node in the tree (in the rightmost position in the bottom row) and place it at the root. Let it "sift down": if it is bigger than either child, swap it with the smaller child and repeat (Figure 4.11(e)–(g)). Again this takes $O(\log n)$ time.

The regularity of a complete binary tree makes it easy to represent using an array. The tree nodes have a natural ordering: row by row, starting at the root and moving left to right within each row. If there are $n$ nodes, this ordering specifies their positions $1, 2, \ldots, n$ within the array. Moving up and down the tree is easily simulated on the array, using the fact that node number $j$ has parent $\lfloor j/2 \rfloor$ and children $2j$ and $2j+1$ (Exercise 4.16).

### 4.5.3 *d*-ary heap

A $d$-ary heap is identical to a binary heap, except that nodes have $d$ children instead of just two. This reduces the height of a tree with $n$ elements to $\Theta(\log_d n) = \Theta((\log n)/(\log d))$. Inserts are therefore speeded up by a factor of $\Theta(\log d)$. Deletemin operations, however, take a little longer, namely $O(d \log_d n)$ (do you see why?).

The array representation of a binary heap is easily extended to the $d$-ary case. This time, node number $j$ has parent $\lceil (j-1)/d \rceil$ and children $\{(j-1)d+2, \ldots, \min\{n, (j-1)d+d+1\}\}$ (Exercise 4.16).

## 4.6 Shortest paths in the presence of negative edges

### 4.6.1 Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point $s$ to any node $v$ must pass exclusively through nodes that are closer than $v$. This no longer holds when edge lengths can be negative. In Figure 4.12, the shortest path from $S$ to $A$ passes through $B$, a node that is further away!

What needs to be changed in order to accommodate this new complication? To answer this, let's take a particular high-level view of Dijkstra's algorithm. A crucial invariant is that the `dist` values it maintains are always either overestimates or exactly correct. They start off at $\infty$, and the only way they ever change is by updating

**Figure 4.11** (a) A binary heap with 10 elements. Only the key values are shown. (b)–(d) The intermediate "bubble-up" steps in inserting an element with key 7. (e)–(g) The "sift-down" steps in a delete-min operation.

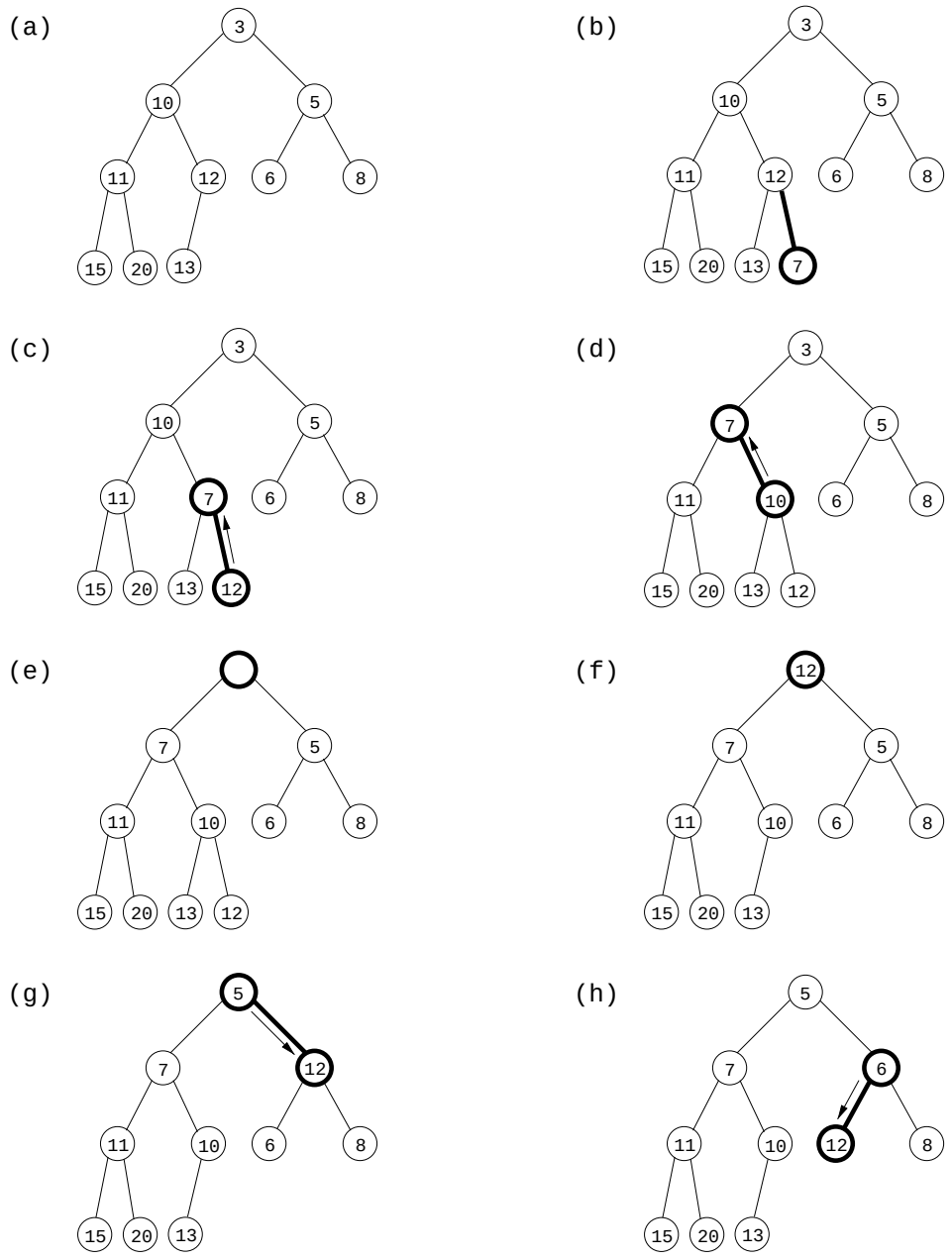
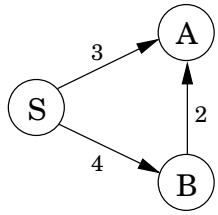


**Figure 4.12** Dijkstra's algorithm will not work if there are negative edges.



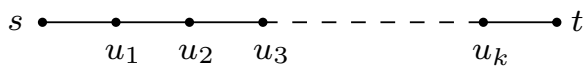

along an edge:

```
procedure update((u, v) ∈ E)
dist(v) = min{dist(v), dist(u) + l(u, v)}
```

This *update* operation is simply an expression of the fact that the distance to $v$ cannot possibly be more than the distance to $u$, plus $l(u, v)$. It has the following properties.

1. It gives the correct distance to $v$ in the particular case where $u$ is the second-last node in the shortest path to $v$, and `dist`$(u)$ is correctly set.
2. It will never make `dist`$(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

This operation is extremely useful: it is harmless, and if used carefully, will correctly set distances. In fact, Dijkstra's algorithm can be thought of simply as a sequence of `update`'s. We know this particular sequence doesn't work with negative edges, but is there some other sequence that does? To get a sense of the properties this sequence must possess, let's pick a node $t$ and look at the shortest path to it from $s$.



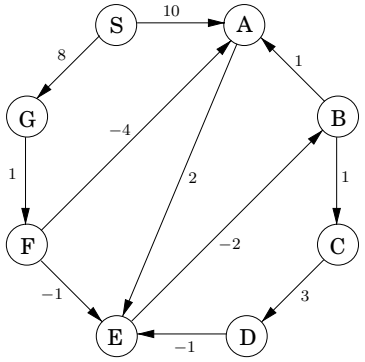

This path can have at most $|V| - 1$ edges (do you see why?). If the sequence of updates performed includes $(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t)$, *in that order* (though not necessarily consecutively), then by the first property the distance to $t$ will be correctly computed. It doesn't matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are *safe*.

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order? Here is an easy solution: simply update *all* the edges, $|V| - 1$ times! The resulting $O(|V| \cdot |E|)$ procedure is called the Bellman-Ford algorithm and is shown in Figure 4.13, with an example run in Figure 4.14.

