Surveying the field of computing

third edition

by Carl Burch

Contents

List of exercises

Syllabus

Course overview

Our goal is *to understand the **scope**, **techniques**, and **contributions** of the field of computer science.*

Topics

This is a survey course; we will move fairly rapidly through a series of topics. (Table 0.1 gives an anticipated schedule.) The course is structured around four major areas, of which you should have a fair understanding by the end of PGSS.

Programming fundamentals. You will be programming-literate by the end of the course.

Recursion. Between induction in Discrete Math and recursion in CS, you will have mastered the use of constructive self-reference.

Internet. You will have learned the fundamentals of the Internet's structure and operation.

Algorithms. You should appreciate the mathematical side to the study of algorithms, understanding especially the mathematical analysis of running time.

The course should be challenging, enlightening, and fun for everybody; please demand it!

For the inexperienced...

The CS core is particularly frustrating because a wide experience gap separates the students. Many students have never attempted to understand computing before; a few have several computer science courses behind them. No matter what, the beginners will be frustrated, and the experienced students will be bored. Expect frustration.

But don't be daunted: You are in the majority! The material is all targeted at you.

We will move very quickly. I encourage beginners especially to ask questions when they don't feel they completely understand the topic. It's tempting to think that maybe if you sit back you will begin to understand later... but this doesn't work. Ask!

The assignments will have multiple levels and will indicate what the beginner should do. As an overachieving student, you may find it hard not to go to the highest level; nonetheless, I recommend that beginners avoid this.

Experience shows that some students grasp programming quickly while others do not. This is naturally frustrating for those among the latter. If programming doesn't seem to be clicking for you, I advise you to stick with it nonetheless — you will be a better person for the struggle, and the effort will pay off when you try again in a few years.

<table>
<tr><th>Session</th><th>Material</th></tr>
<tr><td>1</td><td rowspan="2">Prologue</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="5">Programming</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td><td rowspan="2">Recursion</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td><td rowspan="2">Internet</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td><td rowspan="3">Algorithms</td></tr>
<tr><td>13</td></tr>
<tr><td>14</td></tr>
<tr><td>15</td><td>Cryptography</td></tr>
<tr><td>16</td><td>TBA / Quiz</td></tr>
<tr><td>17</td><td>Evaluation / Epilogue</td></tr>
</table>

Table 0.1: Tentative schedule of topic coverage

For the experienced...

Everybody should find the CS core interesting and challenging! Unfortunately it is not possible within PGSS to have multiple sections of the CS core. But I still believe students who have already studied computer science intensively can still learn a lot — at least if they allow themselves.

This class employs three strategies for teaching experienced students without neglecting those who have not seen much computer science before.

1. The assignments have multiple levels to which they may be completed. While the lower levels are acceptable — nay, encouraged — for beginners, experienced programmers should find the more advanced levels more interesting.

2. We will get programming behind us as quickly as possible, so that we can discuss material experienced students are less likely to have seen.

3. During these programming lectures (Sessions 2–8 in Table 0.1), students who already know how to program have the option of pursuing self-paced study. You can read more about this at the end of this syllabus.

Learning resources

The CS core includes many facets intended to make this course challenging, enlightening, and fun for everybody. Please take full advantage of them!

CS staff

Without question, the greatest resource at your disposal is the CS staff. We are employed full-time to help you learn about CS. Never hesitate to ask for help!

Carl Burch	`burch@andrew.cmu.edu`	CS core instructor
Kirk Yenerall	`ky0h@andrew.cmu.edu`	lab/CTW instructor
Matt Huenerfauth	`matt@udel.edu`	TA
Rob Liebscher	`liebsche@cse.psu.edu`	TA
Merrie Ringel	`Meredith_Ringel@brown.edu`	TA

Programming classwork

Each weekly assignment includes a programming portion. Additionally, students in self-paced study will have more programming assignments to complete. You will find these assignments are the most effective learning resource for this class. Take them very seriously!

Collaboration: *Strongly* encouraged! On the weekly programming assignments, you may work in groups of up to three. *Collaboration* together is essential; you may *not* merely cooperate by splitting the assignment between group members. You *may* get some help from course staff and classmates, but the work must be primarily yours. List all collaborators and helpers on what you hand in.

Help from staff: Encouraged! Feel free to buttonhole any of the course staff, any time, to talk about assignments or class material. To make this easier for you, we frequently visit the Baker Hall clusters to see if students can use our help. Sometimes this will be announced (in class or on the Web page); other times we may just drop in.

Suggestion: Start early! Programming is notorious for almost always taking longer than expected. Also, starting early will avoid problems with computer and TA availability.

Submissions: Your group's programming assignments must be submitted together electronically via the 'Handin' Web page. The electronic deadline is 2:30 pm Monday.

Solutions and evaluations: Soon after the deadline, we will post our solutions on the 'Solutions' Web page. The course staff will evaluate your work; their evaluations will appear on the 'Grades' Web page.

Cheating: *Not* encouraged! By not giving grades, PGSS removes the only incentive to cheat. Any detected cheating cases will go straight to Dr Berget. Expulsion in such cases is likely. It's not worth it!

Written classwork

This class includes a weekly written assignment. Again, these are a vital learning resource for the class that you should take very seriously!

Collaboration: Encouraged! Feel free to work with your classmates (or with course staff) about the written assignments as you search for a solution. You must write up your solution on your own, however, and you should not show your write-up to other students. Be sure to list all collaborators and helpers (including course staff) on what you hand in.

Help from staff: Encouraged! Again, feel free to buttonhole any of the course staff, any time, to talk about the written work. We will also be happy to look over your write-up and indicate any significant shortcomings. (Do *not* have classmates look over them, however.)

Submissions: Your written assignment may be submitted electronically (via the 'Handin' Web page) or on paper (via the dorm mail room's CS box). The deadline is 2:30 pm Monday.

Quiz

At the end of PGSS, the course staff must evaluate your performance for the PGSS recommendation letter. A drawback of emphasizing groupwork so much on assignments is that there is no reliable individual evaluation mechanism. The final quiz at the end is meant to address this.

There will be a short, 15-minute quiz at the penultimate session of class. Let me tell you three reasons why you should *not* worry about this quiz.

- You may use both the textbook and any papers. (You cannot use computers, calculators, people, and especially tangerines.)
- The quiz will primarily emphasize what has appeared on the assignments. You should understand them already (after all, you completed them), but anyway it is a small amount to study. The 1998 quiz appears in this book on page x.
- For the sake of argument, let's say the material has simply not clicked. Then no doubt the course staff has helped you many times. If, then, you perform poorly on the quiz, we will remember your struggle, and our evaluation will emphasize this, not quiz performance. (Of course, it's even better if you will do well enough that we can emphasize both struggle and success.)

Web page

The course Web page is a major component of course administration. Its address is (you might as well memorize it now): `http://www.cburch.com/`. This syllabus mentions much of Web page elsewhere, but there are a few other useful parts.

Registration: Maintain your Web page account and password. This account is necessary for submitting assignments, reading evaluations, and completing polls. You will receive a registration ticket on the first day of class to allow you to create your account.

Polls: Answer questions about how you would get the most from the CS core experience.

Comment box: Submit anonymous comments to me about what would contribute to the CS core experience. Comments about any aspect of the course are welcome at any time, whether anonymous or not.

Web links: Links to other sites on the Web relevant to the material we cover in class.

Lectures

Questions: Asking questions during lecture is *strongly recommended at all times*! It is very easy to complacently sit back and become very lost, thinking you can get back on track later; but in fact getting back on track is often impossible. Don't let this happen! Ask questions when you begin to feel even vaguely that you are falling behind.

Notes: I personally believe that note-taking (as commonly performed) often detracts from student understanding. If you find your own notes useful as a learning aid, go ahead. In case you would like to have a copy of notes during class for reference or annotation, anticipated notes will be available on the 'Lecture' Web page by the evening before lecture in time for you to print them out.

Textbook

We will cover the textbook *Surveying the field of computing* (third edition) fairly closely, at a rate of roughly one chapter per lecture.

Self-paced track

The self-study option is for students with a fair amount of computer programming background already (in any sophisticated programming language).

If you are eligible and select this option, instead of sitting in lecture studying the fundamentals of programming, you will be in the computer clusters (Baker 140) practicing. The CS TAs will be there to answer questions and to monitor your work. (Mr Carl Burch, who will be lecturing, regrets that he will be unable to attend.)

PGSS approves this option with certain restrictions:

1. The self-study option is available *only* to those who have significant programming background. More specifically, the student is expected to be somewhat comfortable at least with loops and arrays in some programming language.

2. You must be either at lecture or in Baker 140 during the scheduled class time. If you are in Baker 140, you must study CS. (You cannot defer self-study to other times.) The TAs will monitor and take attendance.

3. During this time, you will complete and submit exercises listed on the Web page. (Except for Programming Exercise 0, you may *not* work on the weekly assignments during class time.) Your submissions will be evaluated by the course staff and will count toward your final evaluation.

4. The TAs are there to answer any questions you have. In addition, we will often be in the cluster at other times to answer questions. And of course you should always feel free to stop any course staff, any time, to get help.

The self-study programming assignments are divided into exercises and projects. The exercises are to ensure that you learn what is taught in lecture; you must do these first. The projects give you the opportunity to learn material that we will not cover elsewhere.

1998 quiz

Quiz questions

You may use your book and your notes; you may not use classmates.

1. Write a program to determine whether a number the user enters is a multiple of 7. A sample run:

```
Test what number? 1245
Not multiple of 7
```

```
#include <iostream>
#include <string>

int main() {







}
```

2. Write a function to count the number of occurrences of the number 9 in an array. For example, if our parameter array arr is $\langle 5, 9, 3, 2, 9, 0 \rangle$, the function should return 2, since 9 occurs twice.

```
int countNines(const vector<int> &arr, int arr_len) {
    int i;


    for(i = 0; i < arr_len; i = i + 1) {




    }


}
```

3. Using big-O notation in terms of n, what is the running time of the following function? Explain your answer in one or two sentences.

```
public static int squareRootD(int n) {
    int i = n;
    while(i * i >= n) {
        i = i - 1;
    }
    return i + 1;
}
```

Solutions

1.
```
#include <iostream>
#include <string>

int main() {
    cout << "Test what number? ";
    int num;
    cin >> num;          // read number from user
    if(num % 7 == 0) { // test whether it is divisible by 7
        cout << "Multiple of 7" << endl;
    } else {
        cout << "Not multiple of 7" << endl;
    }
}
```

2.
```
int countNines(const vector<int> &arr, int arr_len) {
    int i;
    int count = 0; // this is the #9's we have seen so far
    for(i = 0; i < arr_len; i = i + 1) {
        if(arr[i] == 9) {
            count = count + 1; // we found another one
        }
    }
    return count;
}
```

3. $O(n)$. The program begins i at n and subtracts 1 from it until it reaches $\sqrt{n}$. So the computer goes through the loop approximately $n - \sqrt{n}$ times; the $\sqrt{n}$ term grows more slowly than the n term, so we go through the loop $O(n)$ times. Each iteration takes O(1) time (Constant Rule), so by the Iteration Rule, the total time is $O(n)$.

Acknowledgments

The Pennsylvania Governor's School for the Sciences provided the occasion to put this book together. I most especially want to recognize the 1997 PGSS class, who faithfully stuck with me on the first iteration of this course, and the 1998 PGSS class, whose comments led to this current edition. The suggestions and struggles of these students have shaped this course and this book. I wish the students could benefit from what they taught me.

The author acknowledges Carnegie Mellon University and Charles and Cheri Burch for providing the computing facilities on which this document was produced.

First Unit

Foundations

This begins the first unit of this text, *Foundations*. In this part, we learn about some of the most fundamental aspects of computer science. We begin in Chapter 1 with an overview of computer science and of this book. In Chapter 2 we look at the types of problems computer science investigates and the solutions that it seeks. Finally, in Chapter 3, we examine at two common ways for expressing procedure concisely, pseudocode and flowcharts.

Chapter 1

Overview

In this book we undertake to understand the discipline of computer science. Surveying this vast field is a high goal, but it is nonetheless ours to pursue.

We are particularly interested in learning about the scope, techniques, and contributions of computer science. Our approach is to look at a variety of different parts, so that you can draw an educated picture about what computer science is about. To use the cliché: We will see the trees; you will draw the forest.

But before we attempt to examine these trees, we glance at the author's vision of the forest. In this chapter we look at this outlook, and we see an outline describing how this book approaches its goal of surveying the field of computing.

1.1 What is computer science?

"What is computer science?" is exactly the question this book attempts to answer. But let's take a crack at the question directly: **Computer science** is the field of inquiry that asks, "How can we solve problems most effectively?"

Understanding how to solve problems effectively is vital to advancing technology, even without computers. That computer science was not a separate field until computers became usable in the 1950's, therefore, is surprising.

You may object that *computer* science should be at least somewhat related to computers. In fact, it is. Because computers can solve many problems automatically, they are important problem-solving tools. But as computer scientists we are aware that computers are not fit for all problems; our goal is to try to extend the domain of problems that can be approached automatically (by computer).

As you might expect of a brilliant gem such as computer science, it has many facets. Each is enough for a lifetime of study. In this book we will hit many of them, but for now let us spend just a few seconds on each of a selection of some of the more prominent facets.

Computational complexity asks, "What is the inherent hardness of problems?" This leads to deep mathematical questions. Computational complexity researchers look for lower bounds on how quickly particular problems can be solved.

Algorithms researchers ask, "What are provably fast algorithms for problems?" This question is the inverse of computational complexity's — instead of asking what *can't* be done for a problem, it asks what *can* be done.

Artificial intelligence (abbreviated **AI**) asks, "How can we automatically behave 'intelligently'?" AI researchers attempt to find methods of emulating complex aspects of human behavior. Learning, understanding, and planning are all difficult aspects that researchers are examining.

Human–computer interaction (abbreviated **HCI**) asks, "How can humans efficiently specify problems?" In many cases the biggest time sink in problem-solving is specifying the problem itself. HCI researchers seek ways to improve computers' interfaces so that humans can use them more efficiently. They want to make computer use faster, more pleasant, and more intuitive.

Programming languages asks, "How can we best describe approaches to problems?" Researchers seek better ways of specifying a method exactly, with mathematical rigor, in a way that both humans and computers can understand the method quickly and easily.

Software systems asks, "How can computers better help solve problems?" Software systems researchers look for new ways to use computers' capabilities. Some examples of current software systems research include development of different programs to archive video, to share files across a network, and to safely transfer money over the Internet.

Software engineering asks, "How can we develop complex systems better?" We want faster, safer, simpler, more powerful software without spending too much more. Software engineering researchers look for ways to alter the development process so that better software is produced efficiently.

Hardware systems asks, "How can we develop better problem-solving machines?" Among other things, researchers look for ways to build faster networks, to improve computer reliability, and to use many processors in the same computer effectively.

These facets are widely different, touching a variety of the academic disciplines. Computer scientists often collaborate with people from other university departments. Some of the more common sources for collaboration are mathematics (computational complexity and algorithms), the natural sciences (algorithms and AI), philosophy (AI and programming languages), psychology (AI and HCI), sociology (HCI and software engineering), management (software engineering), and electrical engineering (hardware systems). And of course computer scientists work with each other, often across different parts of the field.

Exercise 1.1: We define computer science as the study of solving problems effectively. Do you think this a reasonable definition? Is it general enough (covering all the facets)? Is it specific enough (not hitting other fields)? Can you recommend an alternative definition?

1.2 Outline

All these questions that computer scientists ask can be overwhelming. The point is that computer scientists ask a lot of neat questions. We can't possibly cover all of the field in a few hours, in a few years, or even a few lifetimes. We can only sample a selection.

This text is structured around five *units*, each roughly corresponding to a major concept of the course. Each unit contains two to six chapters; a chapter is meant to be a lesson conveying a single topic, of a length reasonable for a single reading or a single lecture.

First Unit: *Foundations* We lay a foundation for learning about computer science, with a brief introduction to the field and the problems it tries to solve.

Second Unit: *Programming* We examine how to write a program in a particular programming language called C++. Our overall goal is this book is not to learn how to program. But knowing how to program is crucial to a strong understanding of computer science (and it is a useful skill anyway).

Third Unit: *Recursion* Once we understand programming fundamentals, we will be ready to talk about more involved concepts. *Recursion*, the third unit, addresses a very useful programming device called recursion, which requires a more abstract understanding of programming. We will examine how it works and how we can use it for game-playing. The game-playing chapter represents our first foray into "real" computer science.

Fourth Unit: *Internet* We concentrate on a major recent contribution of computer science to the world at large: new approaches to information networks. In particular, we look at many of the inherent problems, and we see details of how these are solved in the Internet.

Fifth Unit: *Algorithms* Finally, we look at mathematical aspects of algorithm design. We particularly concentrate on how to construct fast solutions for a problem.

After all this, we should have some idea of the breadth of computer science. We're omitting entire fascinating subdisciplines in this survey. We simply cannot cover everything. The above outline carves out a huge chunk that is more than enough. What we have selected makes an exciting and interesting journey.

1.3 What to expect

The first thing you should draw from this book is a much deeper understanding of computer science. This is the book's purpose. But there are other benefits you will also accrue along the way: writing procedure, manipulating procedure, and handling complexity.

Through studying this material you should become better at writing procedure. This is most obvious as we study *Programming* and *Recursion*. Good programming involves communicating a procedure well — so well, in fact, that even a computer can understand it. Programming is a good way to learn how to think precisely about procedure.

Related to this, but slightly different, is manipulating procedure. Computer science treats programs not only as solutions, but things to be considered and analyzed. We will be reasoning closely about procedure in *Algorithms*.*

Finally, through computer science you can learn to handle complexity better. Much of computers and their software are terrifically complex instruments, and much of system design involves handling complexity. The principal tool for attacking complexity is *abstraction*. This is one of the big lessons of *Internet*, one of the most complex human systems developed.

Thus this course not only teaches specific material, it also teaches new, useful ways of thinking. We begin our journey with the first step, as we examine more closely what we mean by *problem* when we define computer science as the *study of solving problems effectively*.

*A word about this reasoning: It often gets rather mathematical. If you ever get mired in proofs, don't become too discouraged. Most of the material is disjoint, so if you skip over a proof because it is giving you real difficulty, you should be able to rejoin after it. The mathematics is meant to be accessible to those with a solid foundation in high school algebra, but understanding mathematics well takes practice. Even very talented, mathematically-oriented computer scientists are often confused.

Chapter 2

Problems and algorithms

If computer science is about solving problems effectively, we should be specific about our terms. In this chapter we look more carefully at what we mean by *problem* and *solution*.

2.1 Problems

A **problem instance** is a question whose exact answer is well-defined. "What is $23 \cdot 42$?", "Is 4097 prime?", and "What's the shortest way to Tulsa?" are problems. "What is the meaning of life, the universe, and everything?" is not; there is no accepted way to prove an answer wrong.

Usually we want techniques to solve entire sets of problem instances. A computer that only calculates $23 \cdot 42$ is not very useful; but it is useful if it can calculate $x \cdot y$ for any x and y. A set of related problem instances we call a **problem**.

A problem is specified by its **inputs**. In the case of a computer that calculates $x \cdot y$, the inputs would be the specific values of x and y. In response to the inputs, the computer should produce an answer, called its **output**. We describe a problem by specifying the input and desired output.

Problem Multiplication:
Input: two numbers x and y.
Output: the product of x and y.

Another problem is Primality. (Recall that n is **prime** if only 1 and n divide n exactly.)

Problem Primality:
Input: an integer n that is greater than 1.
Output: `true` if n is prime, and `false` if not.

You might think from these two examples that problems in computer science are typically computational in nature. And it is true that in this book we will investigate Primality (next section) and Multiplication (Chapter 18). But the problems tackled by computer science span a wide spectrum. Two more examples will suffice to demonstrate this.

One problem that interests computer scientists comes up in chess. As input, the computer receives a current chess board. The desired output is a move that guarantees a win no matter what the opponent does. Of course this is a very difficult problem for even moderately complicated boards. We will visit game-playing problems like this in Chapter 11.

Another problem is the Fire-Hydrant problem (more commonly known as Dominating-Set). In this problem we are given a map of houses as in Figure 2.1, with lines connecting

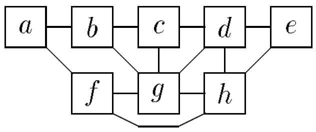


Figure 2.1: An example Fire-Hydrant problem.

adjacent houses. We want to build as few fire hydrants as possible so that every house either has a hydrant or is next to one that does. The output should be the set of houses at which to build hydrants. In the example of Figure 2.1, the output might say to build hydrants at houses a and d. (One hydrant is not enough for this map.)

These two problems are closer than Multiplication and Primality to the problems computer scientists typically investigate. Neither is inherently very computational, but they are very interesting from a computer science perspective.

Exercise 2.1: (Solution, 116) The Search problem is to determine whether a number occurs in a list of numbers. Give a formal description of the inputs and outputs for this problem, similar to those we gave for Primality and Multiplication.

2.2 Solutions

As computer scientists we want techniques to solve problems automatically. Such an automatic solution is called an **algorithm** if it eventually finds the correct output for any valid input. That is, algorithms are recipes with two important properties: They always stop for any input (**termination**), and they always output the correct answer (**correctness**).

You already know many algorithms. For Multiplication, for example, you were taught an algorithm in grade school. (But in Chapter 18 we'll see an algorithm faster than the one you probably know.)

To illustrate algorithms, let's consider an algorithm for Primality, which we call Prime-Test-Exhaustive. In this algorithm, when we want to see if n is prime, we try all the numbers between 2 and $n - 1$ to see if any of them divide the input exactly. If so, then we know the number is not prime; otherwise, it is.

For example, say we want to know if 7 is prime. We first try dividing 7 by 2. After some labor, we find that there is a remainder of 1, so 2 is not a divisor of 7. Then we try 3. Again, there's a remainder of 1. Now we try 4, which we find is not a divisor of 7 either. We continue to 5, then 6. We now know that none of the possibilities are divisors, and so we output `false` — 7 is not prime.

Actually, we can make a much faster algorithm using a simple observation: If n is not prime, then it must have a divisor of n that is at most $\sqrt{n}$. This is because if $p \cdot q = n$, then either p or q is at most $\sqrt{n}$ — they can't both be more than $\sqrt{n}$, because then their product would be more than n. So to determine if a number is prime, we can stop once we pass $\sqrt{n}$. We call this Primality algorithm Prime-Test-All.

Now if we're using Prime-Test-All for 7, we would test 2 and stop. (Since $3 > \sqrt{7}$, there's no point in going farther.) This algorithm is much faster than before; if n were $1,000,003$, then Prime-Test-Exhaustive would try $1,000,001$ possible divisors. Now we try only 999.

This simple example illustrates a theme that we develop much more extensively in this book's fifth unit, *Algorithms*: Often, with a little cleverness, we can find a faster algorithm.

Exercise 2.2: (Solution, 116) In Prime-Test-All, we want to find the square root of a number n. This suggests the following problem.

Problem Square-Root:
Input: an integer n.
Output: the largest integer k so that $k^2 \leq n$.

Think of two significantly different algorithms for this problem. Describe each of them, and in 3–4 sentences compare what you think are their relative advantages and disadvantages.

(You may have seen complicated algorithms similar to long division for this type of problem. Feel free to think of simpler, more obvious ways to solve this problem.)

2.3 Undecidability (optional)

A natural question to ask is, is there an algorithm (perhaps very slow) for every problem? In this section we will see that the answer is no. Problems that have no algorithm are **undecidable**.

Consider the following problem, which we will show does not have an algorithm.

Problem Halting:
Input: a program P and input to the program x.
Output: `true` if P ever stops given input x, `false` otherwise.

This is not obviously unreasonable. For many programs, we can reason whether or not they will stop. But, we will show, this is not true for all programs.

Say that we have an algorithm Does-Halt for the Halting problem. By definition, Does-Halt has the twin properties of correctness and termination. We will show that the existence of such an algorithm leads to a contradiction.

In particular, consider the following program (which we call Does-Not-Halt), which takes a program P as input: First, Does-Not-Halt runs Does-Halt to see if P will halt when given P as input. If Does-Halt returns `true`, then Does-Not-Halt enters a segment of code that simply repeats itself without ever stopping. If Does-Halt returns `false`, then Does-Not-Halt immediately stops.

Now we ask the following, simple question: What will Does-Not-Halt do if the input P is Does-Not-Halt itself? There are two possible behaviors: It either eventually stops, or it does not.

If Does-Not-Halt never stops given the input Does-Not-Halt, then (because Does-Halt, as an algorithm, always terminates correctly) Does-Halt must output `false`. So, if we see how Does-Not-Halt behaves when given itself as input, it enters the second case and so immediately stops. This is a contradiction, so Does-Not-Halt cannot loop forever given itself.

If Does-Not-Halt does stop, then Does-Halt must output `true`. But this means that Does-Not-Halt never stops when given the input of Does-Not-Halt. This also is a contradiction.

So there is no correct answer for what Does-Not-Halt does when given the input of Does-Not-Halt. Our contradiction is that neither of its two possible behaviors is possible! Our assumption of the existence of Does-Halt must have been wrong.

This proof may sound like a trick; it's not. It may also sound very abstract — but again, it's not; the proof has practical implications. A program which says whether a program will eventually stop would be useful, since it could ensure that a computer never becomes stuck in a useless loop. This proof says that no program can possibly achieve this correctly.

Chapter 3

High-level procedure

Writing programs is largely a process of describing algorithms in an exact, formal way. An important step toward this is to look at how to describe algorithms less formally. This helps us conceptualize algorithms, a crucial element of programming.

One way to describe an algorithm is with simple English text (or any other human language), but this has two major disadvantages. English text buckles under the load of complicated algorithms. And English is also often woefully ambiguous. Therefore, in this chapter we look at two more systematic communication media for algorithms: *pseudocode* and *flowcharts*.

3.1 Pseudocode

Pseudocode is a formatted mixture of English and mathematics intended to communicate an algorithm's structure concisely and exactly. It is *not* a definite, single way of writing an algorithm.

Pseudocode has been around a long time, and you have certainly seen it before. Open any book of recipes for a multitude of examples (or see Figure 3.1). Recipes tend to make poor pseudocode, though, since they are often ambiguous. Exactly how thick a layer of butter should be placed on that foil? When do I stop beating these eggs?

Prime-Test-Exhaustive

The simplest way to learn about pseudocode is to look at an example. Remember Prime-Test-Exhaustive from Chapter 2? That Primality algorithm takes a number n and checks each number between 2 and $n - 1$ to see if that number is a divisor of n. One way to write this in pseudocode is as follows.

Algorithm Prime-Test-Exhaustive(n)
1. For each number i between 2 and $n - 1$, do the following:
    a. Divide n by i.
    b. If the remainder is 0, then output `false` and stop.
2. If none of these numbers divide n, then output `true` and stop.

This book tends to use a slightly more systematic method that more closely resembles computer programs.

Algorithm Gingerbread($foil$, $butter$, $brown\_sugar$, $white\_sugar$, $molasses$, $eggs$, $flour$, $baking\_soda$, $ginger$, $cinnamon$, $allspice$)
1. In a large bowl, cream 1 c $butter$, $1\frac{3}{4}$ c $brown\_sugar$, and $1\frac{1}{4}$ $white\_sugar$.
2. Put 2 tbsp $molasses$ and 6 $eggs$ into bowl and beat.
3. Into another large bowl, sift 6 c $flour$, 2 tsp $baking\_soda$, 1 tbsp $ginger$, 1 tbsp $cinnamon$, and 1 tbsp $allspice$.
4. Combine contents of both bowls and kneed into a small ball of $dough$.
5. Cover $dough$ and place into refrigerator.
6. Wait at least 30 minutes.
7. Repeat the following until $dough$ is $\frac{1}{4}$ inches thick:
   a. Flour surface.
   b. Roll $dough$.
8. Cut $dough$ into pieces.
9. Line cookie sheets with $foil$.
10. Apply $butter$ and $flour$ to $foil$.
11. Using a spatula, gently lift $dough$ and place it on $foil$.
12. Preheat the oven to $325°$F.
13. Bake 15–20 minutes or until slightly firm.
14. Let cool on racks until firm enough to handle.

Figure 3.1: An algorithm for the Hungry problem.

Algorithm Prime-Test-Exhaustive(n)
for each i **from** 2 **to** $n - 1$, inclusive, **do:**
    if i divides n, **then:**
        output `false`
        stop
    end of if
end of loop
output `true`

You can write pseudocode any way you want, as long as it is structured (note the use of indentation) and precise.

Notice that our pseudocode sometimes skips over steps. For example, in our pseudocode for Prime-Test-Exhaustive, we did not say exactly how to divide two numbers. This is because division is built into computers, so that computers already "know" how to divide; thus we didn't need to explain this. As you learn to program, you will get a better feel for exactly what is built into computers and what you need to explain. (For now, you can draw the line at what you would explain to another person if you were trying to describe the crucial steps of the algorithm.)

Mode-Tally

Let's look at another example, an algorithm for finding the mode of a list of scores. The **mode** of a list of numbers is the number that occurs most frequently. For example, if a group of students take a test, and their scores are $\langle 3, 8, 7, 8, 2, 9, 8, 9\rangle$, the mode score is 8, since 8 occurs thrice and all other scores occur less often.

Problem Find-Mode:
Input: A list L of scores between 0 and 100.
Output: The score that occurs most frequently in L. (In the case of a tie, any of the most-frequently-occurring is acceptable.)

There are many interesting ways to compute a mode. Here we consider just one of them, called Mode-Tally. In this technique, we create 101 empty tally boxes, labeled with the numbers 0 through 100. Then we go through the list; for each score x in the list, we find the tally box with x as its label and add a tally mark to that box. Finally, we go through the tally boxes to see which has the most marks, and return the label of this box.

How can this be expressed in pseudocode? One way is the following, almost identical to the above English description.

Algorithm Mode-Tally(L)
Create 101 empty tally boxes, labeled 0 through 100.
for each score x in L, **do:**
    Find the box labeled x.
    Add a mark to it.
end of loop
Let $mode$ be 0.
Let $modeCount$ be the number of marks in the box labeled 0.
for each number x **from** 1 **to** 100, **do:**
    Let $xCount$ be the number of marks in the box labeled x.
    if $xCount > modeCount$, **then:**
        Let $mode$ be x instead.
        Let $modeCount$ be $xCount$ instead.
    end of if
end of do
output $mode$.

This example happens to be very explicit about how to find the maximum in the list, but this is not really necessary, since this is so obvious to humans.

Mode-Tally is actually a fairly complex procedure. If you understand it, and if you can write pseudocode for similarly complex procedures, then you are well on your way to learning to program. The only thing left is learning how to translate pseudocode into a completely formal language that a computer can understand. In any case, you will improve as you study the details of programming later in this book.

Exercise 3.1: (Solution, 117) Write pseudocode describing each Square-Root algorithm you made in Exercise 2.2.

3.2 Flowcharts

Flowcharts are another general way of describing procedure. Flowcharts use a graphical language to emphasize how procedure flows through the algorithm.

Again, the best way to learn about flowcharts is to look at an example. Figure 3.2 gives a flowchart for our Prime-Test-All algorithm. Recall that Prime-Test-All is like Prime-Test-Exhaustive, except that it tests only numbers up to $\sqrt{n}$.

In this flowchart, you will notice three shapes. Ovals are reserved for the "START" and "STOP" indicators, which indicate where to begin the algorithm and where to stop. Boxes are

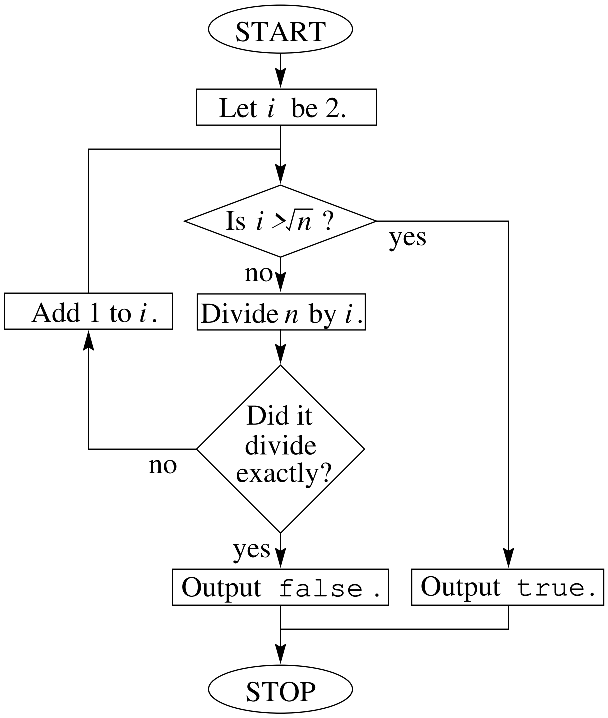


Figure 3.2: A flowchart for Prime-Test-All.

meant for simple instructions, like "Add 1 to i." A box always has exactly one outgoing arrow. Finally, diamonds are meant for questions, like "Is $i > \sqrt{n}$?". Diamonds have two outgoing arrows, one for each possible answer (*yes* and *no*).

We'll see some more flowcharts as we look at programming. In practice, programmers use flowcharts primarily for very-high-level diagrams relating components of a large software system, not for low-level procedures like Prime-Test-All. For such procedures, pseudocode is usually more convenient. Nonetheless, flowcharts are very useful to beginners as a way to conceptualize procedure.

Exercise 3.2: (Solution, 118) Draw a flowchart for one of the Square-Root algorithms you made in Exercise 2.2.

SECOND UNIT

Programming

We turn to learning the fundamentals of programming. This isn't the place to try to learn every single detail about a particular programming language. If we tried this, we would run out of space and time to discuss other interesting bits of computer science. Instead we cover the basic building blocks of programs. Understanding this much will enable us to write useful programs and prepare us to understand computer science more completely.

The specific programming language we study is C++. There are many prominent programming languages today, most very similar in approach. We could choose any of them, but C++ has the advantage of being widely used both in education and in industry.

C++ is a notoriously huge language, and we cannot hope to cover every detail in only a few short chapters. We therefore study a subset of the language, illustrating the conceptual details of programming and enabling you to write genuinely useful C++ programs. In order to maintain compatibility with other introductory courses, this subset is largely a restriction of the College Board's C++ subset used in their AP Computer Science exams. (But this is not an AP course: It includes only about half of the AP subset.)

Our introduction to C++ consists of six chapters. In each chapter, we'll introduce a new piece to the programming puzzle by describing the concept, illustrating it, and examining an extended example program putting everything together.

Chapter 4 discusses the overall programming process and looks at a brief example program.

Chapter 5 examines how a program can manipulate data using *variables*.

Chapter 6 examines how a program can control the execution of instructions using *control statements*.

Chapter 7 examines how larger tasks can be broken into more manageable subtasks using *functions*.

Chapter 8 examines how to use conglomerations of data called *arrays* and *structures*.

Chapter 9 introduces the concepts of *objects* and *object-oriented design*.

By the end of these chapters, you should be able to write a wide variety of useful programs, and you should have some idea of what programming is about.

Three appendices to this book supplement this material. Appendix A is a quick-reference outlining C++ syntax. Appendix B lists common names for keyboard symbols. And Appendix D provides solutions for many of the exercises.

As you reach exercises, you should try them out. When it asks that you write a program, preferably you will write it on a computer and test it yourself. But if that isn't immediately available, at least do it on paper. Then read the solutions in Appendix D: Often, the solutions introduce new material; in all cases, you're likely to learn from seeing alternative approaches to the questions.

Chapter 4

Programming overview

In this chapter we look at programming at the broad, high-level view, starting with the programming process, continuing with a simple C++ program, and ending with some programming guidance. The details of programming we leave to following chapters.

4.1 The programming process

The pipeline from idea to action consists of three phases: *programming*, *compiling*, and *executing*. Figure 4.1 illustrates the process.

In **programming**, you as the programmer translate your mental concept of how the computer should behave to corresponding **code** written in a **programming language** such as C++, Java, or Ada. A programming language is a compromise between what humans find most natural for expressing procedure and what computers can easily interpret in an efficient way. Important tools in the programming phase include pseudocode and flowcharts, as well as a good editor for writing and editing code.

Computers, as built, cannot actually understand programming languages; they are built to understand a much more primitive language called **machine language**. (Different types of machines have different machine languages — the machine language for a PC is totally different from the machine language for a Macintosh.) In **compiling**, the computer runs a program to translate the programmer-written code to machine language. This program is called a **compiler**, and it is the primary programming tool for the compile phase.

In the final phase, the computer **executes** the machine language that the compiler produced. At this point the machine finally does the job that the programmer originally conceived… at least if all the phases proceed flawlessly.

During these phases, errors crop up due to programmer errors. For all these errors, the solution is for the programmer to discover where the code is wrong, to fix the code, and to repeat the compile and execution phases.

Executing: A **run-time error** occurs during execution; one example of a common run-time error is when the machine code instructs the computer to divide a number by zero. Often such errors *crash* the program; that is, execution stops abruptly.

Compiling: A **compile-time error** is an error that prevents the compiler from interpreting the program. If the code contains a mistyped name, for example, then this causes a compile-time error. The result of a compile-time error is that the compiler refuses to compile the

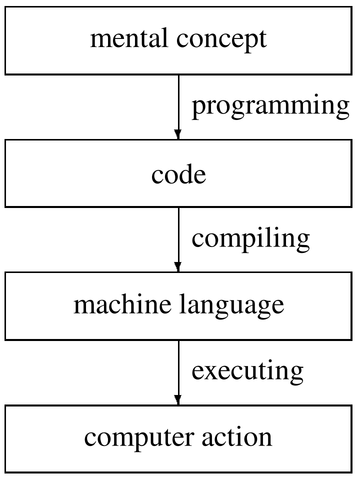


Figure 4.1: The programming process.

program, instead issuing a description of what is wrong with the program. This is the easiest type of error to fix, since the compiler usually points directly to the problem.

Programming: A **logic error** arises when the programmer has written code that compiles and executes without any problems, but the machine's behavior does not correspond with the original concept. For example, if the user indicates to the computer to save a file, but the computer does nothing, this is probably due to a logic error.

4.2 A simple program

To get a feel for the programming process, let us look at a very simple program, a classic program whose job is simply to say, "Hello, world." (The line numbers are for reference in this book; they are not part of the program.)

```
#include <iostream>
#include <string>

int main() {
    // this program prints the words ``hello, world'' and exits
    cout << "hello, world" << endl;
    return 0;
}
```

To compile and run this program, create a file called "`hello.cc`" containing the above program. Then tell your compiler to compile and run it. If all goes well, the computer will print the following to the screen before finishing.

```
hello, world
```

We now briefly examine this simple program, to get a general idea of what is happening. Do *not* yet become concerned about learning the details or learning how to do this yourself; we will examine this in following chapters.

Lines 1–2: Just mindlessly include these two lines in all your programs. It tells the compiler that your program may be using keyboard input or screen output, and that your program may be using character strings. (The `io` in `iostream` stand for *input* and *output*.)

Line 3: In C++, blank space (empty lines, tabs, and spaces) is not significant. The compiler uses them only as a way of separating words. But when we write programs, we often use blank space to structure ideas so that they are easier to understand. You see this with the blank line separating lines 1 and 2 from the rest of the program, and with the indentation used in later lines.

Line 4: This tells the computer that the *main* part of the program is beginning.The main part of the program will be contained in left and right braces ('`{`' and '`}`'). Actually, this line is declaring a *function*, which we discuss in Chapter 7; for now, just think of this line as being required in each program to tell the computer where execution is meant to begin.

Line 5: This is a **comment**. A comment begins with two slashes ('`/`') and continues to the end of the line. The computer ignores all comments; they exist to help humans understand what the program does.

Line 6: This line is the first functional part of the program. In this case, it says to print the words "hello, world", followed by an end-of-line (`endl`) to `cout` (this is what C++ calls the screen). The double-quotes delimit a **string**, which is a sequence of characters that the computer should treat as a single piece.

Line 7: A **`return` statement** says to halt running the program.

Line 8: This closing brace (which corresponds to the opening brace in line 4) says that this is the end of the definition of the *main* part of the program. In this case, we are defining only this part of the program, so the file ends just after this brace. (In a longer program, we might want to define more parts of the program after the closing brace.)

Exercise 4.1: Find a C++ compiler, and type the "hello, world" program.

a. Compile and run the program to see what happens. If things go well, this will demonstrate how a working program works.

b. Insert a typo by removing a quotation mark from line 6. Compile this new program to see what a compile-time error looks like.

c. Finally, insert a run-time error by replacing line 6 with the following fragment.

```
int x = 0;
cout << "300 / 0 = " << (300 / x) << endl;
```

If this compiles successfully, when you run the program, the computer should attempt to divide 300 by 0, causing a run-time error.

4.3 Tips for writing programs

Experienced programmers follow some guidelines to make their jobs easier. Their collected wisdom can be helpful to you, too, as you begin to develop programs.

Design the program on paper first. Even for small programs, take the time to decide how to best accomplish the task. Pseudocode descriptions, as we saw in Chapter 3, are useful, especially for beginners.

As you think about how to approach the problem, criteria to consider include:

Will the program be correct? In many systems (transportation and military applications, for example) absolute correctness is the primary consideration. Simple approaches tend to be easier to verify and test.

Will the program take too much time? Time is not always an issue, especially as computer speed increases, but sometimes it is. Often it is best to try the simplest method, even if you think it may be too slow; then, if your suspicions prove correct, you can write the faster but more complicated program. We will study the analysis of program speed later in this book (Chapter 17).

Will the program use too much memory? Sometimes, but rarely, memory is a limitation.

Is the approach easy to understand and program? Usually this is the most important consideration, especially for small programs. It makes little sense to spend hours writing a program if it will save only seconds the few times it is executed.

Start simple. Although the preliminary, on-paper design should consider how to extend the program, as you begin developing code it is good to begin with a small piece and to get it right before extending it.

Make your program readable. That programs be understandable is important. This is not only important so that graders can understand your assignment. It is actually more important in the workplace, where programmers often need to modify a program written by others. Readability makes a program easier to fix or enhance. A program is not worth much if it is so unreadable that a programmer must spend hours studying it to learn how to tweak a small piece.

The following are some rules for writing readable code.

Convey structure using blank space. In C++, blank space is not significant. All the examples in this book, however, are indented in a particular way. (Other indentation styles are also good, as long as they are consistent and convey structure.) We also use blank lines to separate distinct ideas. These uses of blank space make a program's structure easier to see.

Use meaningful names. When you name things in the program, the name should describe their purpose. For example, the name `n` is very poor; it means almost nothing. A more descriptive name, like `to_test`, would be better.

Include comments to structure and annotate your code. For beginners a frequent question is how many comments are necessary. Generally, line-by-line description is unnecessary. The author offers the following loose guideline: Use blank lines to break each function into paragraphs of 5–10 lines, and include a comment at the beginning of each paragraph describing its purpose.

Break large problems into subtasks. By breaking the problem into pieces, you aid your code's readability and make error-finding easier. (C++ helps you express subtasks through *functions*; we study these in Chapter 7.)

Beginning programmers often find it difficult to decide at what point to break something into subtasks. If you find yourself wanting to duplicate a segment of code, likely the duplicated code should really be part of a separate function. Another general, rough rule is that you should consider breaking your pieces so that no piece has more than about 40 lines.

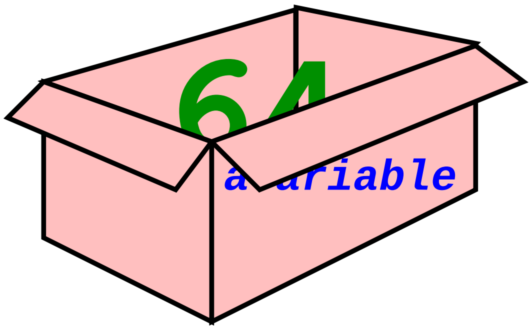
Test the program thoroughly. After you finish writing the program, or after you finish an intermediate step, you should test the program to find any errors. You should understand the program well enough and test the program thoroughly enough that you are confident that it will always work.

For all but the most simple programs, you will not always be able to try all cases, but you should be able to hit many of them. Try simple cases first, since understanding the program's behavior in these cases is easiest. (With a program involving numbers, the simple cases may include 0, 1, and -1.) After that, try extreme cases. (If the program should work for numbers up to 1000, try 1000.) Try any special cases, and then if it still works, try some common cases.

Chapter 5

Variables

A **variable** reserves a place in the computer's memory to hold some information. They're called *variables* because a program can vary them from time to time. Think of a variable as a named box that holds a value.



The picture illustrates a variable named `aVariable` which currently holds the number 64.

Variables hold information about how things are going. For example, a word processor could have a variable for each open window, a variable for the current font choice, a variable for the clipboard contents, and many more.

Much of a program involves manipulating variables and updating their values. In this chapter we learn about creating new variables in *declarations*, changing their values in *assignments*, and accessing variables in *expressions*.

First, though, we'll see several different types of data that our boxes can hold.

5.1 Basic data types

There are several **basic data types** that C++ recognizes. (We call them *basic* to distinguish them from the more complex types we study in Chapter 8.) Let's look at each of them.

`int` An **integer** is a number with no fractional part, like 0, 42, or -5. This turns out to be the most useful single type. You can refer to specific integers in C++ exactly as you would expect: "`42`", for example.

`double` This is the C++ way of saying "real number." Examples of `double`s are "`-2`", "`3.14`", and "`22.1e-3`" (which stands for the number 22.1×10^{-3} — the '`e`' stands for *exponent*). It's occasionally important to remember that, because computers only allocate so much room for storing a `double`, the representation cannot be exact for all numbers (for example, $2/3$ is actually something like 0.666666666666667). This sometimes causes a **round-off error**.