**Figure 4.13** The Bellman-Ford algorithm for single-source shortest paths in general graphs.

```
procedure shortest-paths(G, l, s)
Input: Directed graph G = (V, E);
       edge lengths {l_e : e ∈ E} with no negative cycles;
       vertex s ∈ V
Output:For all vertices u reachable from s, dist(u) is set
       to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

**Figure 4.14** The Bellman-Ford algorithm illustrated on a sample graph.




| | Iteration | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Node | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | ∞ | 10 | 10 | 5 | 5 | 5 | 5 | 5 |
| B | ∞ | ∞ | ∞ | 10 | 6 | 5 | 5 | 5 |
| C | ∞ | ∞ | ∞ | ∞ | 11 | 7 | 6 | 6 |
| D | ∞ | ∞ | ∞ | ∞ | ∞ | 14 | 10 | 9 |
| E | ∞ | ∞ | 12 | 8 | 7 | 7 | 7 | 7 |
| F | ∞ | ∞ | 9 | 9 | 9 | 9 | 9 | 9 |
| G | ∞ | 8 | 8 | 8 | 8 | 8 | 8 | 8 |

A note about implementation: for many graphs, the maximum number of edges in any shortest path is substantially less than $|V| - 1$, with the result that fewer rounds of updates are needed. Therefore, it makes sense to add an extra check to the shortest-path algorithm, to make it terminate immediately after any round in which no update occurred.

### 4.6.2 Negative cycles

If the length of edge $(E, B)$ in Figure 4.14 were changed to $-4$, the graph would have a *negative cycle* $A \to E \to B \to A$. In such situations, it doesn't make sense

to even ask about shortest paths. There is a path of length 2 from $A$ to $E$. But going round the cycle, there's also a path of length 1, and going round multiple times, we find paths of lengths $0, -1, -2$, and so on.

The shortest-path problem is ill-posed in graphs with negative cycles. As might be expected, our algorithm from Section 4.6.1 works only in the absence of such cycles. But where did this assumption appear in the derivation of the algorithm? Well, it slipped in when we asserted the *existence* of a shortest path from $s$ to $t$.

Fortunately, it is easy to automatically detect negative cycles and issue a warning. Such a cycle would allow us to endlessly apply rounds of `update` operations, reducing `dist` estimates every time. So instead of stopping after $|V| - 1$ iterations, perform one extra round. There is a negative cycle if and only if some `dist` value is reduced during this final round.

## 4.7 Shortest paths in dags

There are two subclasses of graphs that automatically exclude the possibility of negative cycles: graphs without negative edges, and graphs without cycles. We already know how to efficiently handle the former. We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence. The key source of efficiency is that

*In any path of a dag, the vertices appear in increasing linearized order.*

**Figure 4.15** A single-source shortest-path algorithm for directed acyclic graphs.

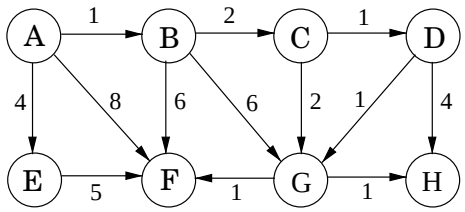
```
procedure dag-shortest-paths(G, l, s)
Input: Dag G = (V, E);
       edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
       to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
      update(u, v)
```

Therefore, it is enough to linearize (that is, topologically sort) the dag by depth-first search, and then visit the vertices in sorted order, updating the edges out of each. The algorithm is given in Figure 4.15.

Notice that our scheme doesn't require edges to be positive. In particular, we can find *longest paths* in a dag by the same algorithm: just negate all edge lengths.

## Exercises

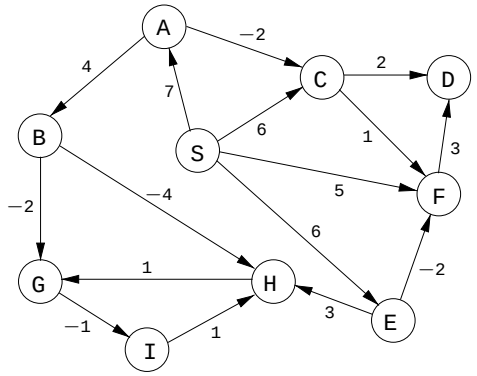
4.1. Suppose Dijkstra's algorithm is run on the following graph, starting at node $A$.




(a) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

(b) Show the final shortest-path tree.

4.2. Just like the previous problem, but this time with the Bellman-Ford algorithm.




4.3. *Squares.* Design and analyze an algorithm that takes as input an undirected graph $G = (V, E)$ and determines whether $G$ contains a simple cycle (that is, a cycle which doesn't intersect itself) of length four. Its running time should be at most $O(|V|^3)$.

You may assume that the input graph is represented either as an adjacency matrix or with adjacency lists, whichever makes your algorithm simpler.

4.4. Here's a proposal for how to find the length of the shortest cycle in an undirected graph with unit edge lengths.

When a back edge, say $(v, w)$, is encountered during a depth-first search, it forms a cycle with the tree edges from $w$ to $v$. The length of the cycle is

level[$v$] − level[$w$] + 1, where the level of a vertex is its distance in the DFS tree from the root vertex. This suggests the following algorithm:

- *Do a depth-first search, keeping track of the level of each vertex.*
- *Each time a back edge is encountered, compute the cycle length and save it if it is smaller than the shortest one previously seen.*

Show that this strategy does not always work by providing a counterexample as well as a brief (one or two sentence) explanation.

4.5. Often there are multiple shortest paths between two nodes of a graph. Give a linear-time algorithm for the following task.

*Input:* Undirected graph $G = (V, E)$ with unit edge lengths; nodes $u, v \in V$.
*Output:* The number of distinct shortest paths from $u$ to $v$.

4.6. Prove that for the array `prev` computed by Dijkstra's algorithm, the edges $\{u, \texttt{prev}[u]\}$ (for all $u \in V$) form a tree.

4.7. You are given a directed graph $G = (V, E)$ with (possibly negative) weighted edges, along with a specific node $s \in V$ and a tree $T = (V, E')$, $E' \subseteq E$. Give an algorithm that checks whether $T$ is a shortest-path tree for $G$ with starting point $s$. Your algorithm should run in linear time.

4.8. Professor F. Lake suggests the following algorithm for finding the shortest path from node $s$ to node $t$ in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node $s$, and return the shortest path found to node $t$.

Is this a valid method? Either prove that it works correctly, or give a counterexample.

4.9. Consider a directed graph in which the only negative edges are those that leave $s$; all other edges are positive. Can Dijkstra's algorithm, started at $s$, fail on such a graph? Prove your answer.

4.10. You are given a directed graph with (possibly negative) weighted edges, in which the shortest path between any two vertices is guaranteed to have at most $k$ edges. Give an algorithm that finds the shortest path between two vertices $u$ and $v$ in $O(k|E|)$ time.

4.11. Give an algorithm that takes as input a directed graph with positive edge lengths, and returns the length of the shortest cycle in the graph (if the graph is acyclic, it should say so). Your algorithm should take time at most $O(|V|^3)$.

4.12. Give an $O(|V|^2)$ algorithm for the following task.

*Input:* An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; an edge $e \in E$.
*Output:* The length of the shortest cycle containing edge $e$.

4.13. You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph $G = (V, E)$. Each stretch of highway $e \in E$ connects two of the cities, and you know its length in miles, $l_e$. You want to get

from city $s$ to city $t$. There's one problem: your car can only hold enough gas to cover $L$ miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length $l_e \leq L$.

(a) Given the limitation on your car's fuel tank capacity, show how to determine in linear time whether there is a feasible route from $s$ to $t$.

(b) You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from $s$ to $t$. Give an $O((|V| + |E|) \log |V|)$ algorithm to determine this.

4.14. You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between *all pairs of nodes*, with the one restriction that these paths must all pass through $v_0$.

4.15. Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V| + |E|) \log |V|)$ time.

*Input:* An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.
*Output:* A Boolean array `usp`$[\cdot]$: for each node $u$, the entry `usp`$[u]$ should be `true` if and only if there is a *unique* shortest path from $s$ to $u$. (Note: `usp`$[s]$ = `true`.)

4.16. Section 4.5.2 describes a way of storing a complete binary tree of $n$ nodes in an array indexed by $1, 2, \ldots, n$.

(a) Consider the node at position $j$ of the array. Show that its parent is at position $\lfloor j/2 \rfloor$ and its children are at $2j$ and $2j + 1$ (if these numbers are $\leq n$).

(b) What are the corresponding indices when a complete $d$-ary tree is stored in an array?

Figure 4.16 shows pseudocode for a binary heap, modeled on an exposition by R. E. Tarjan.[2] The heap is stored as an array $h$, which is assumed to support two constant-time operations:

- $|h|$, which returns the number of elements currently in the array;
- $h^{-1}$, which returns the position of an element within the array.

The latter can always be achieved by maintaining the values of $h^{-1}$ as an auxiliary array.

(c) Show that the `makeheap` procedure takes $O(n)$ time when called on a set of $n$ elements. What is the worst-case input? (*Hint:* Start by showing that the running time is at most $\sum_{i=1}^{n} \log(n/i)$.)

(d) What needs to be changed to adapt this pseudocode to $d$-ary heaps?

[2] See: R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.

**Figure 4.16** Operations on a binary heap.

```
procedure insert(h, x)
bubbleup(h, x, |h| + 1)

procedure decreasekey(h, x)
bubbleup(h, x, h^{-1}(x))

function deletemin(h)
if |h| = 0:
    return null
else:
    x = h(1)
    siftdown(h, h(|h|), 1)
    return x

function makeheap(S)
h = empty array of size |S|
for x ∈ S:
    h(|h| + 1) = x
for i = |S| downto 1:
    siftdown(h, h(i), i)
return h

procedure bubbleup(h, x, i)
(place element x in position i of h, and let it bubble up)
p = ⌈i/2⌉
while i ≠ 1 and key(h(p)) > key(x):
    h(i) = h(p);  i = p;  p = ⌈i/2⌉
h(i) = x

procedure siftdown(h, x, i)
(place element x in position i of h, and let it sift down)
c = minchild(h, i)
while c ≠ 0 and key(h(c)) < key(x):
    h(i) = h(c);  i = c;  c = minchild(h, i)
h(i) = x