`char` A **character** is a single letter, digit, space, or punctuation mark. To express a character in C++, enclose it in single quotes: "`'C'`", for example, is the letter '`C`'. (A few characters require a backslash before them — "`'\''`", for example, represents the single-quote character, and "`'\\'`" represents '`\`'.)

`string` A **string** is a sequence of characters. A C++ program encloses a string in double-quotes ('`"`'): We saw the `"hello, world"` string in Chapter 4 already. (Technically, `string` is not a basic data type, but you can think of it as one.)

Exercise 5.1: (Solution, 118) What is the type of each of the following constants?

a. `"3.4"`	**c.** `45.0`	**e.** `-1e10`
b. `0`	**d.** `"a"`	

5.2 Declarations

A **variable declaration** tells the computer to create a new box. Associated with this box is the ability to hold data of a certain type (an `int`, `double`, or whatever).

⟨***typeOfVariable***⟩ ⟨***variableToDefine***⟩;

This creates a new variable of the specified type. For example, if our program includes the statement

```
int aVariable;
```

then we get a variable named `aVariable` for holding integers, which we can use and change as we please in the future.

A detail worth remembering	Name your variables with care. It is much easier to program and it is much easier to understand a program when variable names indicate the variables' purpose. In C++, variable names can contain letters (upper-case or lower-case), digits, and underscores ('_'). The name can have as many of these as you like, but it must not begin with a digit. Letter case is significant. (`Square` and `square` are different names, for example.)

Exercise 5.2: (Solution, 118) Which of the following are valid variable names? For those that are well-named, what type would be best for them?

a. `name`	**c.** `letter`	**e.** `#students`	**g.** `r2d2`
b. `num_points`	**d.** `char`	**f.** `temperature`	**h.** `2i`

5.3 Assignments

When a variable is created using a declaration, it doesn't have a defined value yet. (You cannot accurately predict its initial value; it may be something completely useless, like 1597.) So the program needs to give it a value for the variable to be useful.

To change the contents of a variable's box, we use an **assignment statement**. An assignment statement in C++ looks something like this.

⟨***variableToChange***⟩ `=` ⟨***valueToGiveIt***⟩`;`

Say we want C++ to change the value of `aVariable` to the number 64. Then we would use the C++ statement

```
aVariable = 64;
```

When the machine executes the machine language corresponding to this statement, it will replace whatever is in the box corresponding to `aVariable` with the number 64.

C++ also allows the declaration and assignment statements to be combined.

⟨***typeOfVariable***⟩ ⟨***variableToDefine***⟩ `=` ⟨***valueToGiveIt***⟩`;`

So instead of the above two lines, we could combine them into one:

```
int aVariable = 64;
```

A detail worth remembering	Don't let the equal sign '=' confuse you: We are not discussing algebraic equality here. The assignment statement actually *changes* the value of the variable mentioned. The statement "`k = k + 1`", for example, is entirely reasonable (even if it is algebraic nonsense): It replaces the value in the `k` box with something 1 more than it was previously. For example, if the `k` box was holding the value 1, after the statement it would hold the value 2 (which is $1 + 1$) instead.

5.4 Expressions

An **expression** is anything that can be evaluated to a value. They can turn up in many situations; one common place is on the right-hand side of an assignment. Using an expression as a statement itself is also sometimes useful.

A constant value (like "`64`" or "`'c'`") is the simplest expression; its value is the fixed value itself. A variable name (like "`aVariable`") makes another expression; its value is the value currently in the variable's box.

But we can also combine expressions using **operators**. C++ recognizes a number of different operators, some of which we'll introduce later. The most familiar operators are the arithmetic operators.

`(···)`	parentheses
`-`	negation (as in "`-x`")
`*`	multiplication
`/`	division
`%`	remainder
`+`	addition
`-`	subtraction (as in "`5-x`")

C++ observes the order of operations when evaluating expressions.

A detail worth remembering	The division operator / often causes problems when applied to two items the compiler understands to be integers. In this case, the machine will perform **integer division** — which means that any remainder will be ignored. So in C++, the value of "`3 / 2`" is 1, not 1.5 as you might hope. A decimal point (as in "`3.0 / 2`") makes the C++ compiler understand a number as a `double` instead.

Let's look at some examples of expressions. Say we already have an integer variable named `k` whose value is 2.

expression	value
`-k`	-2
`9 / k`	4
`9.0 / k`	4.5
`9 + k * (k + 1)`	15
`30 % 4`	2 (the remainder of $30 \div 4$)

A detail worth remembering	Often beginners expect the caret ('`^`') to do something useful (like exponentiation). It doesn't. In C++, the value of "`3^5`" is 6, not 243. Explaining what the caret means takes longer than it's worth. It's rarely useful anyway, so just don't use it.

Exercise 5.3: (Solution, 118) For each of the following expressions, say whether it is valid and, if so, compute its value if `x` holds the value 98.6 and `k` holds the value 42.

a. `k % 8` **c.** `k / 9` **e.** `2 k + 5`
b. `x - k * 2` **d.** `-x / 2`

5.5 Input and output

The **input/output operators** >> and << are also useful operators in expressions. Between them, they form a way for programs to communicate with the person at the computer (who we call the **user**). For printing things to the screen, the program can apply the << operator to the automatically-defined variable `cout`. (We saw an example of this in our "hello, world" program of Section 4.2.) And to read data from the user, the program can apply >> to the automatically-defined variable `cin`.

To illustrate input and output — and to illustrate the other concepts we have seen in this chapter — we examine a complete program to convert temperatures. Again, the line numbers are for reference in this book, and are not part of the actual program.

```
#include <iostream>
#include <string>

int main() {
    double fahrenheit;
    cout << "What temperature? "; // prompt user and read number
    cin >> fahrenheit;
    double celsius = (fahrenheit - 32) / 1.8; // print conversions
    cout << "It is " << celsius << 'C' << endl;
    cout << "It is " << (celsius + 273.15) << 'K' << endl;
    return 0;
}
```

Lines 1–4 and the last line are exactly the same as in the "hello, world" program of Section 4.2. The change is in lines 5–11. Once the computer compiles and runs this program, it will begin at line 5 and proceed line by line until reaching a `return` statement (which, in this case, occurs in line 11).

When we run this program, what it does looks something like the following. (Boldface indicates what the user types.)

```
What temperature? 98.6
It is 37C
It is 310.15K
```

Let us trace how it managed to do this, line by line.

Line 5: The computer begins at line 4 because this is what immediately follows the opening brace for the *main* function. In this line, the computer creates a box labeled `fahrenheit`, designated for holding numbers. At this point, the box contains some weird, useless value.

Line 6: The computer writes `"What temperature?"` to the screen, to ask the user for what to convert.

Line 7: The computer waits for the user to type a piece of data for the `fahrenheit` box. Since `fahrenheit` has the `double` type, the computer knows to read a number. Once the user types a number and presses the Enter or Return key, the computer reads it and places the corresponding number into the `fahrenheit` box.

Line 8: The computer creates a new box called `celsius` and assigns its value to be the result of subtracting 32 from the contents of the `fahrenheit` box and then dividing this difference by 1.8.

Line 9: The computer prints several things to the screen: First it prints the string `"It is "`, followed by the contents of the `celsius` box, followed by the character '`C`'; finally, it ends this line of output.

Line 10: Again, the computer prints several things to the screen. This time, it computes the expression "`celsius + 273.15`" and prints the result to the screen.

Line 11: The computer encounters the `return` statement and so stops.

What we know so far, then, is enough to write moderately useful programs. But to write genuinely interesting programs, we must learn about statements that control the flow of execution. It is these that we examine in the next chapter.

Exercise 5.4: (Solution, 118) Write a program to determine your age on January 1 of a particular year. The following should be a sample transcript; it will differ slightly if you were born in a different year.

```
It is January 1 of which year? 1999
You are 25 years old.
```

Chapter 6

Control statements

The fundamental unit of a program is a statement. We have already seen several types of statements: declaration statements, assignment statements, expressions, and `return` statements. For most tasks, we also need statements to *control* what other statements the computer executes. These are **control statements**. In this chapter, we look at examples of both types of control statements: *conditional statements* and *iteration statements*

6.1 Conditional statements

An **`if` statement** tells the computer to execute a sequence of statements only *if* a particular condition holds. This is a type of **conditional statement**, since it allows us to execute some statements only in some circumstances. In C++ an `if` statement looks like this:

```
if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfTrue⟩
}
```

This corresponds to the flowchart in Figure 6.1. The part in parentheses after the word `if` (and the parentheses must be there) is a *condition* — that is, an expression with a value of *true* or *false*. (We'll see several examples of conditions soon.) If this expression's value is *true*, then the computer sequentially executes the statements between the braces. Then it goes on to do whatever follows the braces. If the value is *false*, then the computer skips the statements in braces and goes directly to whatever follows them.

For example, consider the following code fragment.

```
double abs = num;
if(num < 0.0) {
    abs = -num;
}
cout << "Absolute value = " << abs << endl;
```

In this code fragment, we create a variable `abs`, which initially holds the value of `num`. If `num` is less than 0, then we instead put the value of "`-num`" into `abs`, and then we continue to the statement printing this as the absolute value. But if `num` is not negative, we skip the "`abs = -num`" statement and go directly to print `abs` as the absolute value (in this case, the printed value is the same as that of `num`).

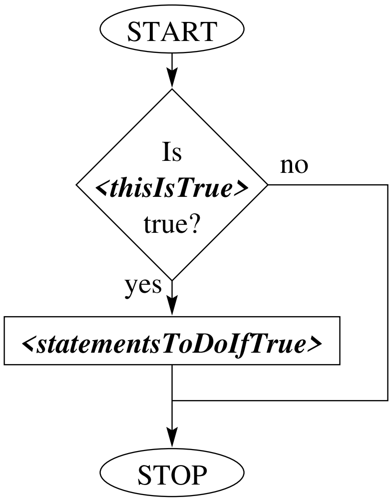


Figure 6.1: A flowchart for the `if` statement.

Conditions

A **condition** is an expression with a **logical value**, which can be *true* or *false*. In C++, a logical value is represented by an `int`: 0 represents *false* and any nonzero value represents *true*.

We have already seen one example of a condition: "`abs < 0.0`", which introduces the < operator. C++ includes operators for all six comparison possibilities.

`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	at most (less than or equal to)
`>=`	at least (greater than or equal to)

(Why does the exclamation point in `!=` mean *not*? There's not a good reason; just play along.)

A detail worth remembering	Some people at first find the distinction between the comparison operator == and the assignment symbol = confusing. Use = when you want to *change* a variable's value, and use == when you merely want to *compare* two values without changing anything. Generally you want to use == only in conditions (as in the `if` statement), while you want to use = only in assignment statements.

A detail worth remembering	One occasional pitfall when comparing `double`s is that round-off error can cause unexpected results. For example, `1 - 1.0/3.0` may not equal `2.0/3.0`, because the first may be something like `0.666667` and the second something like `0.666666`. To avoid this, when you are testing to see if two `double`s are equal, you should instead test to see if the absolute value of their difference is very small.

In addition to comparison operators, C++ includes operators for combining logical values.

`&&`	*and* (*true* if both sides are *true*)
`||`	*or* (*true* if either side is *true*, or if both sides are *true*)
`!`	*not* (*true* if expression is *false*)

rank	operators
1	`!` `-` (negation)
2	`*` `/` `%`
3	`+` `-` (subtraction)
4	`<<` `>>`
5	`<` `<=` `>=` `>`
6	`==` `!=`
7	`&&`
8	`||`
9	`=`

Table 6.1: Order of precedence for C++ operators.

A few examples illustrate how you can use these.

expression	value
`k >= 0 && k <= 3`	*true* if `k` is at least 0 and at most 3
`!(k >= 0 && k <= 3)`	*true* if `k` is not between 0 and 3
`k < 0 || k > 3`	*true* if `k` is less than 0 or greater than 3

A detail worth remembering	C++ does not provide any way to express the concept of "betweenness". If you want to see if `k` is between 0 and 3, you should use "`0 <= k && k <= 3`". (Using "`0 <= k <= 3`" doesn't work.)

A detail worth remembering	In combining multiple logical operators in an expression, you should always parenthesize to indicate the order of evaluation. Until now, the operators' order of precedence has been what you expect — multiplication precedes addition, for example. But C++ uses a weird order for the logical operators. Table 6.1 lists all the operators we have seen in their order of precedence. Notice that the *not* operator (`!`) is near the top, while the *and* operator `&&` is just above the *or* operator `||` near the bottom. Because of this weird ordering, you're best off parenthesizing every time you combine more than one logical operator.

The `else` clause

Sometimes we want to do one thing if the condition is true and another thing if the condition is false. In this case the `else` keyword comes in handy.

```
if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfTrue⟩
} else {
    ⟨statementsToDoIfFalse⟩
}
```

Figure 6.2 contains a flowchart diagramming this type of statement.

For example, if we wanted to compute the larger of two values `x` and `y`, then we might use the following code fragment.

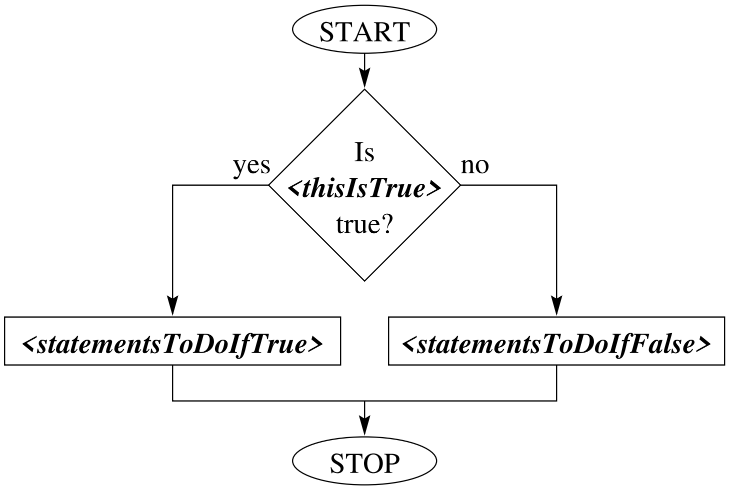


Figure 6.2: A flowchart for the `else` clause.

```
int max;
if(x > y) {
    max = x;
} else {
    max = y;
}
```

This function says place the value of `x` into `max` if `x` holds a larger value than `y`; otherwise — it says — place the value of `y` there.

Sometimes it's useful to string several possibilities together. This is possible by inserting "`else if`" clauses into the code.

```
char order;
cout << "What would you like? ";
cin >> order;
double price = 0.00;
if(order == 's' || order == 'S') {         // sandwich ordered
    cout << "Would you like fries with that?" << endl;
    price = 4.20;
} else if(order == 'f' || order == 'F') { // fries ordered
    cout << "Is that all?" << endl;
    price = 2.10;
} else if(order == 'd' || order == 'D') { // drink ordered
    cout << "Soda or pop?" << endl;
    price = 0.80;
} else {                                   // unrecognized order
    cout << "That's gibberish!" << endl;
}
cout << "That will be $" << price << "." << endl;
```

Exercise 6.1: (Solution, 119) For each of the following conditions, describe the variable values for which it is true.

a. `'a' != 'A'`
b. `x * x == x && x > -1`
c. `score > 90 || bonus && score == 89`
d. `!k == 1`

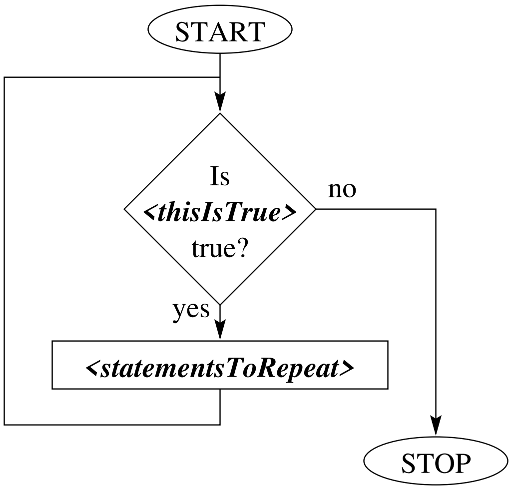


Figure 6.3: A flowchart for the `while` statement.

Exercise 6.2: (Solution, 119) Write a condition to test whether the `int` variable `year` represents a leap year. (Remember that a year is a leap year if it is a multiple of 4, except for years that are multiples of 100 but not 400. For example, 1992 and 2000 are leap years; 2100 is not.)

Exercise 6.3: (Solution, 119) Describe all the errors in the following code fragment, and write a version correcting all of them.

```
char ch;
if k = 2 {
    ch = "?";
}
```

Exercise 6.4: (Solution, 119) Write a program to tell whether an integer divides another exactly. It should behave something like the following. (You may find the modulo operator `%` useful.)

```
What is the numerator? 13
What is the denominator? 2
2 does not divide 13.
```

6.2 Iteration statements

The statements we have seen so far allow us to write programs following a top-down sequence. With conditional statements, we are able to tell the computer to sometimes skip some statements, but in all cases we can only go downward.

So we introduce **loops**, which allow us to tell the computer to execute a sequence of statements several times. They're called *loops* because they introduce loops into flowcharts. Each time through this sequence of statements is called an **iteration**; of course, a loop may iterate many times before finally proceeding to instructions past the loop.

`while` loops

The simplest C++ loop is the **`while` statement**. It is constructed exactly like an `if` statement.

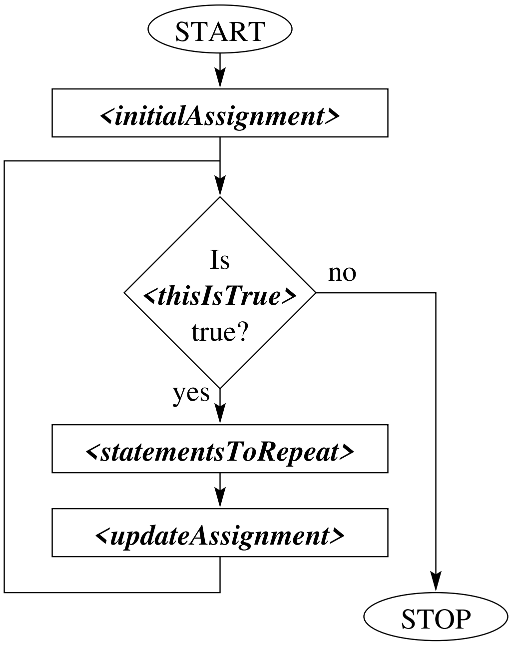


Figure 6.4: A flowchart for the `for` statement.

```
while(⟨thisIsTrue⟩) {
    ⟨statementsToRepeat⟩
}
```

The difference is that when the computer finishes executing the statements within thet braces, it retests the condition. If it still holds, the computer repeats the statements in braces again and again until it finally finishes the statements in braces and the condition no longer holds. Once the computer gets to this point, it continues to the first statement following the loop (after the closing brace). (If the condition never held in the first place, the computer skips past the loop immediately.) Figure 6.3 illustrates this process.

Let's look at a particularly useless code fragment illustrating a `while` statement at work.

```
char cont = 'y';
int i = 0;
while(cont == 'y' || cont == 'Y') {
    cout << "Iteration " << (i + 1) << ": Shall I continue (y for yes)? ";
    cin >> cont;
    i = i + 1;
}
cout << "Ok; there were " << i << " iterations." << endl;
```

Here is a sample run of this fragment.

```
Iteration 1: Shall I continue (y for yes)? y
Iteration 2: Shall I continue (y for yes)? Y
Iteration 3: Shall I continue (y for yes)? y
Iteration 4: Shall I continue (y for yes)? n
Ok; there were 4 iterations.
```

`for` loops

The **`for` loop** is a different iteration statement which is also frequently useful. It is meant for executing a sequence of statements *for* every value in a set (especially for iterating over some statements *for* every integer in a range). A `for` loop looks like the following in C++.

```
for(⟨initialAssignment⟩; ⟨thisIsTrue⟩; ⟨updateAssignment⟩) {
    ⟨statementsToRepeat⟩
}
```

This corresponds to the flowchart in Figure 6.4. The syntax here is a bit awkward, but it turns out to be quite useful. You may find it easiest to understand this by the following `while` loop which is equivalent for our purposes.

```
⟨initialAssignment⟩;
while(⟨thisIsTrue⟩) {
    ⟨statementsToRepeat⟩
    ⟨updateAssignment⟩;
}
```

As an example, let's consider a fragment to compute the factorial of a number `num`. (The **factorial** of a number is the product of all the integers up to the number; so the factorial of 6 is $1 \times 2 \times 3 \times 4 \times 5 \times 6$.)

```
int fact = 1; // we begin with 1
int i;
for(i = 1; i <= num; i = i + 1) {
    fact = fact * i; // multiply the value of i into fact
}
cout << num << " factorial = " << fact << endl;
```

This fragment begins by creating two variables `fact` (initially 1) and `i`. For the first iteration of the `for` loop, we put the value 1 into `i`. As long as `i` does not exceed `num`, we execute the statements in the loop ("`fact = fact * i`") and then add 1 to `i`. So we'll execute the loop once with `i` being 1, then once with `i` being 2, once with `i` being 3, and so on until we get to where `i` is more than `num`, at which point we are finished with the loop. Then we print out the current value of `fact` as the factorial of `num`.

If `num` happened to hold 6 as the computer begins executing this code fragment, then within the fragment the computer would print

```
6 factorial = 720
```

(You'll frequently see a variable named `i` as the variable being changed each time through the loop. The name choice is arbitrary — we could name it `a_loop_variable`, if we wanted — but many programmers name their loop variables `i` for some reason.)

6.3 Extended example

Now that we know our control statements, we can at last write genuinely useful programs. Let's look at an example program incorporating everything we've seen so far in an implementation of the Prime-Test-All algorithm from Chapter 2.

```
 1  #include <iostream>
 2  #include <string>
 3
 4  int main() {
 5      // read in a number from the user
 6      int to_test;
 7      cout << "What do you want to test? ";
 8      cin >> to_test;
 9
10      // test each possible divisor up to sqrt(to_test)
11      int i;
12      for(i = 2; i * i <= to_test; i = i + 1) {
13          if(to_test % i == 0) { // then we have found the divisor i
14              cout << to_test << " is not prime." << endl;
15              return 0;
16          }
17      }
18      cout << to_test << " is prime." << endl;
19      return 0;
20  }
```

Here's a sample run of this program.

```
What do you want to test? 25
25 is not prime.
```

Let us do a step-by-step trace to see how this came about. First the computer prompts the user (line 7) for a number and reads the number from the user (line 8). The user types **25**, so the variable `to_test` now contains the number 25. Now we create a new variable `i` before entering the `for` loop (line 11). We assign `i` to hold 2 as the `for` statement instructs.

First iteration: Since $\mathtt{i}^2 \leq 25$, we go through our first iteration. In line 13, we test to see if "`to_test % i`" is 0, which it is not (the remainder is 1); thus we continue past the statements in these braces (lines 14 and 15) to what follows them (line 17). This ends the statements within the `for` loop, so we execute the assignment "`i = i + 1`" from the `for` statement. Now `i` holds the value 3.

Second iteration: Still $\mathtt{i}^2 \leq 25$, so we go through another iteration. This time "`to_test % i`" has the value 1, so we skip the statements within the braces to line 17. This ends the statements within the `for` loop. We execute "`i = i + 1`"; now `i` holds the value 4.

Third iteration: Still $\mathtt{i}^2 \leq 25$, so we go through another iteration. This time "`to_test % i`" has the value 1, so we skip the statements within the braces to line 17. This ends the statements within the `for` loop. We execute "`i = i + 1`"; now `i` holds the value 5.

Fourth iteration: Still $\mathtt{i}^2 \leq 25$, so we go through another iteration. This time "`to_test % i`" has the value 0, so we execute the statements within the braces. In this case, the statements tell us to print that `to_test` is not prime (line 14) and then to exit the program immediately (line 15). We are thus done with our trace.

Exercise 6.5: (Solution, 119) Translate the following into a fragment using a `while` loop instead, and explain what the fragment does.

```
double total = 1;
for(i = 30; i > 0; i = i / 2) {
    cout << total << endl;
    total = 2 * total;
}
```

Exercise 6.6: (Solution, 119) Describe all the errors in the following code fragment, write a version correcting all of them, and describe what the corrected version does.

```
for(i == 30, i != 0, i == i + 1) {
    cin << num
    product = product * num
}
```

Exercise 6.7: Write a program to help balance a checkbook. A run of the program should look like this.

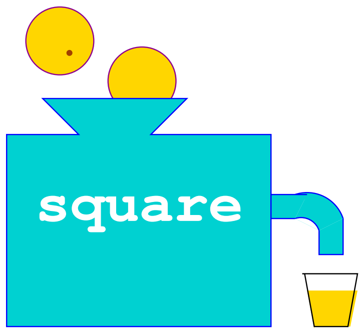
```
To add? 30.25
+ 30.25 = 30.25
To add? -20.30
- 20.3 = 9.95
To add? 998.23
+ 998.23 = 1008.18
To add? -447.87
- 447.87 = 560.31
To add? 0.0
```

The user should be able to type as many entries as desired; but when the user types zero, the program exits. (Using zero this way allows you to read the user-typed value into a `double` instead of doing something more complicated.)

Chapter 7

Functions

A **function** is a packaged sequence of statements to accomplish a certain task. A useful analogy is to think of a function as a sort of juicer.



A function takes some **parameters** and produces a **return value**. In our juicer analogy, the fruit would be parameters, and the juice produced would the return value. If you give it oranges, it makes orange juice. Give it lemons, and it makes lemonade.

One purpose of a function is to allow the programmer to describe a procedure once, even if the program executes the procedure many times. For example, maybe your program determines whether a number is prime in many places. By making a function to test primality, you can write your algorithm only once (as the machinery for that function) and then easily use this function in many places. This simplifies the task of writing the program, but more importantly it makes modifying the primality-testing method very easy later.

Another major purpose of functions is to decompose a program into subtasks. Even if there is only one location in your program where you decide whether a number is prime, it may be useful to make it a separate function anyway, just because in the larger program the details of exactly how you determine this are irrelevant. By using functions in this way, a program becomes easier to read, and it becomes possible to write much larger programs.

7.1 Function calls

To use a function, a program uses **function calls**. A function call is a part of an expression, which begins with the name of the function being called, followed by parentheses with the parameter value placed within the parentheses.

For example, one of the functions that is already built into C++ is the `abs()` function. (The name is actually just `abs`; but to distinguish functions from variables, conventionally function names are written with parentheses.) It takes one integer as a parameter and returns that integer's absolute value. The following line of code illustrates a call to `abs()`.

```
cout << x << " and " << y << " are " << abs(x - y) << " apart." << endl;
```

In this case, we have used the `abs()` function with the value of the expression "`x - y`" as its parameter. So if `x` held 1 and `y` held 5, then `abs()` would be called with its parameter equal to -4, and the output of this code fragment would be

```
1 and 5 are 4 apart.
```

7.2 Function definitions

Actually, we have already seen several function definitions: Each of our programs defines a function called `main()`. Until now, however, we have ignored this as necessary verbiage; now we examine exactly what is happening.

A function definition looks like the following.

⟨***returnValueType***⟩ ⟨***functionName***⟩`(`⟨***parameterList***⟩`) {`
    ⟨***statements***⟩
`}`

Let's break apart a very simple example to illustrate this.

```
double square(double to_square) {
    return to_square * to_square;
}
```

`double` This definition begins with the word `double`, saying that we are about to define a function whose return value is a `double`.

(C++ includes a special return type called `void` for functions returning nothing useful. Of course the return value of such a function is not useful, but we may still want to call the function if it has other useful effects like printing information to the screen. We'll see examples of this later.)

`square` Then we find the name of the function — `square` in this case. The rules for function names are the same as for variable names. Always think carefully about how to name your functions so as to best communicate the function's purpose.

`(double to_square)` In the parentheses are the function's *parameters* — the lemons for our juicer. In this case, the function takes one parameter, which is of type `double`; within the function, we refer to its value with the name `to_square`. The rules for parameter names are the same as for function names and for variable names, and as for them you should try to use names that describe their purpose.

–...˝ The brace characters surround the statements that say what the function does. This is called the **function body**. This corresponds to the machinery within the juicer.

`return to_square * to_square;` In our example, the function body is a single statement, which is a **`return` statement**. When the computer gets to a `return` statement, it stops working on the function. The value of the expression between the word `return` and the semicolon is used as the function's return value (the lemonade, according to our analogy). In this case we want the return value to be the square of `to_square`, so our return value is `to_square` multiplied by itself.

7.3 Extended example

To illustrate how functions work, we trace through a complete program including a function.

```
 1  #include <iostream>
 2  #include <string>
 3
 4  int fact(int what) {
 5      int ret = 1; // this will be the factorial of what
 6      int i;
 7      for(i = 1; i <= what; i = i + 1) {
 8          ret = ret * i; // multiply the value of i into ret
 9      }
10      return ret;
11  }
12
13  int main() {
14      int n;
15      int r;
16      cout << "Choose how many of how many? ";
17      cin >> r >> n;
18      cout << (fact(n) / fact(r) / fact(n - r)) << " choices" << endl;
19      return 0;
20  }
```

Here is a sample run of this program.

```
Choose how many of how many? 2 6
15 choices
```

Let's see how this came about.

Lines 14–17: We begin at the beginning of `main()`. We create two variables `n` and `r` and wait for the user to give their values. Now `r` contains 2 and `n` containts 6.

Line 18: In order to compute the first expression to be printed, we compute the value of "`fact(n)`". Since `n` holds 6, we call `fact()` with the parameter `what` holding 6.

Lines 5–9: We run through the code of `fact()` with `what` holding 6. This code multiplies all the integers between 1 and 6 together; we finally reach line 10 with `ret` holding 720.

Line 10: We return the value of `ret` (that is, 720) and continue with executing line 18.

Line 18: Now that we know the dividend is 720, we compute the divisor. Again, we call `fact()`, this time with `what` holding what `r` holds, the value 2.

Lines 5–10: We run through `fact()` with `what` holding 2. When we reach line 10, `ret` holds 2 and so 2 is the return value.

Line 18: Now that we have computed the first two calls, we divide to get $720/2 = 360$. But we still have to perform another division. We know the dividend is 360; to get the divisor, we call `fact()` again, this time with `what` holding the value of "`n - r`", which is 4.

Lines 5–10: We run through `fact()` one more time, this time with `what` holding 4. The return value is 24.

Line 18: Now that we have 24 from `fact()`, we perform the second division to get $360/24 = 15$. This is the first number printed. Then we print `" choices"`, followed by an end-of-line.

Line 19: We reach the `return` statement in `main()`, so we are finished with running the program.

7.4 Parameters and variables

The variables available within a function are exactly those declared within the function. The code within a function cannot see variables defined in other functions. For this reason, whenever you want to call a function, you should include among its parameters any information that the function needs.

Often we want a function with several parameters. In this case we list the parameters, separated by commas. For example, we might want a function `choose()`.

```
int choose(int n, int r) {
    return fact(n) / fact(r) / fact(n - r);
}
```

To call such a function, you list the expressions for the arguments in the same order they are defined, separated by commas.

```
cout << choose(6, 2) << " choices" << endl;
```

This will call our `choose()` function with the parameter `n` holding the value 6 and the parameter `r` holding the value 2.

Note that when we call a function, values are copied into the parameters. So if we happen to change a parameter's value, this does not alter anything outside the function. As an example, consider the following program.

```
void setToZero(int n) {
    n = 0;
    return;
}

int main() {
    int i = 1;
    setToZero(i);
    cout << i << endl;
    return 0;
}
```

(Remember that C++ uses `void` as the return value type for functions that do not return anything useful.) This program will print the value `1`. Setting the value of `n` to 0 in `setToZero()` has no effect on the value of `i` in `main()`. This system of parameter passing is called **call-by-value**.

C++ also allows for **call-by-reference** parameters. This alters the behavior so that changes to the parameter *do* effect the variable passed to it. You can indicate a call-by-reference parameter using an ampersand '`&`' just before the parameter name. For example, we can instead define `setToZero()` as follows.

```
void setToZero(int &n) {
    n = 0;
    return;
}
```

When we use this in place of our earlier `setToZero()`, the change of parameter value *does* affect the value of `i` in `main()`, and so the program prints `0`. (Of course, whenever you use a call-by-reference variable, you must pass something whose value can be changed. You can't use "`setToZero(2)`" in an attempt to change 2, for example.)

A final type of parameter passing that C++ provides is the **constant-reference parameter**. Here the value is not copied to the parameter, but C++ does not allow the programmer to change the value of the parameter. You can indicate a constant-reference parameter by including the word `const` beforehand.

```
void setToZero(const int &n) {
    n = 0;  // this now causes a compile-time error
    return;
}
```

This fragment will cause a compile-time error because the statement "`n = 0`" attempts to alter the value of the constant-reference parameter `n`.

Exercise 7.1: Define a function to find the greatest common divisor of two numbers, and use this to modify the checkbook exercise of Exercise 6.7 to work with fractions rather than `double`s. The program should keep the total in lowest terms.

```
To add? 5 6
+ 5/6 = 5/6
To add? 3 4
+ 3/4 = 19/12
To add? 5 12
+ 5/12 = 2/1
To add? 0 0
```

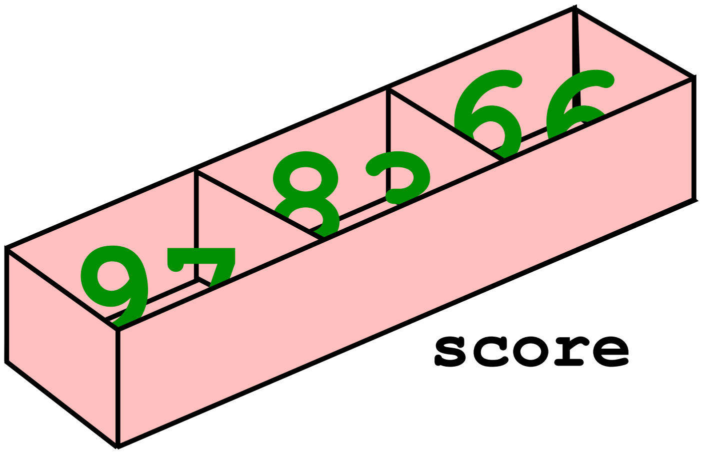
Chapter 8

Complex data types

Besides allowing for variables holding only a single item of data (a number or character), C++ allows variables to hold conglomerations of data. These allow programs to work with more massive data.

8.1 Arrays

An **array** holds a sequence of values of the same type. This is especially useful when you want to store a large group of related data, like data points in a graphing program or students' scores in a gradebook program. Each individual value in the array is called an **array element**.



You can declare a new variable to be an array using the following format.

vector<⟨***typeOfElement***⟩> ⟨***variableToDefine***⟩(⟨***lengthOfArray***⟩);

For example, the following creates an array named `score` to hold 3 numbers.

```
vector<double> score(3);
```

More generally, the array length can be any expression ("`2 * num_students`" instead of "`3`", for example).

To work with an array, we must refer to individual elements using their **array indices**. The array elements are automatically numbered from 0 up to one less than the array length.

A detail worth remembering	Yes, that's *one less than the array length*. So if you declare an array `score` of length 3, the array indices are 0, 1, and 2. C++ always begins at 0. (This turns out to be more convenient than the intuitive choice of 1.) If you try to access an undefined array index, the program may behave weirdly (unexpectedly crash, for example), so be careful with array indices.

To refer to an array element in an expression, type the array variable name, followed by the element's array index enclosed in brackets. You can also do this on the left-hand side of an assignment to alter an array element's contents.

```
vector<double> score(3);
score[0] = 97.0;
score[1] = 83.0;
score[2] = 66.0;
cout << "Average = " << ((score[0] + score[1] + score[2]) / 3.0) << endl;
```

In these statements we create an array of three numbers, called `score`. We assigned its three boxes to hold three test scores, 97, 83, and 66. And finally we printed the average of these. The computer will display `82`.

Parameters can be arrays too. Use the type `vector<`$\langle$***typeOfElement***$\rangle$`>` to indicate that a parameter is an array,

A detail worth remembering	When you pass arrays as parameters to a function, they should be either reference parameters (if you want to change the elements) or constant-reference parameters (if you do not). Do not pass arrays as call-by-value parameters; the inefficiency of copying arrays is too large to ignore.

Let's look at an implementation of the Mode-Tally algorithm we discussed in Chapter 3. This implementation combines everything we have seen about arrays.

```
 1  int modeTally(const vector<int> &score, int num, int max) {
 2      vector<int> tally(max + 1);
 3      int i;
 4      for(i = 0; i <= max; i = i + 1) { // set all tallies to 0
 5          tally[i] = 0;
 6      }
 7      for(i = 0; i < num; i = i + 1) { // tally up the scores
 8          tally[score[i]] = tally[score[i]] + 1;
 9      }
10      int mode = 0; // find the most-frequently-occurring score
11      for(i = 1; i <= max; i = i + 1) {
12          if(tally[i] > tally[mode]) {
13              mode = i;
14          }
15      }
16      return mode;
17  }
```

Line 1: Here we have an array of integers named `score` as one of our parameters. (Notice that `score` is a constant-reference parameter here.) We have another parameter `num` to tell the function how many elements `score` has.

Line 2: We define a new array `tally` to hold "`max + 1`" integers. We use "`max + 1`" rather than just "`max`" because scores can range from 0 to `max`, and we want a tally box for each one of these possibilities.

Lines 3–6: Just after creating an array, the array element contents are undefined. We want all the tally boxes to be initially empty, so we go through the array and set all elements to zero.

Lines 7–9: Now we tally up the scores. For each array index i for the `score` array (and the indices on `score` range from 0 to `num - 1`), we add 1 to tally box `tally[score[i]]`. (You may find it a bit weird to have array indices within array indices, but this is perfectly legitimate. If it helps to understand what is going on, there is no problem with separating this statement into two, the first defining a variable `x` to be "`score[i]`" and the second adding 1 to tally box `tally[x]`.)

Lines 10–16: Finally we go through `tally` to find which entry is the largest. We use `mode` to hold the largest entry found so far. We begin with this being 0 (we see only the first array element), but as we go along the array we compare more elements to the best we have found so far. When we see something better, we store the index of that element in `mode` instead.

Exercise 8.1: (Solution, 120) Write a function `removeDuplicates()` to replace the elements of an array of `ints` with the sequence with adjacent duplicates removed. The function should return the number of elements in the new sequence. For example, given the array $\langle 3, 7, 7, 7, 8, 3, 3\rangle$, the function should replace the elements with $\langle 3, 7, 8, 3\rangle$ and return 4.

Exercise 8.2: (Solution, 120) Write a program that computes the median of a set of integer test scores between 0 and 100. (The **median** is an element that falls in the middle if we list the scores in sorted order.) Here is a sample transcript.

```
How many numbers? 5
#1: 83
#2: 32
#3: 83
#4: 71
#5: 65
Median = 71
```

8.2 Strings

The `string` type is quite similar to the `vector` type: Each individual element of the `string` is a character, and you can access individual characters using brackets.

In addition, you can get a string's length (the number of characters in it) by using the `length()` function. The following fragment illustrates all these aspects at work.

```
string name;
cout << "Who are you? ";
cin >> name;
cout << "Hello, ";
for(int i = 0; i < name.length(); i = i + 1) {
    if(name[i] == '_') {
        cout << ' ';
    } else {
        cout << name[i];
    }
}
cout << "!" << endl;
```

This fragment reads a word the user types into the variable `name`. (When you use the input operator >> with a string, the string it reads includes the characters up to the first space or return character. Subsequent letters are left for later inputs.) After reading this word, the fragment prints out the name, but prints spaces in place of the underscore characters ('_').

```
Who are you? J_Edgar Hoover
Hello, J Edgar!
```

You can also compare strings with operators like < and == just as you compare numbers. The ordering is similar to a dictionary's, but all the upper-case letters precede all the lower-case case letters. So "`"Washington" <= "cherry"`" has a value of *true*.

Exercise 8.3: (Solution, 121) Write a program that gets a word that the user types and then tells the user whether or not the word is a palindrome. (A **palindrome** is a sequence of letters that read the same forwards and backwards. Examples include *redder* and *deified*.)

8.3 Structures

It's also sometimes useful to have variables conglomerating several very different pieces of data. For example, we may want a variable to represent a student. A student may have an ID number, a raw score, and a letter grade. To make a single variable to contain all this data, you can use a *structure*.

A **structure** is a programmer-defined type combining several other pieces of data. You can use the `struct` keyword to define a new structure.

```
struct ⟨nameOfStructureType⟩ {
    ⟨elementDeclarations⟩
};
```

The structure definition should appear outside function bodies.

```
struct StudentType {
    int id;       // student id
    double score; // raw score
    char grade;   // letter grade ('A', 'B', 'C', 'D', or 'F')
};
```

A detail worth remembering	Notice that a structure definition introduces a semicolon after the closing brace, in contrast to the rules we have seen elsewhere in C++. It's easy to forget that semicolon, but you must include it to avoid confusing the C++ compiler.

Once you have defined a structure, you can easily create variables of that new type.

```
StudentType stud;
```

In an expression or an assignment statement, you can refer to individual data elements within the structure by using the dot operator `.`.

```
stud.id = 42;
stud.score = 98.3;
stud.grade = 'A';
```

Of course there's no problem with using arrays of structures, representing perhaps an entire array of students.

```
int findMaxScore(const vector<StudentType> &student, int num_students) {
    int max = 0;
    int i;
    for(i = 1; i < num_students; i = i + 1) {
        if(student[i].score > student[max].score) {
            max = i;
        }
    }
    return student[max].id;
}
```

This function takes an array of students as its parameter. It determines the index of the student with the highest score (exactly as we determined the maximum tally in `modeTally()`), and it returns that student's ID number.

Note that, just as an array parameter should be either a reference parameter or a constant-reference parameter, so should a structure parameter.

Exercise 8.4: (Solution, 121) Write a C++ fragment to define a new type representing a library book. The relevant data to include in this structure are the book's name, ID number, the price, and the due date.

Chapter 9

Objects

In large software projects, a primary problem is how to structure and decompose the problem into more manageable brain-sized chunks. Software engineers use a variety of techniques to do this. One fairly recent and very popular technique is **object-oriented design**. In this chapter we look at what this means, and we examine how it works in practice using a particular language designed for object-oriented designs, C++.

9.1 Object-oriented design

The main insight motivating the idea of object-oriented design as that most tasks involve the manipulation of objects. This is certainly true in the real world, and object-oriented design proposes that it holds for computing too. If this is true, then a natural way to structure a program is to define separately each of the objects involved in the task. If these are designed properly, then the program to manipulate the objects should be short and straightforward.

As an example, consider a program for drawing graphs. The program manipulates several objects related to the interface: a menu, individual menu items, a canvas for the graph, a palette of colors, and maybe a toolbox. The program also manipulates objects related to the graph itself: the x-axis, the y-axis, data points, data series, labels, and a legend. And the program manipulates objects for handling graph data: a spreadsheet, files, the printer. In an object-oriented design, each of these objects would be a candidate for being a separate component of the program.

A single component defines a single object. An object has two basic pieces — its state and its interface. The **state** of the object is the information associated with it. The **methods** of the object are the actions the object can perform. Together, these are the **members** of the object. Good designs usually prevent the state from being changed except through the methods. Think of a Rubik's cube: the state would be the current configuration, and the methods would include actions for twisting and turning the cube. The cube does not allow tweaking individual pieces of the state, so the object should not allow the program to do this by changing the object's state directly.

To return to our graphing example, a data point object may have the x- and y-coordinates as its state. Among its methods may be a method for drawing the point on the screen and a method for moving the point.

Deciding where different functionality lies is a matter of taste. Is it the data point's job to draw itself, or is this the purview of the data series, or should the canvas draw it? The designer must decide which is most natural. Object-oriented design then, does not make program design simple: It is a guiding paradigm, which provides guidelines but not answers about how to structure a design.

9.2 Defining objects

A programming language can provide support for object-oriented programs. C++ is one of these languages; this is one of the features most distinguishing it from C, BASIC, and Pascal.

Defining an object's type

To define an object in C++, you first define the object's type, whic specifies its behavior. An object type is called a **class**. Like a structure, this new data type is an alternative to the basic data types we have already seen of `int`, `char`, of `double`. C++ calls one of these types a **class**. An example of a class definition is the following.

```
class DataPoint {
public:
    double my_x;
    double my_y;
    void draw(Graphics &dest);
};
```

This **class definition** should appear outside any function bodies in the program. Here we have defined an object class called *DataPoint*. It has two data members associated with it, called `my_x` and `my_y`. And it has one method, called `draw()`. We haven't defined what this method does yet; we'll get to this soon.

Defining an object

Now that we have defined an object's class, it is time to create an individual object. When we create an object, we set aside some memory to hold that object's state.

```
DataPoint pt;
```

Usually we want the object to define what its initial state is. To do this we include a **constructor** in the class definition. The constructor is called automatically when a new object is created. To define it, we create a method with the same name as the class and without any return type. It can take parameters.

```
class DataPoint {
public:
    DataPoint(double x, double y);
    // declarations of other data and methods
};
```

If the constructor takes parameters, then we must include them when we create an object.

```
DataPoint pt(3, 4);
```

```
class DataPoint {
public:
    double my_x;
    double my_y;

    DataPoint(double x, double y);
    double getX();
    double getY();
    void draw(Graphics &dest);
    void move(double x, double y);
};

DataPoint::DataPoint(double x, double y) {
    my_x = x; my_y = y;
}

double DataPoint::getX() { return my_x; }

double DataPoint::getY() { return my_y; }

void DataPoint::draw(Graphics &dest) {
    dest.drawRect(my_x - 1, my_y - 1, 3, 3);
}

void DataPoint::move(double x, double y) {
    my_x = x; my_y = y;
}
```

Figure 9.1: The `DataPoint` class.