function minchild(h, i)
(return the index of the smallest child of h(i))
if 2i > |h|:
    return 0 (no children)
else:
    return arg min{key(h(j)) : 2i ≤ j ≤ min{|h|, 2i + 1}}
```

4.17. Suppose we want to run Dijkstra's algorithm on a graph whose edge weights are integers in the range $0, 1, \ldots, W$, where $W$ is a relatively small number.

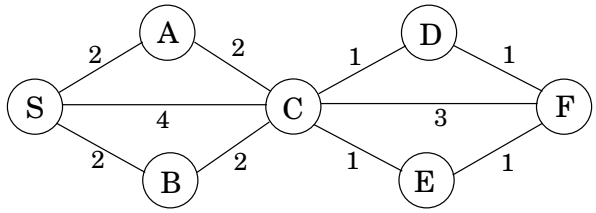
(a) Show how Dijkstra's algorithm can be made to run in time $O(W|V| + |E|)$.

(b) Show an alternative implementation that takes time just $O((|V| + |E|) \log W)$.

4.18. In cases where there are several different shortest paths between two nodes (and edges have varying lengths), the most convenient of these paths is often *the one with fewest edges*. For instance, if nodes represent cities and edge lengths represent costs of flying between cities, there might be many ways to get from city $s$ to city $t$ which all have the same cost. The most convenient of these alternatives is the one which involves the fewest stopovers. Accordingly, for a specific starting node $s$, define

$$\texttt{best}[u] = \text{minimum number of edges in a shortest path from } s \text{ to } u.$$

In the example below, the `best` values for nodes $S, A, B, C, D, E, F$ are $0, 1, 1, 1, 2, 2, 3$, respectively.




Give an efficient algorithm for the following problem.

*Input:* Graph $G = (V, E)$; positive edge lengths $l_e$; starting node $s \in V$.
*Output:* The values of `best`$[u]$ should be set for *all* nodes $u \in V$.

4.19. *Generalized shortest-paths problem.* In Internet routing, there are delays on lines but also, more significantly, delays at routers. This motivates a generalized shortest-paths problem.

Suppose that in addition to having edge lengths $\{l_e : e \in E\}$, a graph also has *vertex costs* $\{c_v : v \in V\}$. Now define the cost of a path to be the sum of its edge lengths, *plus* the costs of all vertices on the path (including the endpoints). Give an efficient algorithm for the following problem.

*Input:* A directed graph $G = (V, E)$; positive edge lengths $l_e$ and positive vertex costs $c_v$; a starting vertex $s \in V$.
*Output:* An array `cost`$[\cdot]$ such that for every vertex $u$, `cost`$[u]$ is the least cost of any path from $s$ to $u$ (i.e., the cost of the cheapest path), under the definition above.

Notice that `cost`$[s] = c_s$.

4.20. There is a network of roads $G = (V, E)$ connecting a set of cities $V$. Each road in $E$ has an associated length $l_e$. There is a proposal to add one new road to this

network, and there is a list $E'$ of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length. As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network $G$ would result in the maximum decrease in the driving distance between two fixed cities $s$ and $t$ in the network. Give an efficient algorithm for solving this problem.

4.21. Shortest path algorithms can be applied in currency trading. Let $c_1, c_2, \ldots, c_n$ be various currencies; for instance, $c_1$ might be dollars, $c_2$ pounds, and $c_3$ lire. For any two currencies $c_i$ and $c_j$, there is an exchange rate $r_{i,j}$; this means that you can purchase $r_{i,j}$ units of currency $c_j$ in exchange for one unit of $c_i$. These exchange rates satisfy the condition that $r_{i,j} \cdot r_{j,i} < 1$, so that if you start with a unit of currency $c_i$, change it into currency $c_j$ and then convert back to currency $c_i$, you end up with less than one unit of currency $c_i$ (the difference is the cost of the transaction).

(a) Give an efficient algorithm for the following problem: Given a set of exchange rates $r_{i,j}$, and two currencies $s$ and $t$, find the most advantageous sequence of currency exchanges for converting currency $s$ into currency $t$. Toward this goal, you should represent the currencies and rates by a graph whose edge lengths are real numbers.

The exchange rates are updated frequently, reflecting the demand and supply of the various currencies. Occasionally the exchange rates satisfy the following property: there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. This means that by starting with a unit of currency $c_{i_1}$ and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$, and finally back to $c_{i_1}$, you would end up with more than one unit of currency $c_{i_1}$. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for risk-free profits.

(b) Give an efficient algorithm for detecting the presence of such an anomaly. Use the graph representation you found above.

4.22. *The tramp steamer problem.* You are the owner of a steamship that can ply between a group of port cities $V$. You make money at each port: a visit to city $i$ earns you a profit of $p_i$ dollars. Meanwhile, the transportation cost from port $i$ to port $j$ is $c_{ij} > 0$. You want to find a cyclic route in which the ratio of profit to cost is maximized.

To this end, consider a directed graph $G = (V, E)$ whose nodes are ports, and which has edges between each pair of ports. For any cycle $C$ in this graph, the profit-to-cost ratio is

$$r(C) = \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}}.$$

Let $r^*$ be the maximum ratio achievable by a simple cycle. One way to determine $r^*$ is by binary search: by first guessing some ratio $r$, and then testing whether it is too large or too small.

Consider any positive $r > 0$. Give each edge $(i, j)$ a weight of $w_{ij} = rc_{ij} - p_j$.

(a) Show that if there is a cycle of negative weight, then $r < r^*$.

(b) Show that if all cycles in the graph have strictly positive weight, then $r > r^*$.

(c) Give an efficient algorithm that takes as input a desired accuracy $\epsilon > 0$ and returns a simple cycle $C$ for which $r(C) \geq r^* - \epsilon$. Justify the correctness of your algorithm and analyze its running time in terms of $|V|$, $\epsilon$, and $R = \max_{(i,j)\in E}(p_j/c_{ij})$.

*Chapter 5*

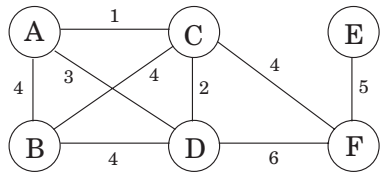
# Greedy algorithms

A game like chess can be won only by *thinking ahead*: a player who is focused entirely on immediate advantage is easy to defeat. But in many other games, such as Scrabble, it is possible to do quite well by simply making whichever move seems best at the moment and not worrying too much about future consequences.

This sort of myopic behavior is easy and convenient, making it an attractive algorithmic strategy. *Greedy* algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal. Our first example is that of minimum spanning trees.

## 5.1 Minimum spanning trees

Suppose you are asked to network a collection of computers by linking selected pairs of them. This translates into a graph problem in which nodes are computers, undirected edges are potential links, and the goal is to pick enough of these edges that the nodes are connected. But this is not all; each link also has a maintenance cost, reflected in that edge's weight. What is the cheapest possible network?




One immediate observation is that the optimal set of edges cannot contain a cycle, because removing an edge from this cycle would reduce the cost without compromising connectivity:

**Property 1** *Removing a cycle edge cannot disconnect a graph.*

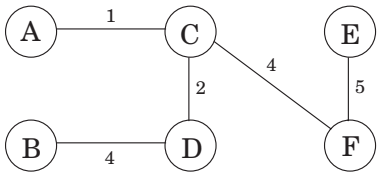
So the solution must be connected and acyclic: undirected graphs of this kind are called *trees*. The particular tree we want is the one with minimum total weight, known as the *minimum spanning tree*. Here is its formal definition.

*Input:* An undirected graph $G = (V, E)$; edge weights $w_e$.

*Output:* A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

In the preceding example, the minimum spanning tree has a cost of 16:




However, this is not the only optimal solution. Can you spot another?

### 5.1.1 A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from $E$ according to the following rule.

*Repeatedly add the next lightest edge that doesn't produce a cycle.*

In other words, it constructs the tree edge by edge and, apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment. This is a *greedy* algorithm: every decision it makes is the one with the most obvious immediate advantage.

Figure 5.1 shows an example. We start with an empty graph and then attempt to add edges in increasing order of weight (ties are broken arbitrarily):

$$B-C,\ C-D,\ B-D,\ C-F,\ D-F,\ E-F,\ A-D,\ A-B,\ C-E,\ A-C.$$

The first two succeed, but the third, $B-D$, would produce a cycle if added. So we ignore it and move along. The final result is a tree with cost 14, the minimum possible.

**Figure 5.1** The minimum spanning tree found by Kruskal's algorithm.

## Trees

A *tree* is an undirected graph that is connected and acyclic. Much of what makes trees so useful is the simplicity of their structure. For instance,

**Property 2** *A tree on $n$ nodes has $n - 1$ edges.*

This can be seen by building the tree one edge at a time, starting from an empty graph. Initially each of the $n$ nodes is disconnected from the others, in a connected component by itself. As edges are added, these components merge. Since each edge unites two different components, exactly $n - 1$ edges are added by the time the tree is fully formed.

In a little more detail: When a particular edge $\{u, v\}$ comes up, we can be sure that $u$ and $v$ lie in separate connected components, for otherwise there would already be a path between them and this edge would create a cycle. Adding the edge then merges these two components, thereby reducing the total number of connected components by one. Over the course of this incremental process, the number of components decreases from $n$ to one, meaning that $n - 1$ edges must have been added along the way.

The converse is also true.

**Property 3** *Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.*

We just need to show that $G$ is acyclic. One way to do this is to run the following iterative procedure on it: while the graph contains a cycle, remove one edge from this cycle. The process terminates with some graph $G' = (V, E')$, $E' \subseteq E$, which is acyclic and, by Property 1 (from page 127), is also connected. Therefore $G'$ is a tree, whereupon $|E'| = |V| - 1$ by Property 2. So $E' = E$, no edges were removed, and $G$ was acyclic to start with.

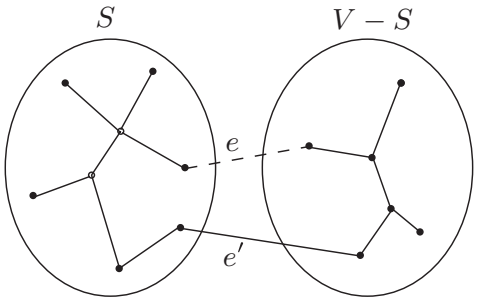
In other words, we can tell whether a connected graph is a tree just by counting how many edges it has. Here's another characterization.

**Property 4** *An undirected graph is a tree if and only if there is a unique path between any pair of nodes.*

In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

On the other hand, if a graph has a path between any two nodes, then it is connected. If these paths are unique, then the graph is also acyclic (since a cycle has two paths between any pair of nodes).

**Figure 5.2** $T \cup \{e\}$. The addition of $e$ (dotted) to $T$ (solid lines) produces a cycle. This cycle must contain at least one other edge, shown here as $e'$, across the cut $(S, V - S)$.




The correctness of Kruskal's method follows from a certain *cut property*, which is general enough to also justify a whole slew of other minimum spanning tree algorithms.

### 5.1.2 The cut property

Say that in the process of building a minimum spanning tree (MST), we have already chosen some edges and are so far on the right track. Which edge should we add next? The following lemma gives us a lot of flexibility in our choice.

**Cut property** *Suppose edges $X$ are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes $S$ for which $X$ does not cross between $S$ and $V - S$, and let $e$ be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.*

A *cut* is any partition of the vertices into two groups, $S$ and $V - S$. What this property says is that it is always safe to add the lightest edge across any cut (that is, between a vertex in $S$ and one in $V - S$), provided $X$ has no edges across the cut.

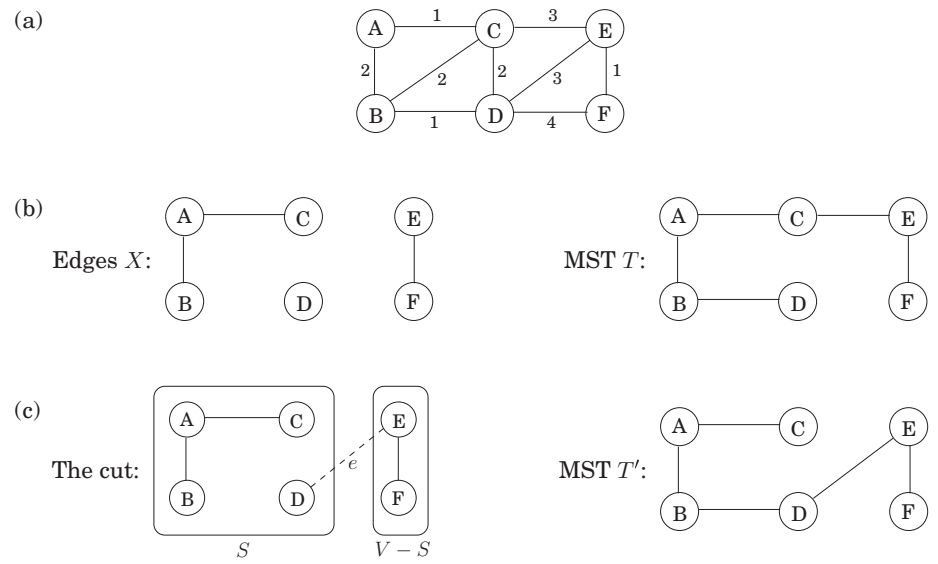
Let's see why this holds. Edges $X$ are part of some MST $T$; if the new edge $e$ also happens to be part of $T$, then there is nothing to prove. So assume $e$ is not in $T$. We will construct a different MST $T'$ containing $X \cup \{e\}$ by altering $T$ slightly, changing just one of its edges.

Add edge $e$ to $T$. Since $T$ is connected, it already has a path between the endpoints of $e$, so adding $e$ creates a cycle. This cycle must also have some other edge $e'$ across the cut $(S, V - S)$ (Figure 5.2). If we now remove this edge, we are left with $T' = T \cup \{e\} - \{e'\}$, which we will show to be a tree. $T'$ is connected by Property 1, since $e'$ is a cycle edge. And it has the same number of edges as $T$; so by Properties 2 and 3, it is also a tree.

Moreover, $T'$ is a minimum spanning tree. Compare its weight to that of $T$:

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

**Figure 5.3** The cut property at work. (a) An undirected graph. (b) Set $X$ has three edges, and is part of the MST $T$ on the right. (c) If $S = \{A, B, C, D\}$, then one of the minimum-weight edges across the cut $(S, V - S)$ is $e = \{D, E\}$. $X \cup \{e\}$ is part of MST $T'$, shown on the right.




Both $e$ and $e'$ cross between $S$ and $V - S$, and $e$ is specifically the lightest edge of this type. Therefore $w(e) \leq w(e')$, and $\text{weight}(T') \leq \text{weight}(T)$. Since $T$ is an MST, it must be the case that $\text{weight}(T') = \text{weight}(T)$ and that $T'$ is also an MST.

Figure 5.3 shows an example of the cut property. Which edge is $e'$?

### 5.1.3 Kruskal's algorithm

We are ready to justify Kruskal's algorithm. At any given moment, the edges it has already chosen form a partial solution, a collection of connected components each of which has a tree structure. The next edge $e$ to be added connects two of these components; call them $T_1$ and $T_2$. Since $e$ is the lightest edge that doesn't produce a cycle, it is certain to be the lightest edge between $T_1$ and $V - T_1$ and therefore satisfies the cut property.

Now we fill in some implementation details. At each stage, the algorithm chooses an edge to add to its current partial solution. To do so, it needs to test each candidate edge $u - v$ to see whether the endpoints $u$ and $v$ lie in different components; otherwise the edge produces a cycle. And once an edge is chosen, the corresponding components need to be merged. What kind of data structure supports such operations?

**Figure 5.4** Kruskal's minimum spanning tree algorithm.