Accessing an object

To access an object, we use a period. To draw `pt` in the `Graphics` object `graph`, for example, we would include the statement

```
pt.draw(graph);
```

This will call the `draw()` method defined in the `DataPoint` class, with the parameter `dest` being `graph`. (As with arrays and structures, when you pass an object as a parameter, you should either make it a call-by-reference paramater or a constant-reference parameter.)

When the computer calls a method (such as `draw()`), it includes information about the object (such as `pt`) on which the action is to operate. Within the method, we can refer to individual members of this object directly without specifying them as part of the object.

With this in mind we create our `draw()` method:

```
void DataPoint::draw(Graphics &dest) {
     dest.drawRect(my_x - 1, my_y - 1, 3, 3);
}
```

(This definition would appear outside the class definition.) Here we have indicated that the action of drawing a point should be completed by drawing a 3×3 square centered on the point's coordinates. Figure 9.1 contains a complete definition of a `DataPoint` class, including the constructor.

9.3 Additional object concepts

The definition of objects is the most basic and important way that C++ and other object-oriented languages provides support for objects. But it also includes other features.

Multiple files

To help modularize a larger program, we will want to split it across files. A natural way to do this in object-oriented designs is to designate a file for each object's definition.

In C++, when we want to have a file for an object, we include both a **header file** and a **definition file**. The header file contains the type definition (the `class` block), while the definition file contains definitions of the methods. The header file's name is typically something like "`DataPoint.h`", while the definition file is typically something like "`DataPoint.cc`".

When we have a file including code using the class definition (and this includes the definition file), we must tell the compiler this. We do this using the `#include` statement at the file's beginning.

```
#include "DataPoint.h"
```

(This is similar to what we have blindly included earlier; there we were telling the compiler that we wanted to use the input and output objects, and that we wanted to use the `string` object. Now we place the file name in quotes rather than angle brackets because the file is not compiler-supplied.)

Protecting information

In the Rubik's cube example of Section 9.1, we noted that generally the program should not alter the state of an object without using the object's interface. C++ provides support for this with its `private` keyword. When we define a data member or method `private` instead of `public`, we disallow that member to be used outside the object's methods.

Good programming practice suggests that all state information (the data members) should be declared as `private`. If the program does this, then a programmer can later change the implementation of the object without worrying about how the rest of the program uses the object, provided that the interface remains unchanged. In our `DataPoint` example, we would instead declare the class as follows.

```
class DataPoint {
private:
    double my_x;
    double my_y;

public:
    DataPoint(double x, double y);
    void draw(Graphics &dest);
    double getX();
    double getY();
    void move(double x, double y);
};
```

Later circumstances may make us decide that it is more convenient to define a data point using polar coordinates, as the distance from the origin and the angle from the x-axis. If `x` and `y` are private members, then we can do this safely by modifying only the `DataPoint` definition. We can ignore the rest of the program, since it does not use `x` and `y` directly.

Extending an object

In larger programs, we may find that some objects are just specific instances of different types. We may decide, for example, that a pencil is just a special type of drawing utensil. Or buttons and checkboxes in the user interface are both special types of user-input devices. We might hope that we can write code that applies to both. C++ and other object-oriented languages provide support through this through the mechanism of **inheritance**. This is a complicated mechanism; we will touch on C++'s support of inheritance only briefly.

We call a class that inherits from another a **subclass**, and the class it inherits from we call a **superclass**. So `Pencil` would be a subclass of `DrawingUtensil`.

In C++, we define a subclass just as a regular class, but we add a colon to indicate what it is a subclass of.

```
class Pencil : public DrawingUtensil {
private:
    int uses_left;

public:
    Pencil();
    void sharpen();
    void draw();
};

Pencil::Pencil() { uses_left = 10; }

void Pencil::sharpen() { uses_left = 10; }

void Pencil::draw() {
    if(uses_left > 0) {
        DrawingUtensil::draw();
        uses_left = uses_left - 1;
    }
}
```

In our subclass definition, we have added some functionality to the pencil. In particular, it includes some new state indicating how sharp the pencil is. We have added a method to sharpen the pencil. And we have inherited any other methods of `DrawingUtensil`. If `DrawingUtensil` has method `erase()`, for example, then we can call `erase()` on `Pencils` too.

One of these methods, `draw()`, we have overridden so that it only draws if the pencil has been sharpened recently. This implementation happens to use the `DrawingUtensil` `draw()` method; it indicates that it wants to use this method (and not `Pencil`'s `draw()`) by using the `::` operator.

In many cases it is natural to have subclasses of subclasses. A `RedPencil` might be a subclass of `Pencil`, for example. We can draw a tree of the different subclasses; this is called an **inheritance hierarchy**. The inheritance hierarchy is a good way to illustrate the structure of an object-oriented program's design.

Conclusion

Object-oriented design is a useful paradigm in many situations. Some people claim that it is the best paradigm for all programs. This seems to be an exaggeration, but object-oriented design has proven useful in many instances. User interfaces (an important component of most commercial software) are particularly conducive to object-oriented design.

Object-oriented languages (like C++) can aid in developing programs with object-oriented designs. These languages add features like data protection and inheritance, which are awkward to simulate in other languages.

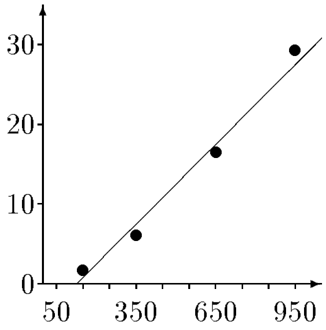


Figure 9.2: Input for line-fitting program.

In any case, structure is essential in very large projects. Design paradigms — like object-oriented design — are natural and very helpful for producing well-structured programs.

We close with a longer program illustrating object-oriented design, in Figure 9.3. This program computes the least-squares fit to a series of data. Here is a sample run of this program for the input of Figure 9.2.

```
How many points?
4
Point 1 (separate numbers with space):
150 1.6
Point 2 (separate numbers with space):
350 6
Point 3 (separate numbers with space):
650 16.4
Point 4 (separate numbers with space):
950 29.24
slope    : 0.0349034
intercept: -5.01429
corr r^2 : 0.986412
```

Exercise 9.1: Define a `Rational` class and use it to convert the checkbook program of Exercise 7.1 to an object-oriented design.

```
#include <iostream>
#include <string>

class DataSeries {
private:
    int num;       double x_sum;  double y_sum;
    double xx_sum; double xy_sum; double yy_sum;

public:
    DataSeries();
    void addPoint(double x, double y);
    double getSlope();
    double getIntercept();
    double getCorrelation();
};

DataSeries::DataSeries() {
    num = 0;      x_sum = 0.0;  y_sum = 0.0;
    xx_sum = 0.0; xy_sum = 0.0; yy_sum = 0.0;
}

void DataSeries::addPoint(double x, double y) {
    num = num + 1;           x_sum = x_sum + x;       y_sum = y_sum + y;
    xx_sum = xx_sum + x * x; yy_sum = yy_sum + y * y; xy_sum = xy_sum + x * y;
}

double DataSeries::getSlope() {
    return (xy_sum - x_sum * y_sum / num) / (xx_sum - x_sum * x_sum / num);
}

double DataSeries::getIntercept() {
    return (y_sum / num) - getSlope() * (x_sum / num);
}

double DataSeries::getCorrelation() {
    double xy_var = xy_sum - x_sum * y_sum / num;
    double x_var  = xx_sum - x_sum * x_sum / num;
    double y_var  = yy_sum - y_sum * y_sum / num;
    return (xy_var * xy_var) / (x_var * y_var);
}


int main() {
    // let num_pts be the number of points
    int num_pts; cout << "How many points? "; cin >> num_pts;
    while(num_pts < 2) {
        cout << "At least two are required. How many? "; cin >> num_pts;
    }

    // add the points to the data series
    DataSeries series;
    for(int i = 0; i < num_pts; i = i + 1) {
        cout << "Point " << (i + 1) << " (separate numbers with space): ";
        double x; double y;
        cin >> x >> y;
        series.addPoint(x, y);
    }

    // print statistics
    cout << "slope    : " << series.getSlope() << endl;
    cout << "intercept: " << series.getIntercept() << endl;
    cout << "corr r^2 : " << series.getCorrelation() << endl;
}
```

Figure 9.3: A program to fit a line.

THIRD UNIT

Recursion

In the third unit of this text, *Recursion*, we look at programming from a slightly more abstract level. We begin in Chapter 10 by extending our programming tools to encompass the concept of *recursion* — that is, functions that use themselves. This concept is very intuitive and allows simple procedures to accomplish complex tasks effectively.

In Chapter 11, we look at the specific task of playing games. Game playing is a huge success of the study of artificial intelligence, and we study many of the most successful game-playing techniques. In our study, we find that recursion plays an important role in game playing. This study also sets the stage for other abstract computer science concepts coming in the following units.

Chapter 10

Recursion

Recursion is a powerful technique, often giving impressive results through simple expressions that are otherwise quite complex. In this chapter, we examine recursion through three specific examples: a definition of Jews, exponentiation, and the Tower of Hanoi puzzle.

10.1 Definition

Recursion is the concept of well-defined self-reference.

Definitions are often recursive. Consider, for example, the following hypothetical definition of a Jew. (We examine this definition only because of its interesting structure. I don't vouch for its validity — I just heard it at a party once.)

> Somebody is a Jew if his or her mother is a Jew.

This definition is **self-referential** because it relies on itself for a definition. This definition has a problem, though; do you see it?

One problem that sometimes comes up with self-referential definitions is that they are circular. For example, "A rose is a rose" is circular. The Jewishness definition would also be circular if it were possible for somebody to be their own mother or their own maternal grandmother (or further down the line); then somebody's Jewishness might depend on her own Jewishness. Barring science-fiction time anomalies, however, this is impossible.

The problem with the definition is that it is missing a **base case**. There has to be at least one person whose Jewishness does not rely on her mother; otherwise, we have a problem of infinite regress: I'm Jewish if my mother is Jewish; my mother is Jewish if her mother is Jewish; she is Jewish if her mother is Jewish; and so on. We never stop. This problem is easy to fix.

> Somebody is a Jew if she is Abraham's wife Sarah, or if his or her mother is a Jew.

So if I want to know if I am a Jew, I look at this definition. I'm not Sarah, so I need to know whether my mother is a Jew. How do I know about my mother? We look at the definition again. She isn't Sarah either, so we ask about her mother. We keep going back through the generations — recursively — until we arrive at Sarah.

We can translate this procedure for determining whether somebody is a Jew into pseudocode.

Algorithm Is-A-Jew$(person)$
if $person =$ Abraham's wife Sarah, **then:**
  return `true`.
else:
  return Is-A-Jew$(person$'s mother$)$.
end of if

This is a *recursive function*, since it uses itself to compute its own value. Every recursive function *must* have a base case. That is, it must have some case (in this example, when $person$ is Sarah) when the function does not call itself recursively. A function without a base case will keep calling itself and will never get around to returning a value. The program will either crash or it will continue until an external effect stops it; it will certainly not find the right value.

Notice that Is-A-Jew still has a problem. What if I am not a Jew? Then we'll ask about my mother, then her mother, then her mother, and so on. We'll never reach Sarah, and the list of mothers will go much further back: We'll never stop. (In this case, we'll crash at some point (maybe when we get to Eve).)

The problem is that in this example we need more than one base case. Here is a repaired version.

Algorithm Is-A-Jew$(person)$
if $person =$ Abraham's wife Sarah, **then:**
  return `true`.
else if $person$ was born before Sarah was born, **then:**
  return `false`.
else:
  return Is-A-Jew$(person$'s mother$)$.
end of if

As this example demonstrates, recursion can involve subtle problems, but it's often useful or even essential.

10.2 Exponentiation

Now we'll look at a very different, very practical problem: exponentiating a number. That is, given a number x and a nonnegative integer n, we want to find x^n.

A C++ implementation

This time, we'll use C++ rather than pseudocode. A recursive function in C++ is written just as you would expect: Call the function exactly as you would any other function.

Notice that when $n > 0$, we have

$$x^n = x \cdot x^{n-1} .$$

This suggests that we might compute x^n by first computing x^{n-1} (using recursion) and then multiplying it by x. We can implement an `exponentiate()` function doing exactly this.

```
double exponentiate(double x, int n) {
    if(n == 0) {
        return 1.0;
    } else {
        return x * exponentiate(x, n - 1);
    }
}
```

This is simple, but it is rather poor for large n. Notice that if we want to take something to the 1000th power, the computer will go 1000 levels deep into the recursion. This takes a while, and it extends the resources of computers, which are often not designed to handle that many layers of function calls.

A faster implementation

Fortunately, recursion suggests a faster way by noticing a different fact about exponents: If n is even, then $x^n = (x^2)^{n/2}$. And if n is odd, then $x^n = x \cdot (x^2)^{(n-1)/2}$.

We can use this fact to write a new solution.

```
double exponentiate(double x, int n) {
    if(n == 0) { // base case
        return 1.0;
    } else if(n % 2 == 0) { // then n is even
        return exponentiate(x * x, n / 2);
    } else { // then n is odd
        return x * exponentiate(x * x, (n - 1) / 2);
    }
}
```

How deep does the recursion go for this new version of `exponentiate()`? Here's a way to bound it: Notice that each time we go one level deeper in the recursion, the value of n at the new level is at most half of what it was. You can see that this will always be true by looking at our definition of `exponentiate()`. (When n is even, the value at the next level is exactly half; when n is odd, the value is a little less.) Therefore, if we go $\log_2 n$ levels deep, the exponent at that level is *at most*

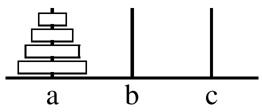
$$n\left(\frac{1}{2}\right)^{\log_2 n} = n\,\frac{1}{2^{\log_2 n}} = n\,\frac{1}{n} = 1\,,$$

where n is the exponent at the top level. When we go one more level deep, the exponent will become 0 and we will have reached the base case. So the deepest the recursion will ever go is $1 + \log_2 n$ levels.

This is much faster than the n levels we saw with our first algorithm. For example, taking something to the 1000th power required going down 1000 levels of recursion, which seemed a bit unreasonable; now we go only 10 levels down (at most).

10.3 Tower of Hanoi

In the Tower of Hanoi puzzle, we have three pegs and several disks, initially stacked from largest to smallest on the left peg. We'll refer to these disks by the numbers 0 through 3 (3 being the largest). For example, the four-disk puzzle is the following.



Our goal is to move the entire tower from the left peg to the middle peg, but we can only move one disk at a time and we can never place a larger disk on a smaller one. (You should try to figure this out on your own before continuing.)

According to folklore*, a 64-disk version of the puzzle lies in a Hanoi monastery, where monks work continuously toward solving the puzzle. When they complete the puzzle, the world will come to an end. This brings up two crucial questions on which the future depends:

- How should the monks solve the puzzle? That is, how can we write a program for solving the puzzle?
- If the monks use our program, how long will the world last?

We'll answer both of these questions in sequence.

Solving the puzzle

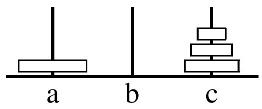
Recursion is the easiest way to explain how to solve this puzzle. Before going on and spoiling the fun, try yourself to think of a way to define the pattern for solving Tower of Hanoi.

Using recursion often involves a key insight that makes everything simpler. Often the insight is determining what data exactly we are recursing on — we ask, what is the essential feature of the problem that should change as we call ourselves? In the case of Is-A-Jew, the feature is the person in question: At the top level, we are asking about a person; a level deeper, we ask about the person's mother; in the next level, the grandmother; and so on.

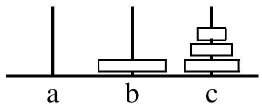
In our Tower of Hanoi solution, we recurse on the largest disk to be moved. That is, we will write a recursive function that takes as a parameter the disk that is the largest disk in the tower we want to move. Our function will also take three parameters indicating from which peg the tower should be moved ($source$), to which peg it should go ($dest$), and the other peg, which we can use temporarily to make this happen ($spare$).

At the top level, we will want to move the entire tower, so we want to move disks 3 and smaller from peg A to peg B. We can break this into three basic steps.

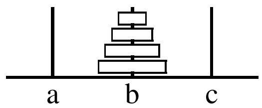
1. Move disks 2 and smaller from peg A ($source$) to peg C ($spare$), using peg B ($dest$) as a spare. How do we do this? By recursively using the same procedure. After finishing this, we'll have all the disks smaller than disk 3 on peg C. (Bear with me if this doesn't make sense for the moment - we'll do an example soon.)



2. Now, with all the smaller disks on the spare peg, we can move disk 3 directly from peg A ($source$) to peg B ($dest$).



3. Finally, we want to move disks 2 and smaller from peg C ($spare$) to peg B ($dest$). We do this recursively using the same procedure again. After we finish, we'll have all disks on $dest$.



*Invented by Edouard Lucas in 1883 to help market his commercial version.

In pseudocode, this looks like the following. At the top level, we'll call Move-Tower with $disk = 3$, $source = A$, $dest = B$, and $spare = C$.

Algorithm Move-Tower$(disk, source, dest, spare)$
1 **if** $disk = 0$, **then:**
2 Move $disk$ from $source$ to $dest$. *// base case*
3 **else:**
4 Move-Tower$(disk - 1, source, spare), dest)$ *// Step 1 above*
5 Move $disk$ from $source$ to $dest$. *// Step 2 above*
6 Move-Tower$(disk - 1, spare, dest, source)$ *// Step 3 above*
7 **end of if**

Note that the pseudocode adds a base case in line 1: When $disk$ is 0, the smallest disk, we don't need to worry about smaller disks, so we can just move the disk directly. In the other cases, we follow the three-step recursive procedure we already described for disk 3 (this is done in lines 4–6).

An example will help to explain what is going on here. First, a definition: The **call stack** is a representation of where we are in the recursion. As we progress through the algorithm, we will have several levels. Each level will have a different status (the variables $disk$, $source$, $dest$, and $spare$ are different at all levels, and we will be at different locations in the different functions. As we proceed, we will put new function calls at the top (end) of the stack, and we will remove function calls from the top (end) of the stack as we finish them.

We'll look a three-disk problem here. We use MT as an abbreviation for Move-Tower.

1. We begin with a call to $\mathsf{MT}(2, A, B, C)$, so that our call stack is simply $\langle(\mathsf{MT}(2, A, B, C), 0)\rangle$. (This representation of the call stack says that there is one function call currently on it. This is a function call to Move-Tower$(2, A, B, C)$, and we are currently at line 0 of the call.) Since $disk \neq 0$, this is not the base case, and we go to line 4.

2. At line 4 of $\mathsf{MT}(2, A, B, C)$, we call $\mathsf{MT}(1, A, C, B)$. So now our call stack becomes

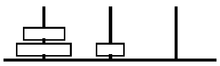
$$\langle(\mathsf{MT}(2, A, B, C), 4); (\mathsf{MT}(1, A, C, B), 0)\rangle .$$

Now at this new call to MT, we have $disk = 1$. We are still not in the base case, so we proceed to line 4.

3. At line 4 of $\mathsf{MT}(1, A, C, B)$, we call $\mathsf{MT}(0, A, B, C)$. So now our call stack becomes

$$\langle(\mathsf{MT}(2, A, B, C), 4); (\mathsf{MT}(1, A, C, B), 4); (\mathsf{MT}(0, A, B, C), 0)\rangle .$$

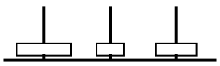
At this call to MT, we have $disk = 0$. We enter the base case (line 1) and move disk 0 from A to B.



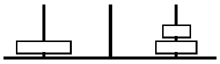
4. We now reach the end of the call to $\mathsf{MT}(0, A, B, C)$. We remove this from the call stack and step to the next line (line 5) of what is now on the top. Our call stack is now

$$\langle(\mathsf{MT}(2, A, B, C), 4); (\mathsf{MT}(1, A, C, B), 5)\rangle .$$

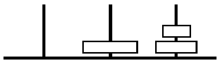
This says to move disk 1 from A to C.



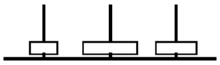
5. Now, at line 6 of $\mathsf{MT}(1, A, C, B)$, we call $\mathsf{MT}(0, B, C, A)$. Our call stack now becomes
$$\langle(\mathsf{MT}(2, A, B, C), 4); (\mathsf{MT}(1, A, C, B), 6); (\mathsf{MT}(0, B, C, A), 0)\rangle \, .$$
In this call, we are at the base case and so move disk 0 from B to C.



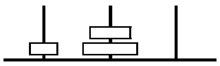
6. We return from this call, making the call stack become
$$\langle(\mathsf{MT}(2, A, B, C), 4); (\mathsf{MT}(1, A, C, B), 7)\rangle \, .$$
Proceeding from line 7 of $\mathsf{MT}(1, A, C, B)$, we find we reach the end and so return from it too. Now the call stack is simply $\langle(\mathsf{MT}(2, A, B, C), 5)\rangle$. Line 5 says to move disk 2 from A to B.



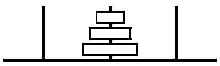
7. At line 6 of $\mathsf{MT}(2, A, B, C)$, we make another function call, now to $\mathsf{MT}(1, C, B, A)$. Our call stack becomes
$$\langle(\mathsf{MT}(2, A, B, C), 6); (\mathsf{MT}(1, C, B, A), 0)\rangle \, .$$
We do not enter the base case, and proceed to line 4 of $\mathsf{MT}(1, C, B, A)$, where we make another function call to make the call stack
$$\langle(\mathsf{MT}(2, A, B, C), 6); (\mathsf{MT}(1, C, B, A), 4); (\mathsf{MT}(0, C, A, B), 0)\rangle \, .$$
In the call $\mathsf{MT}(0, C, A, B)$ we enter the base case and move disk 0 directly from C to A.



8. We return from $\mathsf{MT}(0, C, A, B)$; now the call stack is
$$\langle(\mathsf{MT}(2, A, B, C), 6); (\mathsf{MT}(1, C, B, A), 5)\rangle \, .$$
At line 5 of $\mathsf{MT}(1, C, B, A)$, we move disk 2 to peg B.



9. At line 6, we call $\mathsf{MT}(0, A, B, C)$. The call stack is now
$$\langle(\mathsf{MT}(2, A, B, C), 6); (\mathsf{MT}(1, C, B, A), 6); (\mathsf{MT}(0, A, B, C), 0)\rangle \, ,$$
and from here we proceed to line 1 and move disk 0 from A to B.



10. We return from the call to $\mathsf{MT}(0, A, B, C)$; the call stack is now
$$\langle(\mathsf{MT}(2, A, B, C), 6); (\mathsf{MT}(1, C, B, A), 7)\rangle \, .$$
We return from the call to $\mathsf{MT}(1, C, B, A)$; the stack is now $\langle(\mathsf{MT}(2, A, B, C), 7)\rangle$. We return from the call to $\mathsf{MT}(2, A, B, C)$; the call stack is now empty, and so we are done.

Besides the call stack, another useful way to visualize what happens when you run Move-Tower is a **call tree**. The call tree graphically represents all the recursive calls made by a single function call. For example, Figure 10.1 contains a call tree for $\mathsf{Move\text{-}Tower}(3, A, B, C)$. Each function call in the call tree is called a **node**. The nodes connected just below any node n represent the function calls made by the function call for n Just below the top, for example, are $\mathsf{Move\text{-}Tower}(2, A, C, B)$ and $\mathsf{Move\text{-}Tower}(2, C, B, A)$, since these are the two function calls that $\mathsf{Move\text{-}Tower}(3, A, B, C)$ makes. At the bottom are many nodes without any nodes connected below them — these represent base cases.

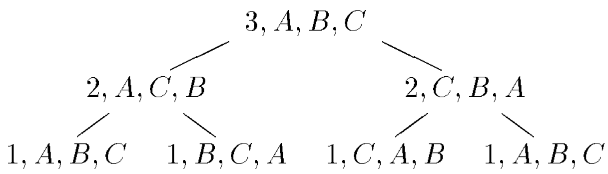


Figure 10.1: Call tree for Move-Tower$(3, A, B, C)$.

Analyzing our solution

Now we ask: If the monks use Move-Tower, how long will it be before the world ends? To answer this question, we need to learn about *recurrences*. A **recurrence** is a well-defined mathematical function written in terms of itself; it's a mathematical function defined recursively.

Take the **Fibonacci sequence** as an example. The Fibonacci sequence is the sequence of numbers

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

The first two numbers of the sequence are both 1, while each succeeding number is the sum of the two numbers before it. (We arrived at 55 as the tenth number, since it is the sum of 21 and 34, the eighth and ninth numbers.)

Let's define a function $F(n)$ that returns the $(n+1)$th Fibonacci number. (Don't let the use of $n+1$ rather than n confuse you; it's just a little more convenient if we number this sequence starting at 0.) First, we knock off two base cases:

$$F(0) = 1$$
$$F(1) = 1$$

Now we consider the other numbers. To get the $(n+1)$th Fibonacci, we just add the nth Fibonacci and the $(n-1)$th Fibonacci.

$$F(n) = F(n-1) + F(n-2) .$$

This function F is called a *recurrence* because it is defined in terms of itself evaluated at other values.

Now we're going to use a recurrence to find how many times the monks will move a disk if they follow our Move-Tower program. Think about this on your own for a while before proceeding.

To answer how long it will take our friendly monks to destroy the world, we write a recurrence (let's call it $M(n)$) for the number of moves Move-Tower takes for an n-disk tower.

The base case — when n is 1 — is easy: The monks just move the single disk directly. Thus we have

$$M(1) = 1 .$$

In the other cases, the monks follow our three-step procedure. First they move the $(n-1)$-disk tower to the spare peg; this takes $M(n-1)$ moves. Then the monks move the nth disk, taking 1 additional move. And finally they move the $(n-1)$-disk tower again (this time to the top of the nth disk), taking $M(n-1)$ moves. This gives us our recurrence relation,

$$M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1 .$$

Since the monks are handling a 64-disk tower, all we need to do is to compute $M(64)$, and that tells us how many moves they will have to make. This would be more convenient if we had $M(n)$ in **closed form** — that is, if we could write a formula for $M(n)$ without using recursion. Do you see what it should be? (It may be helpful if you go ahead and compute the first few values, like $M(2)$, $M(3)$, and $M(4)$.)

Looking at these first few numbers, we see the following.

$$\begin{array}{rcrcr} M(1) & = & & & 1 \\ M(2) & = & 2M(1)+1 & = & 3 \\ M(3) & = & 2M(2)+1 & = & 7 \\ M(4) & = & 2M(3)+1 & = & 15 \\ M(5) & = & 2M(4)+1 & = & 31 \end{array}$$

By looking at this, we can guess that

$$M(n) = 2^n - 1 \,.$$

We can prove this using a simple proof by induction. For $n = 1$, $M(n)$ is 1, which is indeed $2^n - 1$. Now consider any $n > 1$ and say $M(n-1) = 2^{n-1} - 1$. Then $M(n)$, which we have already seen is $2M(n-1) + 1$, is $2(2^{n-1} - 1) + 1 = 2^n - 1$. This completes a proof by induction that $M(n) = 2^n - 1$.

So the monks will move $2^{64} - 1 \approx 18.45 \times 10^{18}$ disks. Even if they could move a disk every millisecond, they'd have to work for 584.6 million years. It looks like we're safe.†

Exercise 10.1: (Solution, 121) Write a recursive program to enumerate all subsets of $\{1, \ldots, n\}$ for some n the user specifies. (The order in which they are printed is not important.)

```
Choose from how many? 2

 2
 1
 1 2
```

Exercise 10.2: Write a recursive program to enumerate all the subsets of a given size from $\{1, \ldots, n\}$ for some n the user specifies. (This is like the program on page 36, except now we actually list the choices.)

```
Choose how many of how many? 2 4
 1 2
 1 3
 1 4
 2 3
 2 4
 3 4
```

One tempting way to do this exercise is to take the answer of Exercise 10.1 and modify it to only print out subsets of the given size. Don't do this; it is impractically slow for large n. (Your program should handle all pairs from 40 numbers quite quickly.)

† Actually, you might object that the monks could use a much faster algorithm. But it turns out that Move-Tower uses the fewest moves possible.

Chapter 11

Playing games

Although artificial intelligence research dates from the dawn of computer science, its goals are so ambitious that it still has far to go. But it has a few successes behind it. One of the most notable examples is in playing games.

The motivation behind game-playing research is much more serious than it sounds. The primary goal is to have computers adapt and plan, so that they can handle serious tasks like driving a car or managing a production line. Game-playing as a topic of study came about because it was fun, manageable, but somewhat beyond current technology. For similar reasons, some robotics researchers today concentrate on creating robotic juggling — not because juggling is a useful task, but because it requires dexterity and quick thinking that robots need but currently lack.

Classical game-playing techniques work for a variety of games with certain common characteristics. We assume that the game involves two players alternating turns. We assume that both players always know everything about the current state of the game. (This is not true for many card games, for example, because a player does not know the other's hand.) And we assume that the number of moves on each turn is limited.

These restrictions still encompass many games, including tic-tac-toe, Connect-4, Othello, checkers, chess, and go. In this chapter we look at the simplest of these, tic-tac-toe. But, except for go, the techniques covered in this chapter work well for all of the games just listed.

In case your childhood somehow lacked tic-tac-toe, let us review the rules. We start with a 3×3 board, all blank. It is X's turn first, and X can place his mark in any of the nine blanks. Then O places her mark in one of the eight remaining blanks. In response X has seven choices. In this way the players alternate turns until one of the players has three marks along a horizontal, vertical, or diagonal line (thus winning the game), or until the board becomes filled (this is a tie if neither player has won).

One approach to writing a tic-tac-toe program is to simply enumerate the situations that may occur and what the computer should do in each case. For example: If the computer is O, and X's first move is in a corner, then O should play in the center. If X's first move is in the center, O should play in a corner. And so on. But this approach has a major problem: It relies on a human to list what to do in every circumstance in advance. The list becomes unmanageable for games more complicated than tic-tac-toe; worse, the computer will never exceed the ability of its programmer.

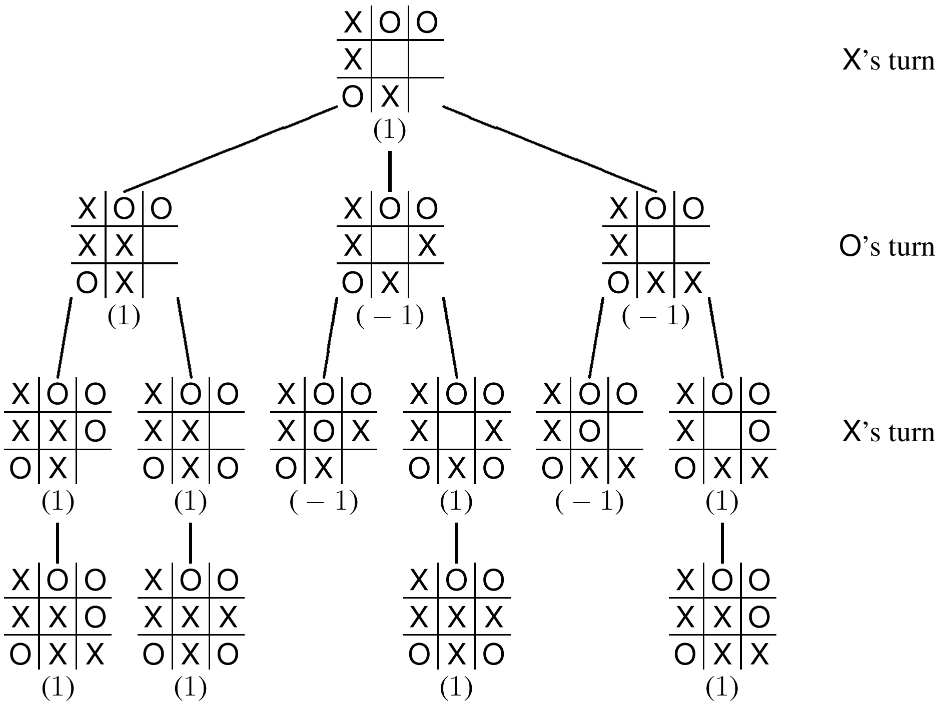


Figure 11.1: Evaluating a board.

11.1 Game tree search

A more general approach is have the computer determine how to move by evaluating all choices. Say the current board is

X	O	O
X		
O	X	

and the computer, playing X, must choose a move. To do this, the computer can consider each of the three possible next boards and consider which is most appealing to O.

X	O	O
X	X	
O	X	

X	O	O
X		X
O	X	

X	O	O
X		
O	X	X

To determine which is best for O, the computer looks at each of O's possibilities. Eventually we end up with what is called a **game tree**, as in Figure 11.1.

The parenthesized numbers in Figure 11.1 indicate the "value" of each board: 0 for a tie, 1 for a guaranteed win for X, and -1 for a guaranteed win for O. At the bottom, when a final board is reached, the value of the board is the outcome for that board: In the figure, the bottom left board is 1 because X has completed the diagonal. For other boards, the value is the best of the choices for the current player. For the top board, we have three choices: a win for X, a win for O, or a win for O. It is X's turn, so X would choose the win for X; hence the board's value is 1, and X should move in the board's center.

Evaluating such a tree is called the **minimax search** algorithm, since X chooses the maximum of its childrens' values and O chooses the minimum. We can write the minimax search algorithm very naturally using recursion.

Algorithm Minimax-Search$(board, player)$
// base case for final state
if $board$ is a win for X **then return** 1.
else if $board$ is a tie **then return** 0.
else if $board$ is a win for O **then return** -1.
end of if
// try all moves, letting $best$ *be value of most desirable*
if $player =$ X **then** let $best$ hold $-\infty$.
else let $best$ hold ∞.
end of if
for each legal $move$ on $board$ **do**
    Make $move$ on $board$.
    Let $value$ hold Minimax-Search$(board,$ opposite $player)$.
    Undo $move$ from $board$.
    if $player =$ X **and** $value > best$ **then** let $best$ hold $value$.
    else if $player =$ O **and** $value < best$ **then** let $best$ hold $value$.
    end of if
end of loop
return $best$.

11.2 Heuristics

The problem with minimax search is that it takes a lot of time. Tic-tac-toe games, which last at most 9 moves, have game trees that computers can exhaust. But a chess game may last more than 50 moves; the game tree is well beyond the total computing capacity of the world.

The solution is simple. We search only to a certain depth of the tree. When we see a board at the depth that is not in a final state, we apply a **heuristic function** to estimate the board's value. The heuristic function is a function written by the programmer that tells roughly how good the board is. In tic-tac-toe, a simple heuristic function may calculate the difference of the number of possible wins for X and the number of possible wins for O, where a possible win is a row, column, or diagonal with none of the opponent's pieces. The board

O	X	
X	O	
X	O	X

has one possible win for X (the right column) and no possible wins for O; its heuristic value would be 1. We should also make the value of guaranteed wins more extreme (10^6 and -10^6, say) to indicate how sure we are of them.

We now evaluate the board by going to a certain depth and using the heuristic function to evaluate the boards at the bottom depth that are not final. We use the same minimax procedure for boards above the maximum depth. Figure 11.2 illustrates an example going to a depth of 2. In this example, X would decide for either the second or third choices.

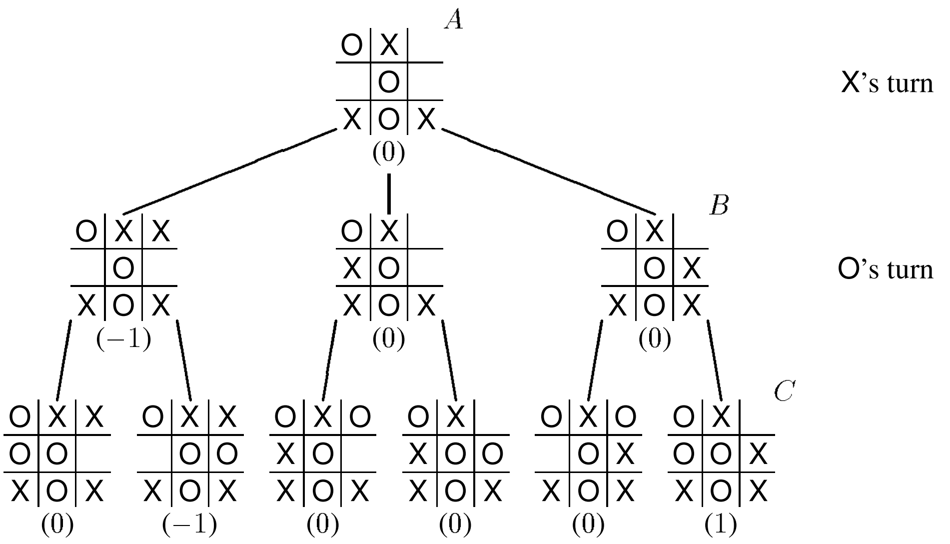


Figure 11.2: Using heuristics to evaluate a board.

11.3 Alpha-beta search

Heuristics allow us to write reasonably good game-playing programs. The professionals are somewhat more sophisticated, though, in choosing which boards to evaluate. One particularly interesting enhancement is called **alpha-beta search**, where we notice that some of the boards need not be evaluated to get the exact answer.

Figure 11.2 provides an example where this applies. Call the right-most board in the bottom level C, its parent B, and the top of the tree A. Notice that, no matter what the value of C is, the value of B will be at most 0, since O will choose the minimum of its children's values and B already knows that the first choice gives 0. Since at A X already knows it can guarantee 0 by choosing the middle route, the exact value of B does not matter. Through this reasoning, then, we can avoid evaluating C.

In this case we would avoid evaluating a single board — not so impressive. But the reasoning can help tremendously for larger games, almost doubling the depth that can be handled within the time limit.

The pseudocode for alpha-beta search is in Figure 11.3. It is not much longer, but it is much harder to interpret. In the code, α (the Greek letter *alpha*) represents the best (i.e., maximum) value we have found for X for any of the boards at or above the current one where it is X's turn. (At the top, α is initially $-\infty$.) The parameter β (the Greek letter *beta*) represents the best (i.e., minimum) value we have found for O for any of the boards at or above the current one where it is O's turn; at the top it is ∞. (These variables are the inspiration for the decidedly lame name computer scientists have given to this technique.) We can stop examining a board when it has $\alpha \geq \beta$.

Summary

The approach of Alpha-Beta-Search is very close to what the best game programs use. They have some additional enhancements. For example, a good chess program will have a large list describing specific moves and responses for the beginning of the game. It may also vary the

Algorithm Alpha-Beta-Search$(board, player, \alpha, \beta, depth)$
// base case for final state
if $board$ is a win for X **then return** 10^6.
else if $board$ is a tie **then return** 0.
else if $board$ is a win for O **then return** -10^6.
end of if
// if we've max'ed out the game tree, return the heuristic value
if $depth = 0$, **then:**
    return Heuristic$(board)$.
end of if
// try all moves, letting $best$ be value of most desirable
for each legal $move$ on $board$ **do**
    Make $move$ on $board$.
    Let $value$ hold Alpha-Beta-Search$(board$, opposite $player, \alpha, \beta, depth - 1)$.
    Undo $move$ from $board$.
    if $player =$ X **and** $value > \alpha$, **then:**
        Let α hold $value$.
        if $\alpha \geq \beta$, **then:**
            return β.
        end of if
    end of if
    if $player =$ X **and** $value < \beta$, **then:**
        Let β hold $value$.
        if $\alpha \geq \beta$, **then:**
            return α.
        end of if
    end of if
end of loop
if $player =$ X **then return** α.
else return β.
end of if

Figure 11.3: The Alpha-Beta-Search algorithm.

search depth based on how good the board looks, rather than going to a fixed depth. But the primary code is very much like what is above, with a sophisticated heuristic function attached.

Philosophically, these techniques are not very satisfying. Can one really say that a computer using exhaustive search is displaying any intelligence? Certainly if the standard is how a human works, no. While major chess computers search through millions of boards for each play, a human grandmaster searches through merely hundreds. One cannot accurately say that a computer is actually reasoning as a human does.

This is really a question for philosophers about the nature of intelligence. For computer scientists, the interesting question raised is how to apply the human's techniques effectively. Through trying to apply them, we can learn more about the the human's techniques. So far attempts to use more human reasoning have had only limited success, however.

More pragmatically, these game-playing techniques do not generalize to other planning tasks, where actions sometimes fail to produce the desired result (steering on ice, as an extreme example) and the world is much larger than a few pieces on a board. These problems are much harder. Researchers are currently addressing them, but a long time will pass before we know how to automatically handle such real-world problems. Game-playing is just a first step.

Exercise 11.1: Prove that X can still guarantee a tie game from the following board.

X	O	

Proving this involves drawing the game tree starting at this board, except that for levels where it is X's turn, you need only include your chosen move for X. (Thus at every other level, each node will have only one child at the next level.)

FOURTH UNIT

Internet

Computer science is hard to discuss without some knowledge of programming, but computer science is not *about* programming — no more than mathematics is about arithmetic, or biology is about identifying animals, or history is about knowing the dates of events. Computer science is about *problem-solving*, and programming is a means to understanding the capacity of computers as problem-solving devices. Programming is a tool, and now that we understand its basics we are ready to look at hard-core computer science.

This unit concentrates on one particular aspect of computer science: the Internet. In the last few decades, the Internet has grown quickly from a minor plaything for computer science researchers to become a major player in the world's economies and lifestyles (see Figure 11.4). The Internet is the product of decades of research and still presents many interesting problems for computer scientists to consider, some of which we see in this unit.

There are a variety of tacks we could take on the Internet. We could, for example, study how to use its components — mail, Web pages, newsgroups, and its other offerings. On the other end of the spectrum, we could study how networks are built and how to create our own. This book takes neither approach. We opt for the middle road, the most interesting from a computer scientists' perspective: We examine how the Internet fits together and transmits messages.

Five chapters divide our approach into more manageable chunks.

Chapter 12 We are introduced to the fundamentals of networks.

Chapter 13 We learn about how Internet messages try to reach their destination.

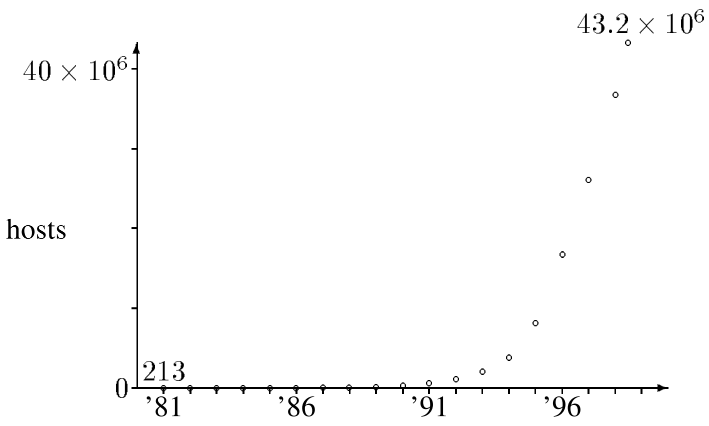


Figure 11.4: Number of Internet host computers.

Chapter 14 Since messages sometimes fail to reach their destination, we study a protocol for resending messages until they get there.

Chapter 15 We examine a sampling of useful protocols that use the techniques from the previous chapters for specific tasks like sending mail and fetching Web pages.

Chapter 16 We examine some of the Internet's weaknesses and learn the fundamentals of cryptography, a solution to many of the weaknesses.

One purpose of our study is to understand how the Internet works, but this is only an incidental benefit. More crucially, we want to understand the questions and techniques that computer science has developed for the problems that the Internet presents. What are these problems? The Internet must be able to scale well — that is, it must be able to grow as quickly as the Internet itself. It must be able to quickly adapt to environmental changes; for example, if a single computer or region's power goes out, other computers must be able to work around the absence gracefully. And, despite the complexities, the Internet must ultimately be manageable by human programmers. These problems will crop up, sometimes obviously but more often subtly; look out for them.

Chapter 12

Networking fundamentals

In this chapter we look at the fundamentals of networks, dividing our approach between the representation of data and the division of labor in networking software.

12.1 Representing data

The original motivation for computers was to manipulate numbers. Most of today's applications appear far from this; e-mail, Web browsers, and computer games are not obviously related to numbers. Despite their appearance, however, in a real sense even these applications reduce their various problems to specific computational problems. Thus a crucial question is, how can we represent numbers to a computer?

Binary numbers

On an even more fundamental level, computers manipulate electricity, routing it between wires. Through this routing, the computer represents numbers. A wire represents 1 when electricity flows through it and 0 when electricity is not. The computer constantly decides whether to route electricity through the wire, depending on whether each wire should represent a 1 or a 0.*

As beings who typically have ten fingers, humans work with the *decimal* (base-10) *numbering system*: It's no coincidence that we call both our fingers and the characters `0` through `9` *digits*. In the decimal numbering system, the sequence of digits *1980* represents the number

$$1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 0 \times 10^0 .$$

Since a wire can represent only the two digits 0 and 1, computers work with a base-2 system: the **binary numbering system**. The basic unit of data is a single binary digit, zero or one. We call this unit a **bit**, from **B**inary dig**IT**. To represent larger numbers, we can expand this just as we represent larger numbers in decimal notation. Consider the binary number $11010_{(2)}$. (The parenthesized subscript here is to emphasize that this is a base-2 number.) Each position in the number now represents a power of two. To convert $11010_{(2)}$ to the decimal representation with which we are familiar, we rewrite the number as

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 16_{(10)} + 8_{(10)} + 0_{(10)} + 2_{(10)} + 0_{(10)} = 26_{(10)} .$$

*Why two values? We have to have at least two values; if there is only one value, then the computer cannot think or communicate anything other than this value, and this is not useful. People have tried giving computers more values than just two, but this complicates things enough to hurt overall efficiency.

letter	code	letter	code	letter	code
‘ ’	$00100000_{(2)} = 32$	‘A’	$01000001_{(2)} = 65$	‘a’	$01100001_{(2)} = 97$
‘.’	$00101110_{(2)} = 46$	‘B’	$01000010_{(2)} = 66$	‘b’	$01100010_{(2)} = 98$
‘0’	$00110000_{(2)} = 48$	$\vdots$		$\vdots$	
‘1’	$00110001_{(2)} = 49$	‘I’	$01001001_{(2)} = 73$	‘m’	$01101101_{(2)} = 109$
$\vdots$		$\vdots$		$\vdots$	
‘9’	$00111001_{(2)} = 57$	‘Z’	$01011010_{(2)} = 90$	‘z’	$01111010_{(2)} = 122$

Table 12.1: An ASCII sampler.