```
procedure kruskal (G, w)
Input:  A connected undirected graph G = (V, E) with edge weights w_e
output: A minimum spanning three defined by the edges X

for all u ∈ V:
   makeset (u)

X = {}
sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

We will model the algorithm's state as a collection of *disjoint sets*, each of which contains the nodes of a particular component. Initially each node is in a component by itself:

`makeset`$(x)$: create a singleton set containing just $x$.

We repeatedly test pairs of nodes to see if they belong to the same set.

`find`$(x)$: to which set does $x$ belong?

And whenever we add an edge, we are merging two components.

`union`$(x, y)$: merge the sets containing $x$ and $y$.

The final algorithm is shown in Figure 5.4. It uses $|V|$ `makeset`, $2|E|$ `find`, and $|V| - 1$ `union` operations.

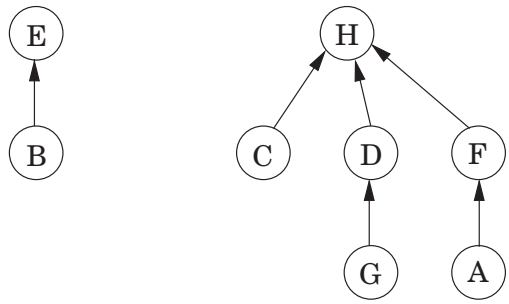
### 5.1.4 A data structure for disjoint sets

**Union by rank:**
One way to store a set is as a directed tree (Figure 5.5). Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a convenient *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

In addition to a parent pointer $\pi$, each node also has a *rank* that, for the time being, should be interpreted as the height of the subtree hanging from that node.

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x):  x = π(x)
return x
```

**Figure 5.5** A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.




As can be expected, `makeset` is a constant-time operation. On the other hand, `find` follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree. The tree actually gets built via the third operation, `union`, and so we must make sure that this procedure keeps trees shallow.

Merging two sets is easy: make the root of one point to the root of the other. But we have a choice here. If the representatives (roots) of the sets are $r_x$ and $r_y$, do we make $r_x$ point to $r_y$ or the other way around? Since tree height is the main impediment to computational efficiency, a good strategy is to *make the root of the shorter tree point to the root of the taller tree*. This way, the overall height increases only if the two trees being merged are equally tall. Instead of explicitly computing heights of trees, we will use the *rank* numbers of their root nodes—which is why this scheme is called *union by rank*.

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y): rank(r_y) = rank(r_y) + 1
```

See Figure 5.6 for an example.

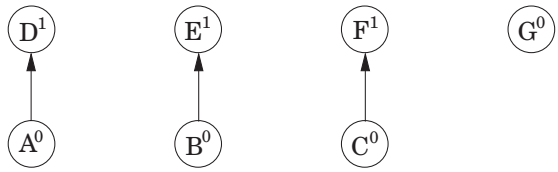
By design, the *rank* of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the *rank* values along the way are strictly increasing.

**Property 1** *For any $x$, $rank(x) < rank(\pi(x))$.*

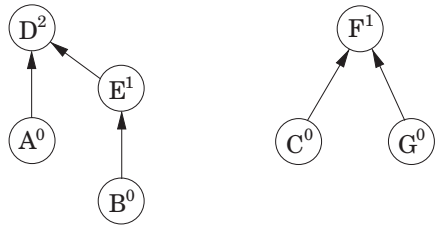
**Figure 5.6** A sequence of disjoint-set operations. Superscripts denote rank.

After `makeset`($A$),`makeset`($B$),...,`makeset`($G$):

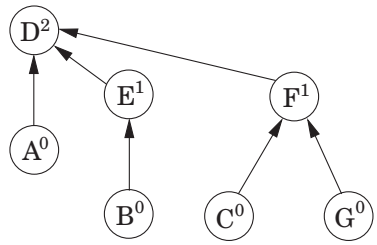

After `union`($A$,$D$),`union`($B$,$E$),`union`($C$,$F$):




After `union`($C$,$G$),`union`($E$,$A$):




After `union`($B$,$G$):




A root node with rank $k$ is created by the merger of two trees with roots of rank $k-1$. It follows by induction (try it!) that

**Property 2** *Any root node of rank k has at least $2^k$ nodes in its tree.*

This extends to internal (nonroot) nodes as well: a node of rank $k$ has at least $2^k$ descendants. After all, any internal node was once a root, and neither its rank nor its set of descendants has changed since then. Moreover, different rank-$k$ nodes

cannot have common descendants, since by Property 1 any element has at most one ancestor of rank $k$. Which means

**Property 3** *If there are $n$ elements overall, there can be at most $n/2^k$ nodes of rank $k$.*

This last observation implies, crucially, that the maximum rank is $\log n$. Therefore, all the trees have height $\leq \log n$, and this is an upper bound on the running time of `find` and `union`.

**Path compression:**
With the data structure as presented so far, the total time for Kruskal's algorithm becomes $O(|E| \log |V|)$ for sorting the edges (remember, $\log |E| \approx \log |V|$) plus another $O(|E| \log |V|)$ for the `union` and `find` operations that dominate the rest of the algorithm. So there seems to be little incentive to make our data structure any more efficient.

But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time? Then the data structure part becomes the bottleneck, and it is useful to think about improving its performance beyond $\log n$ per operation. As it turns out, the improved data structure is useful in many other applications.

But how can we perform `union`'s and `find`'s faster than $\log n$? The answer is, by being a little more careful to maintain our data structure in good shape. As any housekeeper knows, a little extra effort put into routine maintenance can pay off handsomely in the long run, by forestalling major calamities. We have in mind a particular maintenance operation for our union-find data structure, intended to keep the trees short—during each `find`, when a series of parent pointers is followed up to the root of a tree, we will change all these pointers so that they point directly to the root (Figure 5.7). This *path compression* heuristic only slightly increases the time needed for a `find` and is easy to code.

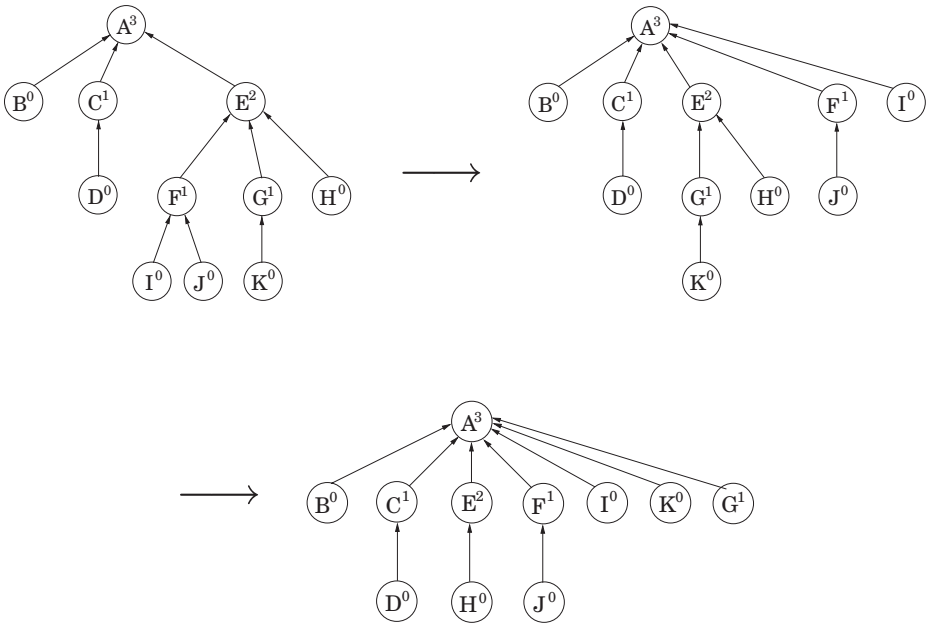
```
function find(x)
if x ≠ π(x): π(x) = find(π(x))
return π(x)
```

The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis: we need to look at *sequences* of `find` and `union` operations, starting from an empty data structure, and determine the average time per operation. This *amortized cost* turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Think of the data structure as having a "top level'' consisting of the root nodes, and below it, the insides of the trees. There is a division of labor: `find` operations (with or without path compression) only touch the insides of trees, whereas `union`'s only look at the top level. Thus path compression has no effect on `union` operations and leaves the top level unchanged.