So $11010_{(2)}$ is decimal 26.

Conversely, to represent the number $100_{(10)}$ in binary, we would break it into a sum of distinct powers of two:

$$100_{(10)} = 64_{(10)} + 32_{(10)} + 4_{(10)} = 2^6 + 2^5 + 2^2 .$$

Hence the binary representation of $100_{(10)}$ is $1100100_{(2)}$.

Types of data

Because the number of bits in a computer is so large and we rarely want to work with numbers between 0 and 1, we find it convenient to break data into groups of eight bits, each called a **byte**. A single byte can represent the numbers between $00000000_{(2)} = 0_{(10)}$ and $11111111_{(2)} = 255_{(10)}$. On top of bytes we build the three basic data types: characters, integers, and floating-point numbers. There are other types of data — pictures and sound, for example — but they tend to be simple conglomerations of these three basic types.

Characters

A **character** is a single letter, digit, punctuation mark, or control character (like a tab or end-of-line). Most of today's computers represent a character with a single byte using an encoding standard called **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). Table 12.1 contains a few of these codes. If we want to interpret a sequence of bits

$$01001001\ 00100000\ 01100001\ 01101101\ 00101110$$

as characters, we divide it into bytes and interpret using ASCII. Here, the message is, I think, "I am."

Integers

The next basic type is the **integer**. These are the numbers without fractional pieces, like 26 or -100 but not 1.62. A single byte, we saw, can represent numbers between 0 and 255. Since we frequently want to use integers outside this range, computers group bytes into *words* to represent numbers. Word sizes vary between computers: Some old computers (the 16-bit machines) use 2-byte words, most current computers (32-bit machines) use 4-byte words, and some (the 64-bit machines) even use 8-byte words.

With 32 bits we can represent any integer between -2^{31} and $2^{31}-1$. Representing positive integers is straightforward: We take the binary representation and place zeroes to the right to fill out the bits. The number $100_{(10)}$ becomes

$$00000000\;00000000\;00000000\;01100100\,.$$

The most popular method for representing negative numbers is the **2's-complement representation**. On a 32-bit machine using 2's-complement representation, the representation of $-x$ is binary representation of the difference of 2^{32} and x. Thus the number $-100_{(10)}$ becomes

$$\begin{array}{rr} & 1\;00000000\;00000000\;00000000\;00000000 \\ - & 00000000\;00000000\;00000000\;01100100 \\ \hline & 11111111\;11111111\;11111111\;10011100 \end{array}$$

Floating-point numbers

The final basic data type of a computer is a *floating-point number*. A floating-point number allows for the representation of a fractional number (like 3.14). These too can be represented in binary. The most common encoding method for floating-point numbers is the **IEEE standard**. It calls for representing the number in base-2 scientific notation. In binary, $3.14_{(10)}$ is

$$11.001000111101011100001\ldots_{(2)} = 1.1001000111101011100001\ldots_{(2)} \times 2^{1}\,.$$

The IEEE standard uses the first bit of a word to represent the sign: 0 for positive, 1 for negative. The next eight bits hold the exponent in the scientific representation plus 128 (adding 128 allows negative exponents). And the final 23 bits give the first 23 bits of the mantissa's fractional part. (Since the number before the decimal is always 1, there is no reason to include it.) So the number 3.14 is represented in 32 bits as

$$0\;10000001\;10010001111010111000010$$

All of these three basic types, then, have representations in bits. Several layers of abstraction separate the raw electronics of the computer and the data types that programmers actually think about. From electricity we built bits, bits begat bytes, and bytes became characters, integers, and floating-point numbers. These are the basic building blocks of data.

12.2 Division of labor

Since networks are extremely complicated objects, researchers find it useful to work with abstractions. They use **layers** to divide the duties of networking into four pieces. We thus tend to envision a single message as something of a layer cake (see Figure 12.1).

application layer The **application layer** interprets messages using some special-purpose **protocol**. For example, HTTP (the World Wide Web protocol) specifies how to interpret Web page requests and how Web browsers should interpret the servers' responses. We will look at the application layer more in Chapter 15.

transport layer The **transport layer** takes a single message from the application layer and divides it into **packets** of about 1000 bytes each. Instead of sending the entire message across the network in one big chunk, the packets are sent individually instead, since many

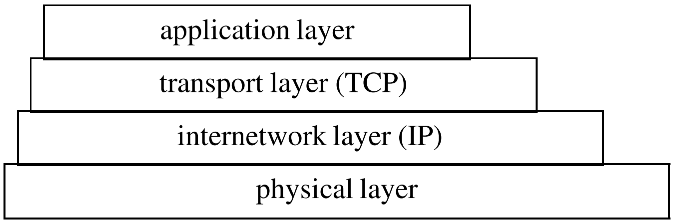


Figure 12.1: The layers of an Internet message.

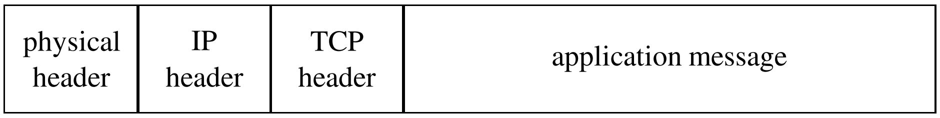


Figure 12.2: Packet headers.

small messages are much easier to handle than big pieces. But the transport layer gives these packets to the internetwork layer, who ships each packet to the destination. The destination's internetwork layer receives these and passes them up to the destination's transport layer, who reconstructs the original message by packing all the packets back together again.

The most widely-used protocol for the transport layer is the **Transport Control Protocol** (**TCP**), which we will study in Chapter 14.

internetwork layer The **internetwork layer** takes a single packet and attempts to route this packet to its destination. The internetwork layer does not guarantee that these packets will arrive at their destination in any particular order or even that they will arrive at all. What the internetwork layer *does* guarantee is *best-effort delivery* — that it will make a reasonable effort to get the packet to its destination if possible. This is inconvenient, since generally all packets need to reach their destination. The jobs of resending lost packets and of ordering the packets correctly go up to the transport layer.

On the Internet, the internetwork layer is implemented using the **Internet Protocol** (**IP**). We will discuss this in Chapter 13.

physical layer The Internet is a networked combination of networks. Any Internet packet must pass through several networks; the **physical layer**'s job is to pass the packet through a single network. We will not examine the physical layer in this book; we simply assume that we already have a facility for transporting packets within a single network.

Each layer adds a **header** to a message giving information about how to handle the message, as in Figure 12.2. For example, the internetwork layer header contains several bytes telling the address of where the packet is headed. Among other things, the transport layer adds bytes identifying where the packet is within the overall message (so that the destination knows where the packet goes within the final message).

Now we begin our journey up the layers. We assume a physical layer to transport messages within a single network. How does the internetwork layer get a packet to its destination?

Chapter 13

Transporting packets

In this chapter we take a look at how the Internet gets a packet from one place to another using IP (**I**nternet **P**rotocol).

The Internet is a network of networks. We assume that each network knows how to deliver a message within itself (the job of routing a packet within an individual network is handled by the physical layer). The Internet's job is to determine how to find its way through the networks to deliver a message, say from San Francisco to Pittsburgh.

There are two basic steps to routing a packet from its source to its destination. First we must determine where the packet is going; then we must send the packet through the Internet to that place.

13.1 Machine names

The first step toward transmitting a message is to determine where it is going. To do this, we use the machine's address. A computer on the Internet typically has two identities. The first is a **mnemonic name**, like

`truffle.bh.andrew.cmu.edu`

Notice that the mnemonic name consists of several parts separated by periods. The first part (`truffle`) is the name of the computer itself. The succeeding parts are called **domains**; they are an indication of where the computer is. So `truffle` is part of the `bh` domain, which is part of the `andrew` domain, which is part of the `cmu` domain, which is part of the `edu` domain. This layering of domains is called the **domain hierarchy**. Figure 13.1 illustrates some of the domain hierarchy.

The second identity of a computer is its 4-byte **IP address**. For example, `truffle`'s IP address is

`128.2.124.147`

This is what computers actually use to identify other computers on the network. The numbers of an IP address are also hierarchical. In this case, the first two numbers of the address, `128.2`, indicate that the computer is in CMU's network.

Although we can use IP addresses to tell where to send a message, the mnemonic names are much easier for humans, so that is what we humans usually use. But to send the message, the computer must have the IP address. So the first step to sending a message is to translate the mnemonic name into the corresponding IP address. This is called **name resolution**.

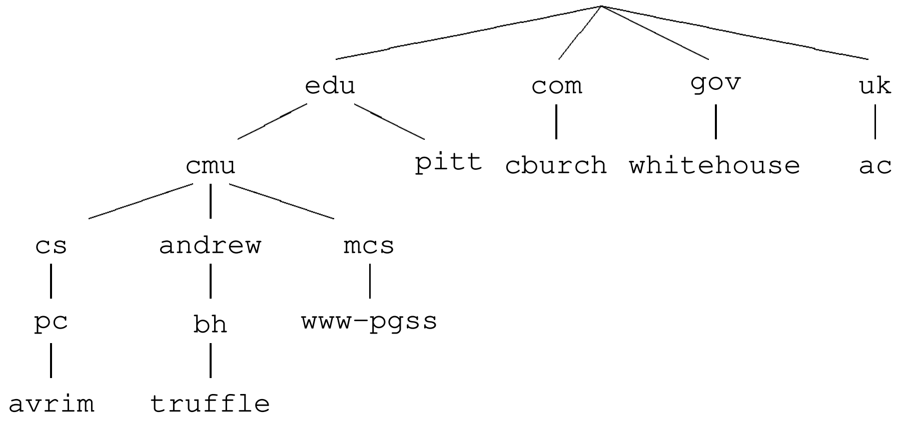


Figure 13.1: Internet domain hierarchy.

The simple solution to name resolution is to store all the translations on every computer. This is an impractical solution: There are too many computers and the Internet changes too rapidly. Every Internet computer would spend a vast amount of resources just try to remember everybody's names! Instead, when a computer sees a new mnemonic name, it goes out to the network to find the corresponding IP address.

To do this, the computer goes down the domains. Each domain has a **domain name server** whose job is to give IP addresses for the domains within it. So, if we want to find `truffle`, we begin at the top-level domain name server to find the `edu` name server. Then we ask the `edu` name server for the `cmu` domain. We ask the `cmu` name server for the `andrew` domain, whose name server we ask for the `bh` domain, whose name server we ask for `truffle`. This name server answers with the IP address of `truffle`, answering our question.

This approach solves the old problems of size and rapid change, but now a new problem arises: A name server (like the `.com` server) cannot conceivably handle the traffic of answering a request every time any Internet computer sends a message to another computer.

So actually a computer stores (**caches**) the important IP addresses it sees. This saves time and communication. So if we have already accessed `truffle`, there is no need to go through the process at all. If we have not, but we have accessed some computer in `bh`, we may remember the `bh` name server's address and ask it directly who `truffle` is. If we don't have this information, but we have accessed the `andrew` domain before, then we can skip to asking the `andrew` name server for the identity of `bh`.

By using domain name servers, the network distributes the job of maintaining name translations to just a small fraction of the computers of the Internet. In practice, a computer using good caching techniques usually spends very little time in translating a name.

13.2 Finding a route

Once we know the IP address of our message's destination, we still must route the message through the Internet. This is not easy: Somehow the messages are to go through the network and end up at a destination, without the benefit of any single map to consult.

But what does it mean to send a message on its way? Remember that we are not going to worry about individual networks; we assume they can work individually. But somehow we have to send messages between networks. To transfer a packet between networks, the Internet has **gateways**. A gateway is a computer that resides on two (or more) networks. This allows

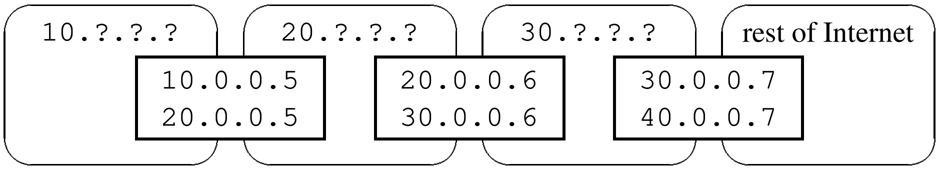


Figure 13.2: Internet gateways. (Boxes are gateways; ovals are networks.)

it to transfer messages between the networks. Figure 13.2 diagrams a series of networks (the light-bordered ovals), connected by gateways (the heavy-bordered rectangles). Notice that a gateway has multiple IP addresses, one for each network it is on. This is so the gateway can be recognized as being part of each of its networks.

To send a message on its way, then, means to send it to the best gateway in the network. To determine this, the computer consults a **routing table**, which tells where to send packets of different destinations. For example, for the gateway between the `20` and `30` networks of Figure 13.2, the routing table might read

if destination is	then route to
`10.?.?.?`	`20.0.0.5`
`20.?.?.?`	destination directly
`30.?.?.?`	destination directly
else	`30.0.0.7`

You can read this table as follows. If the packet's destination is in the `10` network, then the way to get there is through the gateway between the `10` and `20` networks, whose address on the `20` network is `20.0.0.5`. It can route the packet directly to this computer since the computer is on the `20` network. Since the computer is on both the `20` and `30` networks, it can route packets directly to machines in these networks. And since other packets should go through network `40`, the computer routes these packets to the gateway to that network, via the `30.0.0.7` gateway.

Naturally, these routing tables change occasionally. Periodically gateways tell their neighbors about the best routes they know. When a gateway receives this information, it considers whether to update its routing table. If it does, it also forwards the updated routes to its neighbors.

These gateways provide what is called **best-effort delivery**. That is, they try to route packets, but they may ignore (*drop*) packets if routing it is inconvenient. Possible reasons for dropping a packet include: The gateway is too busy with other things, the gateway doesn't know where the packet should go next, the packet has passed through too many computers (and so it may be that it is going in circles), or just the whimsy of the gateway dictates that it should be dropped.

Packet drops are frequent; it is not uncommon for about half of the packets to fail to reach their destination. In this case, the sender and receiver should detect there is a problem, negotiate what to do about it, and send the lost packets again until all packets successfully reach the destination. This task is performed by upper layers, not IP. We consider this issue in the next chapter, where we study TCP.

Chapter 14

Putting packets together

The IP protocol gives us the ability to route packets from their source to their destination with some degree of reliability (if only minimal). Of course programs often want to work with a much stronger system. Therefore computers provide an additional layer separating programs from IP. This layer is called **Transport Control Protocol** (TCP), and it provides reliable delivery of arbitrary amounts of information.

A program using TCP (as built into the computer's system) does not have to worry about the vagaries of lost packets and out-of-order transmission. Instead, the program can treat communication as being as simple as a telephone call. We'll see examples of programs using TCP in the next chapter.

In this chapter we see how TCP achieves its goal. We first see how TCP provides the illusion of a connection between programs. Then we see how TCP provides reliable delivery of information.

14.1 Connections

IP provides a system for computer-to-computer delivery, but programs want a telephone-like connection, and for that they need program-to-program delivery.

Abstract model

To provide this, TCP provides **ports**. A port is not a physical device; it is just a number between 0 and $65,535$. (Why $65,535$? It is $2^{16} - 1$, the largest number that can fit into two bytes.) Each program using TCP reserves a port on its computer for its own use, and each TCP packet header indicates for which TCP port it is intended. When the computer receives a TCP message, it reads the port number from the header, and it routes the body of the message to the program reserving that port.

A **connection** is a pair of Internet addresses A_0 and A_1 and corresponding ports p_0 and p_1. When program 0 wants to send a message to program 1, it sends a message to A_1 including in the header all the information about the connection (A_0, A_1, p_0, and p_1). When A_1 receives this message, it can tell from the header that the message is for program 1, since program 1 is the program reserving port p_1. So the computer gives the message to program 1, and program 1 can tell which connection the message belongs to, since all the information (A_0, A_1, p_0, and p_1) is included in the message.

port	protocol
17	QUOTE (quote of the day)
21	FTP (file transfer)
23	TELNET (remote login to computers)
25	SMTP (e-mail transfer)
37	TIME (time)
42	NAMESERVER (host name server)
53	DOMAIN (domain name server)
80	HTTP (Web page transfer)

Table 14.1: Some well-known port numbers.

Before sending any messages, the connection must be *established*. This is a matter of a simple initial protocol to make sure both machines know what to expect from the other's TCP information.

What really happens

The above description is a bit abstract. Luckily, things are a little easier to follow in practice.

Normally, one program — called a **server** — runs on a computer on a publicized port number. For widely-distributed applications, there are **well-known port numbers** reserved for them. For example, port 80 is reserved for Web servers. Table 14.1 lists several other well-known port numbers.

A program — called a **client** — (maybe on a different computer) reserves a port for it to communicate, and it sends a message asking to be connected with a given port on the server's computer. It includes its own port number in its message to the server, so that the server knows how to send messages back to the client. For example, when you tell your Web browser (which is a client) that you would like a page from `www.whitehouse.gov`, it sends a message to that computer saying that it wants to get in touch with whatever program is running on port 80.

Programs can and often do converse with many programs simultaneously through the same port. (Busy Web servers do this, for example.) This is fine, because each TCP message also includes information about the computer and port from which the message originated, and the program can use this information to distinguish conversations.

Technically, TCP doesn't impose such a client-server relationship between programs. But this is what happens in practice, as we'll see in the next chapter.

14.2 Reliable delivery

TCP's approach to reliable delivery is obvious, but it becomes more complex as we worry about efficiency issues.

Simple acknowledgement protocol

The obvious protocol for ensuring that a packet reaches its destination is to have the destination send an acknowledgement whenever it receives a packet. If the sender does not receive an acknowledgement within a reasonable time after the packet is sent, then the sender concludes

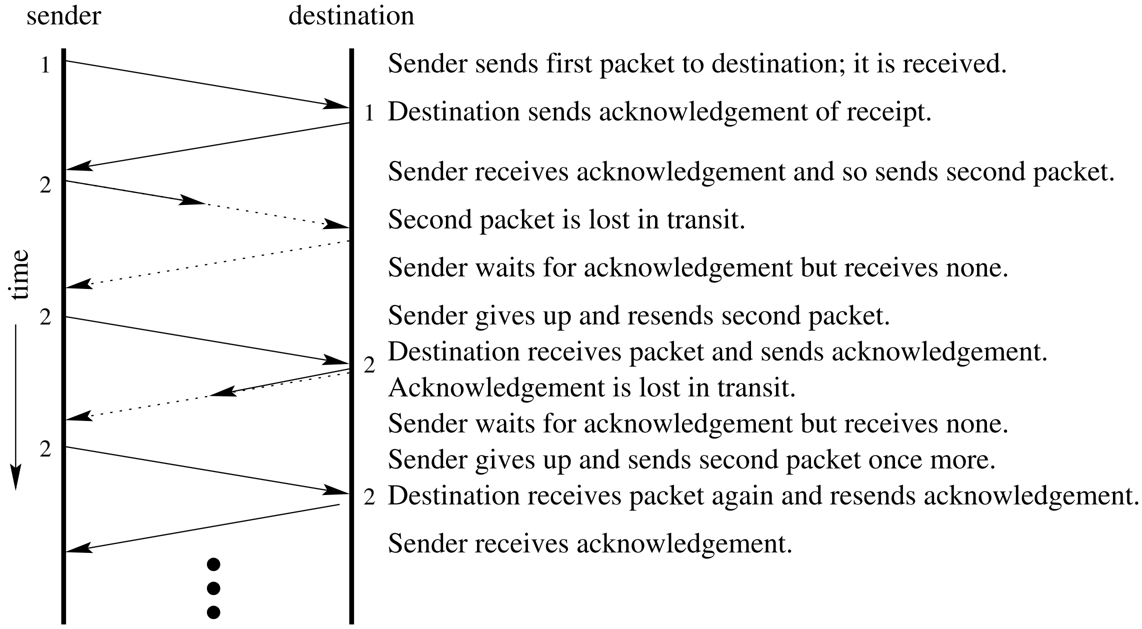


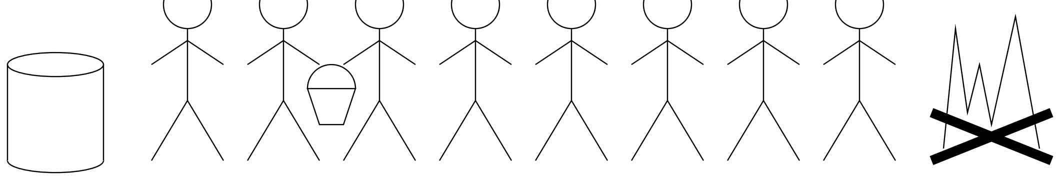
Figure 14.1: Simple acknowledgement protocol.

that the packet may have been lost, and so it resends the information. It continues sending packets until it receives an acknowledgement of receipt. Figure 14.1 diagrams this process.

This protocol necessitates numbering the packets as they appear in the message, since the destination may receive the same packet twice (as in Figure 14.1 when the destination's acknowledgement was lost in transit). Thus the TCP header for a packet includes, besides the port numbers of the source and destination, a **sequence number** telling with which byte the packet begins, relative to the first byte in the connection sent by the sender. It also includes an **acknowledgement number**, which indicates how many bytes the sender has received from the destination since the connection began.

Sliding window protocol

This system is pretty slow, however. The lag time for a packet to reach its destination can be large — we don't really want to wait that long for every single packet. It's like having a bucket brigade with only one bucket.



A bucket brigade is much more efficient with several buckets; likewise, a TCP connection is more efficient when there are several packets on the network at once.

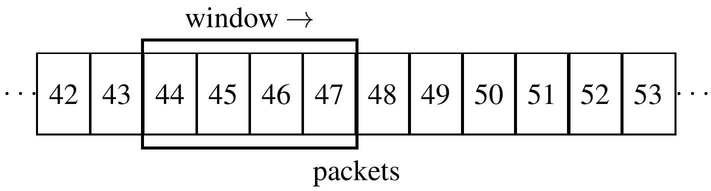


Figure 14.2: The TCP sliding window.