We now know that the ranks of root nodes are unaltered, but what about *nonroot* nodes? The key point here is that once a node ceases to be a root, it never resurfaces,

**Figure 5.7** The effect of path compression: `find`($I$) followed by `find`($K$).




and its rank is forever fixed. Therefore the ranks of all nodes are unchanged by path compression, even though these numbers can no longer be interpreted as tree heights. In particular, properties 1–3 (from page 133) still hold.

If there are $n$ elements, their rank values can range from 0 to $\log n$ by Property 3. Let's divide the nonzero part of this range into certain carefully chosen intervals, for reasons that will soon become clear:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \ldots, 16\}, \{17, 18, \ldots, 2^{16} = 65536\}, \{65537, 65538, \ldots, 2^{65536}\}, \ldots$$

Each group is of the form $\{k+1, k+2, \ldots, 2^k\}$, where $k$ is a power of 2. The number of groups is $\log^* n$, which is defined to be the number of successive log operations that need to be applied to $n$ to bring it down to 1 (or below 1). For instance, $\log^* 1000 = 4$ since $\log\log\log\log 1000 \leq 1$. In practice there will just be the first five of the intervals shown; more are needed only if $n \geq 2^{65536}$, in other words never.

In a sequence of `find` operations, some may take longer than others. We'll bound the overall running time using some creative accounting. Specifically, we will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars. We will then show that each `find` takes $O(\log^* n)$ steps, plus some additional amount of time that can be "paid for'' using the pocket money

of the nodes involved—one dollar per unit of time. Thus the overall time for $m$ `find`'s is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

In more detail, a node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If this rank lies in the interval $\{k+1, \ldots, 2^k\}$, the node receives $2^k$ dollars. By Property 3, the number of nodes with rank $> k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}.$$

Therefore the total money given to nodes in this particular interval is at most $n$ dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $\leq n \log^* n$.

Now, the time taken by a specific `find` is simply the number of pointers followed. Consider the ascending rank values along this chain of nodes up to the root. Nodes $x$ on the chain fall into two categories: either the rank of $\pi(x)$ is in a higher interval than the rank of $x$, or else it lies in the same interval. There are at most $\log^* n$ nodes of the first type (do you see why?), so the work done on them takes $O(\log^* n)$ time. The remaining nodes—whose parents' ranks are in the same interval as theirs—have to pay a dollar out of their pocket money for their processing time.

This only works if the initial allowance of each node $x$ is enough to cover all of its payments in the sequence of `find` operations. Here's the crucial observation: each time $x$ pays a dollar, its parent changes to one of higher rank. Therefore, if $x$'s rank lies in the interval $\{k+1, \ldots, 2^k\}$, it has to pay at most $2^k$ dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.

### 5.1.5 Prim's algorithm

Let's return to our discussion of minimum spanning tree algorithms. What the cut property tells us in most general terms is that any algorithm conforming to the following greedy schema is guaranteed to work.

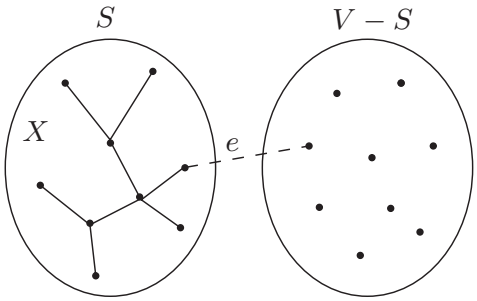
```
X = { } (edges picked so far)
repeat until |X| = |V| − 1:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```

A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges $X$ always forms a subtree, and $S$ is chosen to be the set of this tree's vertices.

On each iteration, the subtree defined by $X$ *grows* by one edge, namely, the lightest edge between a vertex in $S$ and a vertex outside $S$ (Figure 5.8). We can equivalently think of $S$ as growing to include the vertex $v \notin S$ of smallest `cost`:

$$\texttt{cost}(v) = \min_{u \in S} w(u, v).$$

**Figure 5.8** Prim's algorithm: the edges $X$ form a tree, and $S$ consists of its vertices.




This is strongly reminiscent of Dijkstra's algorithm, and in fact the pseudocode (Figure 5.9) is almost identical. The only difference is in the key values by which the priority queue is ordered. In Prim's algorithm, the value of a node is the weight of the lightest incoming edge from set $S$, whereas in Dijkstra's it is the length of an entire path to that node from the starting point. Nonetheless, the two algorithms are similar enough that they have the same running time, which depends on the particular priority queue implementation.

Figure 5.9 shows Prim's algorithm at work, on a small six-node graph. Notice how the final MST is completely specified by the `prev` array.

## 5.2 Huffman encoding

In the MP3 audio compression scheme, a sound signal is encoded in three steps.

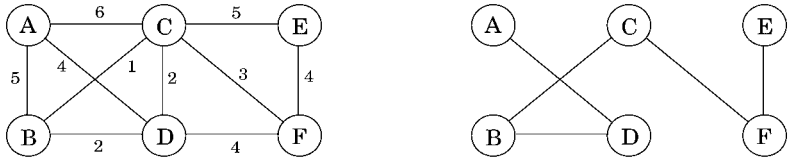
1. It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, at a rate of 44,100 samples per second, a 50-minute symphony would correspond to $T = 50 \times 60 \times 44{,}100 \approx 130$ million measurements.[1]
2. Each real-valued sample $s_t$ is *quantized*: approximated by a nearby number from a finite set $\Gamma$. This set is carefully chosen to exploit human perceptual limitations, with the intention that the approximating sequence is indistinguishable from $s_1, s_2, \ldots, s_T$ by the human ear.
3. The resulting string of length $T$ over alphabet $\Gamma$ is encoded in binary.

It is in the last step that Huffman encoding is used. To understand its role, let's look at a toy example in which $T$ is 130 million and the alphabet $\Gamma$ consists of just four

[1] For stereo sound, two channels would be needed, doubling the number of samples.

**Figure 5.9** *Top:* Prim's minimum spanning tree algorithm. *Below:* An illustration of Prim's algorithm, starting at node $A$. Also shown are a table of `cost/prev` values, and the final MST.

```
procedure prim (G, w)
Input:  A connected undirected graph G = (V, E) with edge
        weights w_e
output: A minimum spanning tree defined by the array prev

for all u ∈ V:
   cost(u) = ∞
   prev(u) = nil
pick any initial node u_0
cost(u_0) = 0

H = makequeue (V)  (priority queue, using cost-values as keys)
while H is not empty:
    v = deletemin (H)
    for each {v, z} ∈ E:
       if cost(z) > w(v, z):
          cost(z) > w(v, z)
          prev(z) = v
```




| Set $S$ | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ |
|---|---|---|---|---|---|---|
| $\{\}$ | $0$/nil | $\infty$/nil | $\infty$/nil | $\infty$/nil | $\infty$/nil | $\infty$/nil |
| $A$ | | $5/A$ | $6/A$ | $4/A$ | $\infty$/nil | $\infty$/nil |
| $A, D$ | | $2/D$ | $2/D$ | | $\infty$/nil | $4/D$ |
| $A, D, B$ | | | $1/B$ | | $\infty$/nil | $4/D$ |
| $A, D, B, C$ | | | | | $5/C$ | $3/C$ |
| $A, D, B, C, F$ | | | | | $4/F$ | |

values, denoted by the symbols $A, B, C, D$. What is the most economical way to write this long string in binary? The obvious choice is to use 2 bits per symbol—say codeword 00 for $A$, 01 for $B$, 10 for $C$, and 11 for $D$—in which case 260 megabits are needed in total. Can there possibly be a better encoding than this?

## A randomized algorithm for minimum cut

We have already seen that spanning trees and cuts are intimately related. Here is another connection. Let's remove the last edge that Kruskal's algorithm adds to the spanning tree; this breaks the tree into two components, thus defining a cut $(S, \overline{S})$ in the graph. What can we say about this cut? Suppose the graph we were working with was unweighted, and that its edges were ordered uniformly at random for Kruskal's algorithm to process them. Here is a remarkable fact: with probability at least $1/n^2$, $(S, \overline{S})$ is the minimum cut in the graph, where the size of a cut $(S, \overline{S})$ is the number of edges crossing between $S$ and $\overline{S}$. This means that repeating the process $O(n^2)$ times and outputting the smallest cut found yields the minimum cut in $G$ with high probability: an $O(mn^2 \log n)$ algorithm for unweighted minimum cuts. Some further tuning gives the $O(n^2 \log n)$ minimum cut algorithm, invented by David Karger, which is the fastest known algorithm for this important problem.

So let us see why the cut found in each iteration is the minimum cut with probability at least $1/n^2$. At any stage of Kruskal's algorithm, the vertex set $V$ is partitioned into connected components. The only edges eligible to be added to the tree have their two endpoints in distinct components. The number of edges incident to each component must be at least $C$, the size of the minimum cut in $G$ (since we could consider a cut that separated this component from the rest of the graph). So if there are $k$ components in the graph, the number of eligible edges is at least $kC/2$ (each of the $k$ components has at least $C$ edges leading out of it, and we need to compensate for the double-counting of each edge). Since the edges were randomly ordered, the chance that the next eligible edge in the list is from the minimum cut is at most $C/(kC/2) = 2/k$. Thus, with probability at least $1 - 2/k = (k-2)/k$, the choice leaves the minimum cut intact. But now the chance that Kruskal's algorithm leaves the minimum cut intact all the way up to the choice of the last spanning tree edge is at least

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{1}{n(n-1)}.$$

In search of inspiration, we take a closer look at our particular sequence and find that the four symbols are not equally abundant.

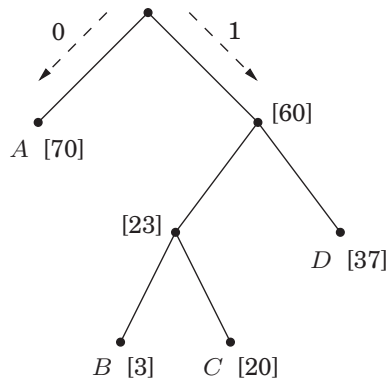
| Symbol | Frequency |
|---|---|
| $A$ | 70 million |
| $B$ | 3 million |
| $C$ | 20 million |
| $D$ | 37 million |

Is there some sort of *variable-length encoding*, in which just *one* bit is used for the frequently occurring symbol $A$, possibly at the expense of needing three or more bits for less common symbols?

A danger with having codewords of different lengths is that the resulting encoding may not be uniquely decipherable. For instance, if the codewords are $\{0, 01, 11, 001\}$,

**Figure 5.10** A prefix-free encoding. Frequencies are shown in square brackets.

| Symbol | Codeword |
|---|---|
| $A$ | 0 |
| $B$ | 100 |
| $C$ | 101 |
| $D$ | 11 |




the decoding of strings like 001 is ambiguous. We will avoid this problem by insisting on the *prefix-free* property: no codeword can be a prefix of another codeword.

Any prefix-free encoding can be represented by a *full* binary tree—that is, a binary tree in which every node has either zero or two children—where the symbols are at the leaves, and where each codeword is generated by a path from root to leaf, interpreting left as 0 and right as 1 (Exercise 5.29). Figure 5.10 shows an example of such an encoding for the four symbols $A, B, C, D$. Decoding is unique: a string of bits is decrypted by starting at the root, reading the string from left to right to move downward, and, whenever a leaf is reached, outputting the corresponding symbol and returning to the root. It is a simple scheme and pays off nicely for our toy example, where (under the codes of Figure 5.10) the total size of the binary string drops to 213 megabits, a 17% improvement.

In general, how do we find the optimal coding tree, given the frequencies $f_1, f_2, \ldots, f_n$ of $n$ symbols? To make the problem precise, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding,

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of } i\text{th symbol in tree})$$

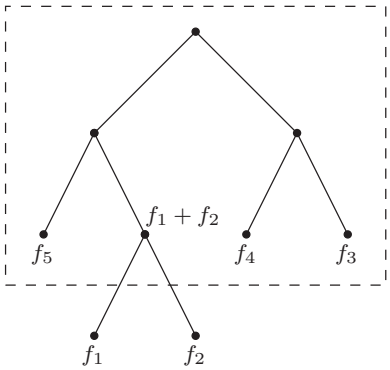
(the number of bits required for a symbol is exactly its depth in the tree).

There is another way to write this cost function that is very helpful. Although we are only given frequencies for the leaves, we can define the frequency of any *internal* node to be the sum of the frequencies of its descendant leaves; this is, after all, the number of times the internal node is visited during encoding or decoding. During the encoding process, each time we move down the tree, one bit gets output for every nonroot node through which we pass. So the total cost—the total number of bits which are output—can also be expressed thus:

*The cost of a tree is the sum of the frequencies of all leaves and internal nodes, except the root.*

The first formulation of the cost function tells us that *the two symbols with the smallest frequencies must be at the bottom of the optimal tree*, as children of the lowest internal node (this internal node has two children since the tree is *full*). Otherwise, swapping these two symbols with whatever is lowest in the tree would improve the encoding.

This suggests that we start constructing the tree *greedily*: find the two symbols with the smallest frequencies, say $i$ and $j$, and make them children of a new node, which then has frequency $f_i + f_j$. To keep the notation simple, let's just assume these are $f_1$ and $f_2$. By the second formulation of the cost function, any tree in which $f_1$ and $f_2$ are sibling-leaves has cost $f_1 + f_2$ plus the cost for a tree with $n-1$ leaves of frequencies $(f_1 + f_2)$, $f_3$, $f_4, \ldots, f_n$:




The latter problem is just a smaller version of the one we started with. So we pull $f_1$ and $f_2$ off the list of frequencies, insert $(f_1 + f_2)$, and loop. The resulting algorithm can be described in terms of priority queue operations (as defined on page 109) and takes $O(n \log n)$ time if a binary heap (Section 4.5.2) is used.

```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output: An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n: insert(H, i)
for k = n+1 to 2n−1:
   i = deletemin(H),  j = deletemin(H)
   create a node numbered k with children i, j
   f[k] = f[i] + f[j]
   insert(H, k)
```

Returning to our toy example: can you tell if the tree of Figure 5.10 is optimal?

## Entropy

The annual county horse race is bringing in three thoroughbreds who have never competed against one another. Excited, you study their past 200 races and summarize these as probability distributions over four outcomes: `first` ("first place"), `second`, `third`, and `other`.

| Outcome | Aurora | Whirlwind | Phantasm |
|---|---|---|---|
| `first` | 0.15 | 0.30 | 0.20 |
| `second` | 0.10 | 0.05 | 0.30 |
| `third` | 0.70 | 0.25 | 0.30 |
| `other` | 0.05 | 0.40 | 0.20 |

Which horse is the most predictable? One quantitative approach to this question is to look at *compressibility*. Write down the history of each horse as a string of 200 values (`first`, `second`, `third`, `other`). The total number of bits needed to encode these track-record strings can then be computed using Huffman's algorithm. This works out to 290 bits for Aurora, 380 for Whirlwind, and 420 for Phantasm (check it!). Aurora has the shortest encoding and is therefore in a strong sense the most predictable.

The inherent unpredictability, or *randomness*, of a probability distribution can be measured by the extent to which it is possible to compress data drawn from that distribution.

$$\text{more compressible} \equiv \text{less random} \equiv \text{more predictable}$$

Suppose there are $n$ possible outcomes, with probabilities $p_1, p_2, \ldots, p_n$. If a sequence of $m$ values is drawn from the distribution, then the $i$th outcome will pop up roughly $mp_i$ times (if $m$ is large). For simplicity, assume these are exactly the observed frequencies, and moreover that the $p_i$'s are all powers of 2 (that is, of the form $1/2^k$). It can be seen by induction (Exercise 5.19) that the number of bits needed to encode the sequence is $\sum_{i=1}^{n} mp_i \log(1/p_i)$. Thus the average number of bits needed to encode a single draw from the distribution is

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i}.$$

This is the *entropy* of the distribution, a measure of how much randomness it contains.

For example, a fair coin has two outcomes, each with probability 1/2. So its entropy is

$$\tfrac{1}{2}\log 2 + \tfrac{1}{2}\log 2 = 1.$$

This is natural enough: the coin flip contains one bit of randomness. But what if the coin is not fair, if it has a 3/4 chance of turning up heads? Then the entropy is

$$\tfrac{3}{4}\log\tfrac{4}{3} + \tfrac{1}{4}\log 4 = 0.81.$$

A biased coin is more predictable than a fair coin, and thus has lower entropy. As the bias becomes more pronounced, the entropy drops toward zero.

We explore these notions further in Exercises 5.18 and 5.19.

## 5.3 Horn formulas

In order to display human-level intelligence, a computer must be able to perform at least some modicum of logical reasoning. Horn formulas are a particular framework for doing this, for expressing logical facts and deriving conclusions.

The most primitive object in a Horn formula is a *Boolean variable*, taking value either `true` or `false`. For instance, variables $x$, $y$, and $z$ might denote the following possibilities.

$x \equiv$ the murder took place in the kitchen
$y \equiv$ the butler is innocent
$z \equiv$ the colonel was asleep at 8 pm

A *literal* is either a variable $x$ or its negation $\overline{x}$ ("NOT $x$"). In Horn formulas, knowledge about variables is represented by two kinds of *clauses*:

1. *Implications*, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal. These express statements of the form "if the conditions on the left hold, then the one on the right must also be true." For instance,

   $$(z \wedge w) \Rightarrow u$$

   might mean "if the colonel was asleep at 8 pm and the murder took place at 8 pm then the colonel is innocent." A degenerate type of implication is the *singleton* "$\Rightarrow x$," meaning simply that $x$ is `true`: "the murder definitely occurred in the kitchen."
2. Pure *negative clauses*, consisting of an OR of any number of negative literals, as in

   $$(\overline{u} \vee \overline{v} \vee \overline{y})$$

   ("they can't all be innocent").

Given a set of clauses of these two types, the goal is to determine whether there is a consistent explanation: an assignment of `true`/`false` values to the variables that satisfies all the clauses. This is also called a *satisfying assignment*.

The two kinds of clauses pull us in different directions. The implications tell us to set some of the variables to `true`, while the negative clauses encourage us to make them `false`. Our strategy for solving a Horn formula is this: We start with all variables `false`. We then proceed to set some of them to `true`, one by one, but very reluctantly, and only if we absolutely have to because an implication would otherwise be violated. Once we are done with this phase and all implications are satisfied, only then do we turn to the negative clauses and make sure they are all satisfied.

In other words, our algorithm for Horn clauses is the following greedy scheme (*stingy* is perhaps more descriptive):

```
Input:  a Horn formula
Output: a satisfying assignment, if one exists

set all variables to false

while there is an implication that is not satisfied:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied:
  return the assignment
else: return "formula is not satisfiable"
```

For instance, suppose the formula is

$$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\overline{w} \vee \overline{x} \vee \overline{y}), \quad (\overline{z}).$$

We start with everything `false` and then notice that $x$ must be `true` on account of the singleton implication $\Rightarrow x$. Then we see that $y$ must also be `true`, because of $x \Rightarrow y$. And so on.

To see why the algorithm is correct, notice that if it returns an assignment, this assignment satisfies both the implications and the negative clauses, and so it is indeed a satisfying truth assignment of the input Horn formula. So we only have to convince ourselves that if the algorithm finds no satisfying assignment, then there really is none. This is so because our "stingy" rule maintains the following invariant:

> *If a certain set of variables is set to* `true`*, then they must be* `true` *in* any *satisfying assignment.*

Hence, if the truth assignment found after the *while* loop does not satisfy the negative clauses, there can be no satisfying truth assignment.

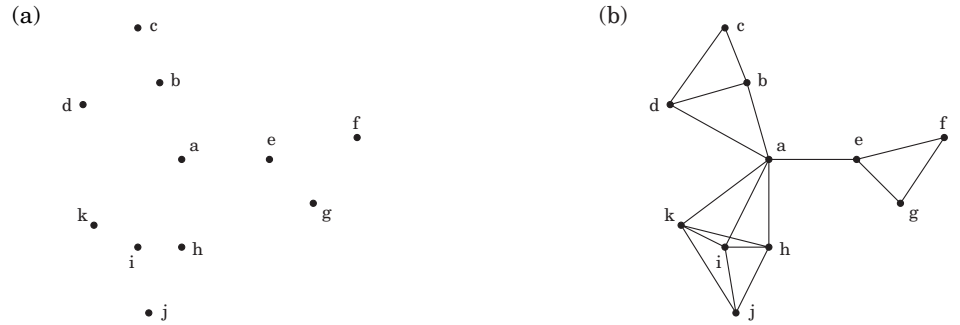
Horn formulas lie at the heart of Prolog ("programming by logic"), a language in which you program by specifying desired properties of the output, using simple logical expressions. The workhorse of Prolog interpreters is our greedy satisfiability algorithm. Conveniently, it can be implemented in time linear in the length of the formula; do you see how (Exercise 5.33)?

## 5.4 Set cover

The dots in Figure 5.11 represent a collection of towns. This county is in its early stages of planning and is deciding where to put schools. There are only two constraints: each school should be in a town, and no one should have to travel more than 30 miles to reach one of them. What is the minimum number of schools needed?

This is a typical *set cover* problem. For each town $x$, let $S_x$ be the set of towns within 30 miles of it. A school at $x$ will essentially "cover" these other towns. The question

**Figure 5.11** (a) Eleven towns. (b) Towns that are within 30 miles of each other.




is then, how many sets $S_x$ must be picked in order to cover all the towns in the county?

**Set Cover**

*Input:* A set of elements $B$; sets $S_1, \ldots, S_m \subseteq B$

*Output:* A selection of the $S_i$ whose union is $B$.

*Cost:* Number of sets picked.

(In our example, the elements of $B$ are the towns.) This problem lends itself immediately to a greedy solution:

Repeat until all elements of $B$ are covered:

Pick the set $S_i$ with the largest number of uncovered elements.

This is extremely natural and intuitive. Let's see what it would do on our earlier example: It would first place a school at town $a$, since this covers the largest number of other towns. Thereafter, it would choose three more schools—$c$, $j$, and either $f$ or $g$—for a total of four. However, there exists a solution with just three schools, at $b$, $e$, and $i$. The greedy scheme is not optimal!

But luckily, it isn't too far from optimal.

**Claim** *Suppose $B$ contains $n$ elements and that the optimal cover consists of $k$ sets. Then the greedy algorithm will use at most $k \ln n$ sets.*[2]

Let $n_t$ be the number of elements still not covered after $t$ iterations of the greedy algorithm (so $n_0 = n$). Since these remaining elements are covered by the optimal

[2] ln means "natural logarithm," that is, to the base $e$.

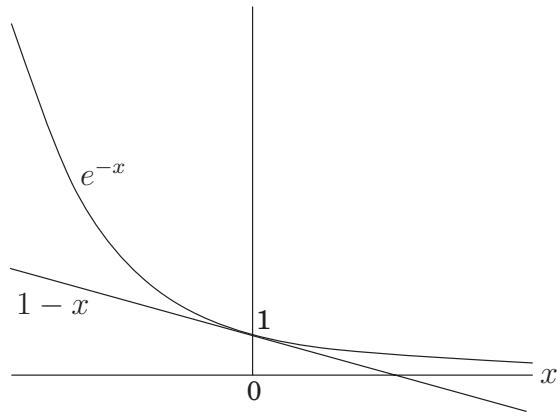
$k$ sets, there must be some set with at least $n_t/k$ of them. Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t\left(1 - \frac{1}{k}\right),$$

which by repeated application implies $n_t \leq n_0(1 - 1/k)^t$. A more convenient bound can be obtained from the useful inequality

$$1 - x \leq e^{-x} \text{ for all } x, \text{ with equality if and only if } x = 0,$$

which is most easily proved by a picture:




Thus

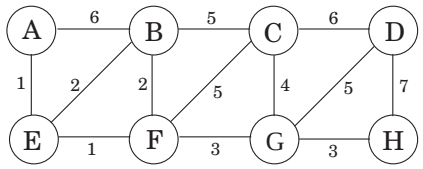
$$n_t \leq n_0\left(1 - \frac{1}{k}\right)^t < n_0(e^{-1/k})^t = ne^{-t/k}$$

At $t = k \ln n$, therefore, $n_t$ is strictly less than $ne^{-\ln n} = 1$, which means no elements remain to be covered.

The ratio between the greedy algorithm's solution and the optimal solution varies from input to input but is always less than $\ln n$. And there are certain inputs for which the ratio is very close to $\ln n$ (Exercise 5.34). We call this maximum ratio the *approximation factor* of the greedy algorithm. There seems to be a lot of room for improvement, but in fact such hopes are unjustified: it turns out that under certain widely-held complexity assumptions (which will be clearer when we reach Chapter 8), there is provably no polynomial-time algorithm with a smaller approximation factor.

## Exercises

5.1. Consider the following graph.

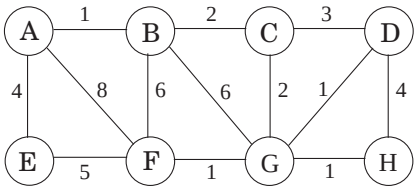



(a) What is the cost of its minimum spanning tree?

(b) How many minimum spanning trees does it have?

(c) Suppose Kruskal's algorithm is run on this graph. In what order are the edges added to the MST? For each edge in this sequence, give a cut that justifies its addition.

5.2. Suppose we want to find the minimum spanning tree of the following graph.




(a) Run Prim's algorithm; whenever there is a choice of nodes, always use alphabetic ordering (e.g., start from node $A$). Draw a table showing the intermediate values of the `cost` array.

(b) Run Kruskal's algorithm on the same graph. Show how the disjoint-sets data structure looks at every intermediate stage (including the structure of the directed trees), assuming path compression is used.

5.3. Design a linear-time algorithm for the following task.

*Input:* A connected, undirected graph $G$.

*Question:* Is there an edge you can remove from $G$ while still leaving $G$ connected?

Can you reduce the running time of your algorithm to $O(|V|)$?

5.4. Show that if an undirected graph with $n$ vertices has $k$ connected components, then it has at least $n - k$ edges.

5.5. Consider an undirected graph $G = (V, E)$ with nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of $G$, and that you have also computed shortest paths to all nodes from a particular node $s \in V$. Now suppose each edge weight is increased by 1: the new weights are $w'_e = w_e + 1$.

(a) Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

(b) Do the shortest paths change? Give an example where they change or prove they cannot change.

5.6. Let $G = (V, E)$ be an undirected graph. Prove that if all its edge weights are distinct, then it has a unique minimum spanning tree.

5.7. Show how to find the *maximum* spanning tree of a graph, that is, the spanning tree of largest total weight.

5.8. Suppose you are given a connected weighted graph $G = (V, E)$ with a distinguished vertex $s$ and where all edge weights are positive and distinct. Is it possible for a tree of shortest paths from $s$ and a minimum spanning tree in $G$ to not share any edges? If so, give an example. If not, give a reason.

5.9. The following statements may or may not be correct. In each case, either prove it (if it is correct) or give a counterexample (if it isn't correct). Always assume that the graph $G = (V, E)$ is undirected and connected. Do not assume that edge weights are distinct unless this is specifically stated.

(a) If graph $G$ has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a minimum spanning tree.

(b) If $G$ has a cycle with a unique heaviest edge $e$, then $e$ cannot be part of any MST.

(c) Let $e$ be any edge of minimum weight in $G$. Then $e$ must be part of some MST.

(d) If the lightest edge in a graph is unique, then it must be part of every MST.

(e) If $e$ is part of some MST of $G$, then it must be a lightest edge across some cut of $G$.

(f) If $G$ has a cycle with a unique lightest edge $e$, then $e$ must be part of every MST.

(g) The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST.

(h) The shortest path between two nodes is necessarily part of some MST.

(i) Prim's algorithm works correctly when there are negative edges.

(j) (For any $r > 0$, define an *r-path* to be a path whose edges all have weight $< r$.) If $G$ contains an $r$-path from node $s$ to $t$, then every MST of $G$ must also contain an $r$-path from node $s$ to node $t$.

5.10. Let $T$ be an MST of graph $G$. Given a connected subgraph $H$ of $G$, show that $T \cap H$ is contained in some MST of $H$.

5.11. Give the state of the disjoint-sets data structure after the following sequence of operations, starting from singleton sets $\{1\}, \ldots, \{8\}$. Use path compression. In case of ties, always make the lower numbered root point to the higher numbered one.

```
union(1, 2), union(3, 4), union(5, 6), union(7, 8), union(1, 4),
    union(6, 7), union(4, 5), find(1)
```

5.12. Suppose you implement the disjoint-sets data structure using union-by-rank but not path compression. Give a sequence of $m$ `union` and `find` operations on $n$ elements that take $\Omega(m \log n)$ time.

5.13. A long string consists of the four characters $A, C, G, T$; they appear with frequency 31%, 20%, 9%, and 40%, respectively. What is the Huffman encoding of these four characters?

5.14. Suppose the symbols $a, b, c, d, e$ occur with frequencies 1/2, 1/4, 1/8, 1/16, 1/16, respectively.

(a) What is the Huffman encoding of the alphabet?

(b) If this encoding is applied to a file consisting of 1,000,000 characters with the given frequencies, what is the length of the encoded file in bits?

5.15. We use Huffman's algorithm to obtain an encoding of alphabet $\{a, b, c\}$ with frequencies $f_a$, $f_b$, $f_c$. In each of the following cases, either give an example of frequencies ($f_a$, $f_b$, $f_c$) that would yield the specified code, or explain why the code cannot possibly be obtained (no matter what the frequencies are).

(a) Code: {0, 10, 11}

(b) Code: {0, 1, 00}

(c) Code: {10, 01, 00}

5.16. Prove the following two properties of the Huffman encoding scheme.

(a) If some character occurs with frequency more than 2/5, then there is guaranteed to be a codeword of length 1.

(b) If all characters occur with frequency less than 1/3, then there is guaranteed to be no codeword of length 1.

5.17. Under a Huffman encoding of $n$ symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case.

5.18. The following table gives the frequencies of the letters of the English language (including the blank for separating words) in a particular corpus.

| | | | | | |
|---|---|---|---|---|---|
| blank | 18.3% | r | 4.8% | y | 1.6% |
| e | 10.2% | d | 3.5% | p | 1.6% |
| t | 7.7% | l | 3.4% | b | 1.3% |
| a | 6.8% | c | 2.6% | v | 0.9% |
| o | 5.9% | u | 2.4% | k | 0.6% |
| i | 5.8% | m | 2.1% | j | 0.2% |
| n | 5.5% | w | 1.9% | x | 0.2% |
| s | 5.1% | f | 1.8% | q | 0.1% |
| h | 4.9% | g | 1.7% | z | 0.1% |

(a) What is the optimum Huffman encoding of this alphabet?

(b) What is the expected number of bits per letter?

(c) Suppose now that we calculate the entropy of these frequencies

$$H = \sum_{i=0}^{26} p_i \log \frac{1}{p_i}$$

(see the box in page 143). Would you expect it to be larger or smaller than your answer above? Explain.

(d) Do you think that this is the limit of how much English text can be compressed? What features of the English language, besides letters and their frequencies, should a better compression scheme take into account?

5.19. *Entropy.* Consider a distribution over $n$ possible outcomes, with probabilities $p_1, p_2, \ldots, p_n$.

(a) Just for this part of the problem, assume that each $p_i$ is a power of 2 (that is, of the form $1/2^k$). Suppose a long sequence of $m$ samples is drawn from the distribution and that for all $1 \leq i \leq n$, the $i^{\text{th}}$ outcome occurs exactly $mp_i$ times in the sequence. Show that if Huffman encoding is applied to this sequence, the resulting encoding will have length

$$\sum_{i=1}^{n} mp_i \log \frac{1}{p_i}.$$

(b) Now consider arbitrary distributions—that is, the probabilities $p_i$ are not restricted to powers of 2. The most commonly used measure of the *amount of randomness* in the distribution is the *entropy*

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i}.$$

For what distribution (over $n$ outcomes) is the entropy the largest possible? The smallest possible?

5.20. Give a linear-time algorithm that takes as input a tree and determines whether it has a *perfect matching*: a set of edges that touches each node exactly once.

5.21. A *feedback edge set* of an undirected graph $G = (V, E)$ is a subset of edges $E' \subseteq E$ that intersects every cycle of the graph. Thus, removing the edges $E'$ will render the graph acyclic.

Give an efficient algorithm for the following problem:

*Input:* Undirected graph $G = (V, E)$ with positive edge weights $w_e$
*Output:* A feedback edge set $E' \subseteq E$ of minimum total weight $\sum_{e \in E'} w_e$

5.22. In this problem, we will develop a new algorithm for finding minimum spanning trees. It is based upon the following property:

Pick any cycle in the graph, and let $e$ be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain $e$.

(a) Prove this property carefully.

(b) Here is the new MST algorithm. The input is some undirected graph $G = (V, E)$ (in adjacency list format) with edge weights $\{w_e\}$.

```
sort the edges according to their weights
for each edge e ∈ E, in decreasing order of w_e:
   if e is part of a cycle of G:
      G = G − e (that is, remove e from G)
return G
```

Prove that this algorithm is correct.

(c) On each iteration, the algorithm must check whether there is a cycle containing a specific edge $e$. Give a linear-time algorithm for this task, and justify its correctness.

(d) What is the overall time taken by this algorithm, in terms of $|E|$?

5.23. You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume $G$ and $T$ are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree $T$ to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give a linear-time algorithm for updating the tree.

(a) $e \notin E'$ and $\hat{w}(e) > w(e)$.

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$.

(c) $e \in E'$ and $\hat{w}(e) < w(e)$.

(d) $e \in E'$ and $\hat{w}(e) > w(e)$.

5.24. Sometimes we want light spanning trees with certain special properties. Here's an example.

*Input:* Undirected graph $G = (V, E)$; edge weights $w_e$; subset of vertices $U \subset V$
*Output:* The lightest spanning tree in which the nodes of $U$ are leaves (there might be other leaves in this tree as well).

(The answer isn't necessarily a minimum spanning tree.)

Give an algorithm for this problem which runs in $O(|E| \log |V|)$ time. (*Hint:* When you remove nodes $U$ from the optimal solution, what is left?)

5.25. A binary counter of unspecified length supports two operations: `increment` (which increases its value by one) and `reset` (which sets its value back to zero). Show that, starting from an initially zero counter, any sequence of $n$ `increment` and `reset` operations takes time $O(n)$; that is, the amortized time per operation is $O(1)$.

5.26. Here's a problem that occurs in automatic program analysis. For a set of variables $x_1, \ldots, x_n$, you are given some *equality* constraints, of the form

"$x_i = x_j$" and some *disequality* constraints, of the form "$x_i \neq x_j$." Is it possible to satisfy all of them?

For instance, the constraints

$$x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4$$

cannot be satisfied. Give an efficient algorithm that takes as input $m$ constraints over $n$ variables and decides whether the constraints can be satisfied.

5.27. *Graphs with prescribed degree sequences.* Given a list of $n$ positive integers $d_1, d_2, \ldots, d_n$, we want to efficiently determine whether there exists an undirected graph $G = (V, E)$ whose nodes have degrees precisely $d_1, d_2, \ldots, d_n$. That is, if $V = \{v_1, \ldots, v_n\}$, then the degree of $v_i$ should be exactly $d_i$. We call $(d_1, \ldots, d_n)$ the *degree sequence of* $G$. This graph $G$ should not contain self-loops (edges with both endpoints equal to the same node) or multiple edges between the same pair of nodes.

(a) Give an example of $d_1, d_2, d_3, d_4$ where all the $d_i \leq 3$ and $d_1 + d_2 + d_3 + d_4$ is even, but for which no graph with degree sequence $(d_1, d_2, d_3, d_4)$ exists.

(b) Suppose that $d_1 \geq d_2 \geq \cdots \geq d_n$ and that there exists a graph $G = (V, E)$ with degree sequence $(d_1, \ldots, d_n)$. We want to show that there must exist a graph that has this degree sequence and where in addition the neighbors of $v_1$ are $v_2, v_3, \ldots, v_{d_1+1}$. The idea is to gradually transform $G$ into a graph with the desired additional property.

i. Suppose the neighbors of $v_1$ in $G$ are not $v_2, v_3, \ldots, v_{d_1+1}$. Show that there exists $i < j \leq n$ and $u \in V$ such that $\{v_1, v_i\}, \{u, v_j\} \notin E$ and $\{v_1, v_j\}, \{u, v_i\} \in E$.

ii. Specify the changes you would make to $G$ to obtain a new graph $G' = (V, E')$ with the same degree sequence as $G$ and where $(v_1, v_i) \in E'$.

iii. Now show that there must be a graph with the given degree sequence but in which $v_1$ has neighbors $v_2, v_3, \ldots, v_{d_1+1}$.

(c) Using the result from part (b), describe an algorithm that on input $d_1, \ldots, d_n$ (not necessarily sorted) decides whether there exists a graph with this degree sequence. Your algorithm should run in time polynomial in $n$.

5.28. Alice wants to throw a party and is deciding whom to call. She has $n$ people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know *and* five other people whom they don't know.

Give an efficient algorithm that takes as input the list of $n$ people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of $n$.

5.29. A *prefix-free encoding* of a finite alphabet $\Gamma$ assigns each symbol in $\Gamma$ a binary codeword, such that no codeword is a prefix of another codeword. A prefix-free

encoding is *minimal* if it is not possible to arrive at another prefix-free encoding (of the same symbols) by contracting some of the keywords. For instance, the encoding $\{0, 101\}$ is not minimal since the codeword 101 can be contracted to 1 while still maintaining the prefix-free property.

Show that a minimal prefix-free encoding can be represented by a full binary tree in which each leaf corresponds to a unique element of $\Gamma$, whose codeword is generated by the path from the root to that leaf (interpreting a left branch as 0 and a right branch as 1).

5.30. *Ternary Huffman.* Trimedia Disks Inc. has developed "ternary" hard disks. Each cell on a disk can now store values 0, 1, or 2 (instead of just 0 or 1). To take advantage of this new technology, provide a modified Huffman algorithm for compressing sequences of characters from an alphabet of size $n$, where the characters occur with known frequencies $f_1, f_2, \ldots, f_n$. Your algorithm should encode each character with a variable-length codeword over the values $0, 1, 2$ such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Prove that your algorithm is correct.

5.31. The basic intuition behind Huffman's algorithm, that frequent blocks should have short encodings and infrequent blocks should have long encodings, is also at work in English, where typical words like `I`, `you`, `is`, `and`, `to`, `from`, and so on are short, and rarely used words like `velociraptor` are longer.

However, words like `fire!`, `help!`, and `run!` are short not because they are frequent, but perhaps because time is precious in situations where they are used.

To make things theoretical, suppose we have a file composed of $m$ different words, with frequencies $f_1, \ldots, f_m$. Suppose also that for the $i$th word, the cost per bit of encoding is $c_i$. Thus, if we find a prefix-free code where the $i$th word has a codeword of length $l_i$, then the total cost of the encoding will be $\sum_i f_i \cdot c_i \cdot l_i$.

Show how to find the prefix-free encoding of minimal total cost.

5.32. A server has $n$ customers waiting to be served. The service time required by each customer is known in advance: it is $t_i$ minutes for customer $i$. So if, for example, the customers are served in order of increasing $i$, then the $i$th customer has to wait $\sum_{j=1}^{i} t_j$ minutes.

We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i).$$

Give an efficient algorithm for computing the optimal order in which to process the customers.

5.33. Show how to implement the stingy algorithm for Horn formula satisfiability (Section 5.3) in time that is linear in the length of the formula (the number of occurrences of literals in it).

5.34. Show that for any integer $n$ that is a power of 2, there is an instance of the set cover problem (Section 5.4) with the following properties:

i. There are $n$ elements in the base set.

ii. The optimal cover uses just two sets.

iii. The greedy algorithm picks at least $\log n$ sets.

Thus the approximation ratio we derived in the chapter is tight.

5.35. Show that an unweighted graph with $n$ nodes has at most $n(n-1)$ distinct minimum cuts.