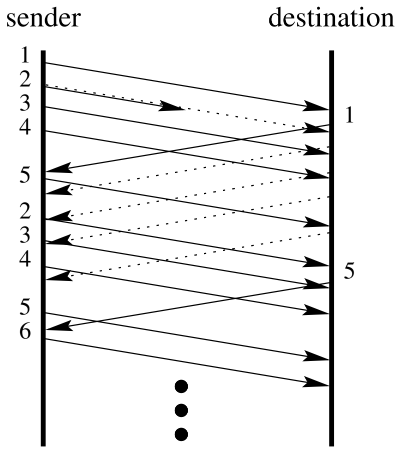
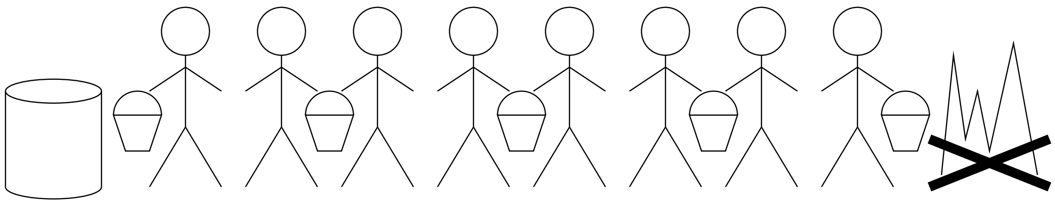


Figure 14.3: Example of communicating with sliding window of size 4.



On the extreme end, the sender might send all the packets in the message simultaneously. This would be a waste, however, since neither the intermediate points in the network nor the receiver can handle such large quantities, and almost all the packets would be lost. Instead, therefore, TCP adopts a compromise, called **sliding windows**.

The sliding window technique maintains a window of several packets that TCP is currently trying to send. (See Figure 14.2.) TCP keeps all the packets in its window on the network. When it receives an acknowledgement of receipt from the destination network, TCP moves the window up so that the first unreceived packet is on the far left of the window, and it sends all the packets that enter the window. When a packet in the window times out (that is, enough time has elapsed that the server gives up on receiving an acknowledgement for that packet), the sender resends the packet.

To illustrate how this works, let's step through the event sequence diagrammed in Figure 14.3. Here we have a sliding window of 4, so the window initially contains packets 1–4. It sends each of these to the destination. Packet 1 reaches the destination, and the destination sends back an acknowledgement of receipt. But packet 2 is lost midstream. In TCP, the destination only sends back an acknowledgement when the window can move forward, so even though the destination receives packets 3 and 4, it does not acknowledge them.

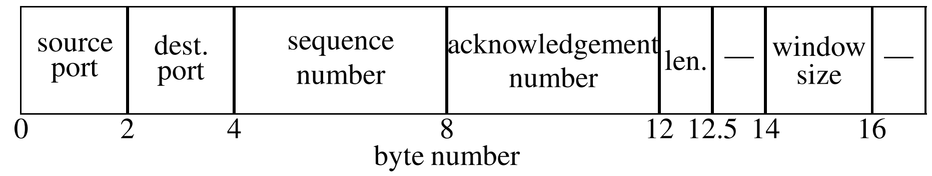


Figure 14.4: TCP header information.

Once the sender receives the acknowledgement for packet 1, the window moves forward, and the sender sends the new packet in the window, packet 5. Eventually it decides that packet 2 must have been lost, and so it resends packet 2. It does the same for packets 3 and 4 when their acknowledgement is long overdue.[*] The destination receives packet 5, but it cannot yet acknowledge it, because it still doesn't have packet 2. But when the destination receives packet 2, it has packets 3, 4, and 5 from their first transmissions. So it sends back an acknowledgement telling the sender that it has received everything through packet 5. Now the sender moves the window to cover packets 6–9, and it sends each packet to its destination. And so the protocol continues.

The actual protocol

A small detail is that TCP works with bytes, not packets. The window has many bytes, and it divides its window up into **segments** and sends each segment via IP.

Because the destination can become over-full with data, whenever the destination sends an acknowledgement, it also tells the sender how big a window it should use (based on how much more data the destination can handle). This is not entirely necessary, since the destination can actually just drop any messages it receives beyond its own capacity, but as long as it is sending an acknowledgement anyway, it might as well try to avoid unnecessary network traffic. This complicates matters slightly for the sender, since the window size will vary.

Another complication in real TCP is the determination of how long to wait between sending a packet and giving up on the acknowledgement. Network traffic varies considerably over time, so it should quickly adapt to changing delays. To do this, the sender keeps track of recent observed delays and computes a weighted average based on this data.

Figure 14.4 diagrams the information appearing in a TCP header that we have seen in this chapter. The TCP includes the source port and the destination port (each 16 bits long) to identify to which connection it belongs (the IP address of the source machine and destination machine are already in the IP header). The sequence number (32 bits) tells which byte of the message begins the segment. The acknowledgement number (32 bits) tells the destination the first byte the sender has not yet received. The header length (4 bits) tells how many 32-bit groups the header contains (so that TCP can tell where the actual message begins). The window size (16 bits) tells how big a window the destination should use (or, equivalently, how much data the sender can handle).

Now that we have a fair understanding of what TCP provides, we can go on to applications using TCP.

[*]Actually, since packets 3 and 4 may have reached their destination (but the sender cannot be sure), TCP implementations are allowed to decide against resending packets 3 and 4 in this case.

Chapter 15

Using messages

TCP and IP give us the ability to send messages reliably between computers. Now we want to use them to do something useful. In this chapter we look at two of the most useful application protocols in existence, the Web-access protocol (HTTP) and the mail protocol (SMTP). We will not learn all of the details of how these protocols work; instead, we look at short common examples of how they are used.

15.1 HTTP

HTTP (**H**yper**t**ext **T**ransfer **P**rotocol) is the basis for Web communication. Since the protocol is so simple, it is ideal for a first look at an application protocol.

Let's say that we want our browser to get the page at

`http://avrim.pc.cs.cmu.edu/index.html`

This jumble of letters means that the browser should use HTTP to request the file "`/index.html`" from `avrim.pc.cs.cmu.edu`. So the browser uses TCP to open a connection to port 80 of `avrim` (80 being HTTP's well-known port number).

Once the browser connects to the server, it tells the server what it wants with the message

```
GET /index.html HTTP/1.1
Accept: text/html
```

The first line says that the browser wants to get the file "`/index.html`" using version 1.1 of HTTP. After this the browser can specify preferences for what it would like. In this example the second line says that the browser prefers HTML. The preferences end with a blank line.

In this case the server responds with the following message and, since there is nothing more to say, closes the connection.

```
HTTP/1.0 200 Document follows
Server: CERN/3.0A
Date: Mon, 11 Jan 1999 03:22:42 GMT
Content-Type: text/html
Content-Length: 115
Last-Modified: Mon, 11 Jan 1999 03:17:24 GMT

<p>I'm <tt>avrim.pc.cs.cmu.edu</tt>; my primary user is
<a href=http://www.cburch.com/>Carl Burch</a>.</p>
```

The first line here gives the basic nature of the response. The server is using version 1.0 of HTTP, and it is responding with a code-200 response; 200 is the code for successful requests. Then the server says several things about the request: The server identifies itself as version 3.0A of CERN's server and tells the time when it received the request. Finally, it says that the file is an HTML file, that it is 115 bytes long, and that it was last modified on January 11. Finally a blank line says that the file is about to start. In this case the file is just two lines of HTML,

```
<p>I'm <tt>avrim.pc.cs.cmu.edu</tt>; my primary user is
<a href=http://www.cburch.com/>Carl Burch</a>.</p>
```

15.2 SMTP

Most mail on the Internet is transfered using SMTP (**S**imple **M**ail **T**ransfer **P**rotocol). It's more complicated than HTTP, but not much worse.

Let's say I'm `spot@cburch.com` working on the machine `avrim.pc.cs.cmu.edu`, and I tell it to send mail to `burch@andrew.cmu.edu`. Then `avrim` opens up a connection to port 25 (SMTP's well-known port number) on the computer `andrew.cmu.edu`. Unlike HTTP, an SMTP transaction is an extended two-sided conversation; in the following, boldface text indicates what `avrim` sends, and normal text indicates what `andrew` sends.

First `andrew` responds with a message welcoming you to the system. Each line begins with a 220 code so that automatic mail systems can just read the code to know what sort of messages are being sent. (We can't expect the automatic system to understand the text.)

```
220-andrew.cmu.edu ESMTP Sendmail 8.8.5/8.8.2
220-Mis-identifying the sender of mail is an abuse of computing facilities.
220 ESMTP spoken here
```

In SMTP, nothing prevents people from lying about who is sending the message. The 'welcome' message at this SMTP server kindly warns you that doing this is abusive behavior. In many cases doing this is grounds for serious penalties (expulsion from school or workplace, perhaps).

Once this is sent, `avrim` sends a message identifying itself using the `helo` command.

```
helo avrim.pc.cs.cmu.edu
250 andrew.cmu.edu Hello AVRIM.PC.CS.CMU.EDU [128.2.185.114], pleased to meet you
```

The server courteously responds that it recognizes the computer. Now the client wants to send mail; first it tells the server the sender and the recipient.

```
mail from: spot@cburch.com
250 spot@cburch.com... Sender ok
rcpt to: burch@andrew.cmu.edu
250 burch@andrew.cmu.edu... Recipient ok
```

The server accepts both of these e-mail addresses as valid. Finally, the client is ready to give the message to be sent using the `data` command. When the server gives the code-354 message, it is ready to receive the message to be sent. The client will insert the message verbatim and finish it off with a line containing a single period.

```
data
354 Enter mail, end with "." on a line by itself
Arf, arf!
.
250 XAA21092 Message accepted for delivery
```

The server commits to delivering the message. The client is now done and so signs off.

```
quit
221 andrew.cmu.edu closing connection
```

Chapter 16

Cryptography

One of the striking things about the Internet protocols is how trusting they are. There is nothing to prevent somebody from listening in on a message (given access to a machine on the path) or from counterfeiting messages.

One of the most attractive options for addressing privacy is **cryptography**. The obvious approach to preventing a spy from reading a message is to hide the message from the spy. Cryptography has a more subtle approach: We do not worry about whether the spy sees the message; instead, we encode the message so that only the intended recipient will understand it.

The idea of cryptography is certainly not new. It has been around at least since Julius Caesar, and war has continued to inspire cryptography. In World War II, when secrets were transmitted by broadcast radio, cryptography blossomed into a full-blown science.

Now cryptography is no longer the domain of soldiers, criminals, and spies. Everybody sends sensitive information (passwords and credit card numbers, for example) across the essentially-public Internet. But using cryptography, we can render electronic communication one of the most secure forms of communication.

This chapter begins by defining different types of cryptographic goals. Then we look at the most simple goal, private-key cryptography. And finally we look at how one can provide interesting and impressive guarantees for some special cases.

16.1 Protocols

Cryptography has a number of applications. In this section we look at a few of the most important goals. In the following, we suppose that Bob wants to deliver a message to Alice, but Eve can eavesdrop.

Private-key cryptography

The traditional form of encryption is **private-key cryptography**. In private-key cryptography, Alice and Bob agree in private on a **key** K. When Bob wants to send his message, he encrypts it so that anybody with K can decrypt it. He sends the encrypted message to Alice, who decodes the message using K. If Eve happens to get what Alice sent, she would have to know K to understand the message.

Public-key cryptography

Private-key cryptography has a crucial shortcoming: Alice and Bob have to agree beforehand on their key. This is inadequate if Alice and Bob have never met privately before. This may happen if Bob has just visited Alice's Web site and decided that he wants to buy something from her store with his credit card.

Public-key cryptography is a way to address Alice and Bob's conundrum. Of course, Alice can't include a private key on her Web site, because Eve could find it too. But she can publicize a public key P that, if a message is encrypted using it, the encryption can only be decoded with a corresponding private key K that only Alice knows. So when Bob responds to Alice's advertisement, he encrypts his message using P. Now only people who have K can decrypt the message, and only Alice has it.

Doing this safely is an ambitious goal. It is somewhat surprising that there are any techniques to do this. But there are a few. One of the most well-known is called **RSA encryption** (named after its inventors, Rivest, Shamir, and Adelson); it is one of the most important pieces of PGP (**P**retty **G**ood **P**rivacy), the most widespread cryptography package on the Internet. An RSA public key is the product of two large prime numbers (of several hundred digits each), and the private key includes the factorization. Breaking RSA essentially requires that the public key be factored. We *believe* this takes impractically long for large numbers, since people have worked on this problem since the ancient Greeks with only moderate success. Prime-Test-All is basically the best algorithm we know, and it takes much to long for large numbers. RSA encryption is therefore interesting and important, but it is too complicated to adequately explain here.

Public-key cryptography is inherently insecure. If Eve can intercept messages between Alice and Bob, then she can pretend to be Alice to Bob and pretend to be Bob to Alice. That is, she creates a set of public and private keys, and she convinces Bob that this public key is really Alice's. Now Bob sends a message which Eve can decrypt. If she wants Alice to receive the message, she can encrypt the message using Alice's real public key. So public-key cryptography has problems. But it forces Eve to masquerade as somebody else rather than just eavesdrop; this is usually much harder to do.

Signatures

A related issue is signing a message. Here we want Alice to be sure that messages from Bob are actually from Bob. For this purpose, Bob publishes a public key P. When he sends a message, he encrypts it using a private key K that Alice doesn't know, in such a way that P can decrypt it. Now Alice can verify that Bob sent the message by seeing if his public key P decrypts it. She can be sure that Bob sent the message insofar as she is sure that P is Bob's public key. Eve couldn't masquerade as Bob unless she could figure out K.

This is called a **signature**. A good signature algorithm is much more difficult to forge than a traditional signature. RSA, it turns out, can also be used as a signature algorithm. But we're still not going to talk about it.

Special-purpose goals

The above protocols were for very general purposes. There are many special-purpose cases that are interesting for cryptographers to consider. How can we vote securely? What is a secure way to bid on an item? How can we transmit money with minimal risk? How can we insure that keys are kept secure? All these are good questions, and cryptographers seek ways to handle these cases.

16.2 Private-key cryptography

It's important that a private key be long enough that computers can't search through all of them to see which one gives a result in an interpretable language. For good cryptographic schemes, searching through all possible keys is often the best known attack.

The simplest and most popularly-understood form of cryptography is the **substitution cipher**, also known as the **secret decoder ring**. In it Alice and Bob agree on a translation between letters. One possible translation is

```
from  _ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 to   A X J E W U I D C H T N M B R L V P O Y G K Z Q F S _
```

So if Bob wants to say, "`I_DO`," to Alice, he would send, "`HAWL`." When Alice receives it, she goes in the opposite direction to get the original message.

There are many ($27! \approx 10^{28}$) possible keys here; Eve is unlikely to be able to try them all to decode Bob's message. But of course, as anybody who has ever solved a newspaper cryptogram knows, this is not very secure. By analyzing letter frequencies, Eve can deduce the original message.

To analyze cryptography rigorously, we need to have a mathematical model of what it means for a protocol to be secure. A particularly strong model, proposed by Alan Turing, is **perfect security**. In perfect security, we model Eve's belief about messages as a probability distribution. Perhaps Eve thinks Bob will say "Yes" with probability 0.7 and "No" with probability 0.3. A protocol is perfectly secure if the message does not change her belief at all; that is, it is secure if for the transmitted message X, for every possible original message M, we have

$$\Pr[M \text{ is original message}] = \Pr[M \text{ is original message, given transmission } X] \ .$$

(This is not true of the substitution cipher, since in seeing X, Eve can immediately eliminate messages not matching the pattern. It is also not true of RSA, since an infinitely-powerful Eve might factor the public key, and this would change her belief.)

Here's a simple scheme worth analyzing, called the **one-time pad**. Here Alice and Bob agree to a stream of random numbers between 0 and 26:

$$\langle 2, 23, 20, 8, 16, 16, 1, 23, 20, 3, \ldots \rangle \ .$$

Now when Bob wants to send the message, he adds the corresponding number to each letter (wrapping around when he reaches `Z`).

$$\begin{array}{cccc} \texttt{I} & \texttt{\_} & \texttt{D} & \texttt{O} \\ +2 & +23 & +20 & +8 \\ \hline \texttt{K} & \texttt{W} & \texttt{X} & \texttt{W} \end{array}$$

So he sends "`KWXW`" to Alice, who then subtracts the same numbers to get the original message. (Bob should also append several spaces to his original message so that Eve doesn't learn anything about the message's length.)

To analyze the one-time pad, we need a fact called **Bayes' theorem**. Bayes' theorem says that for any two events A and B, we have

$$\Pr[A \text{ occurs, given } B \text{ occurs}] = \frac{\Pr[B \text{ occurs, given } A \text{ occurs}] \cdot \Pr[A \text{ occurs}]}{\Pr[B \text{ occurs}]} .$$

The following sequence of equalities is a simple proof of this theorem.

$$\begin{aligned} \frac{\Pr[B \text{ occurs, given } A \text{ occurs}] \cdot \Pr[A \text{ occurs}]}{\Pr[B \text{ occurs}]} &= \frac{\Pr[A \text{ and } B \text{ both occur}]}{\Pr[B \text{ occurs}]} \\ &= \Pr[A \text{ occurs, given } B \text{ occurs}] \end{aligned}$$

We take A to be the event that a particular message M is the actual message Bob sent, and we take B to be the event that Bob sent the encrypted message X. Say that the key and message both have k characters.

Notice that the probability of B occurring given that A occurs (that is, the probability that X is the encryption of M) is $(1/27)^k$, since for each letter of M, there is a $1/27$ chance that the private key happens to include the right rotation to get to the corresponding letter of K. Notice also that the probability of B occurring (regardless of A) is also $(1/27)^k$, since

$$\begin{aligned} \Pr[X \text{ is encryption}] &= \sum_{\text{keys } K} \Pr[K \text{ is key}] \cdot \Pr[M \text{ is } X - K] \\ &= \sum_{\text{keys } K} \left(\frac{1}{27}\right)^k \cdot \Pr[M \text{ is } X - K] \\ &= \left(\frac{1}{27}\right)^k \cdot \sum_{\text{keys } K} \Pr[M \text{ is } X - K] \\ &= \left(\frac{1}{27}\right)^k . \end{aligned}$$

The last step holds because the summation is over all possibilities for the original message M, and the probability that the message is one of its possibilities is exactly 1.

Now we apply Bayes' theorem.

$$\begin{aligned} \Pr[A \text{ occurs, given } B \text{ occurs}] &= \frac{\Pr[B \text{ occurs, given } A \text{ occurs}] \cdot \Pr[A \text{ occurs}]}{\Pr[B \text{ occurs}]} \\ &= \frac{(1/27)^k \cdot \Pr[A \text{ occurs}]}{(1/27)^k} \\ &= \Pr[A \text{ occurs}] . \end{aligned}$$

Thus the one-time pad satisfies our definition of perfect security.

The one-time pad is useful in some very high-security military applications. But it is not adequate for prime-time use, because the number of bits in the key is as long as the message. Transmitting such a long key is as expensive as transmitting the original message itself.

For this reason, we normally go with a goal much weaker than perfect security: **complexity-theoretic security**. In this, we build a cryptographic scheme so that the key cannot be broken quickly unless somebody finds a fast algorithm for a problem for which nobody seriously believes a fast algorithm exists. This is where RSA, factoring, and prime numbers come in: Factoring is a very well-studied problem for which no known algorithm can handle thousands of digits in a practical amount of time.

	Alice	Bob	Carl	Dafna	score
Alice	135	240	301	221	95
Bob	285	363	109	133	88
Carl	135	300	334	83	50
Dafna	132	5	230	116	82
total	286	106	172	152	315

Table 16.1: Communicating the average for $k = 4$, $m = 100$

Exercise 16.1: (Solution, 122)

```
OKXYUOPUAKURCCAPLUBLMR!LBULSUPBMMXYU

BLSU,OQQSVDMIGYUDABLUSJSPUMCUC OTSYU

IOTSUDLACC AK!UBLVMR!LUBLSUBR !SJUDMMXYU

OKXUQRVQ SXUOPUABUIOTSW
```

16.3 Communicating an average

Let's look at a special-purpose cryptographic protocol. Say we have several (k) students, and they want to know their average test score. But none of them wants to tell what their test score is. What can they do?

It turns out that they can find out their average without anybody revealing any information about their score! (This ignores the information that is inherently gained from the average itself.) To do this, we use sums **modulo** $km + 1$, where m is the maximum test score possible. (So $km + 1$ is more than the sum of the scores could possibly be.) The sum of x and y modulo z is the remainder of $x + y$ when divided by z. That is, if x and y are both between 0 and $z - 1$, it will be $x + y$ if $x + y < z$ and $x + y - z$ otherwise.

Say Alice's score is 95. Alice selects $k - 1$ random numbers between 0 and km, and she computes a number n_A between 0 and km such that the sum of n_A and all these random numbers is 95 modulo $km + 1$.

Confidentially, Alice tells each other person one of these random numbers. Each other person does the same: They pick $k - 1$ random numbers, compute some n_i, and communicate the random numbers confidentially. So Alice has been told $k - 1$ numbers, and she still has n_A. Now she adds them up modulo $km + 1$, and tells everybody the result. The sum of what everybody announces will be the sum of all their test scores. (And of course Alice is intelligent enough to divide by k to get the average.)

Now is it really the case that this is the sum? Look at Table 16.1. The score of each person is the sum of the numbers in the row, and we want the total of these. In our protocol, each person announces a column sum, and we add them. These are just two ways of summing all the entries in the table. Since adding modulo $km + 1$ is associative and commutative, both sums are the same.

Now can anybody know anything about Alice's score? They certainly don't from the random number she first told them. And the number she announced to everybody is the sum of her score and several random numbers. The chance that she announces any individual number between 0 and km is $1/(km + 1)$. (The sum of all the random numbers has a uniform distribution between 0 and km, and adding n_A modulo $km + 1$ will only shift this distribution cyclicly, maintaining the uniform distribution.) So that says nothing about n_A. Hence nobody else knows anything more about Alice's score.

Cryptography appears to be the surest way to handle many of the problems that arise in communication. It will continue becoming more prominent as the Internet matures to handle more people and important transactions.

Exercise 16.2: One very important place where cryptography can be used is in voting. Here we have several people wanting to vote yes or no on an issue, and we want to determine which has the majority without revealing any information about individuals' votes. One tempting approach is to use the protocol of Section 16.3 to tally votes. What's wrong with this approach?

FIFTH UNIT

Algorithms

In this unit we look at the study of developing fast algorithms for problems. Our approach is mathematical, in defining algorithms, in proving their correctness, and in analyzing their speed.

Our study begins in Chapter 17 with an examination of how we can analyze an algorithm's speed using *asymptotic analysis*, which basically means that we analyze the algorithm's time requirements for very large problems. This concept is a foundational tool for mathematically analyzing algorithms, so we must learn how to do this well.

After learning the fundamentals, we turn to a selection of two techniques in the development of algorithms that have proven useful for a variety of problems: divide and conquer (Chapter 18) and dynamic programming (Chapter 19). Of course there are many other algorithms and general techniques to study; we choose these two because of their relation to recursion (Chapter 10) and their frequent usefulness.

In each of these latter two chapters, we will see two or three very different problems where the technique applies. As you read each chapter, try to think abstractly about what unites the algorithms in the chapter. By understanding the techniques better, you can become better at writing your own algorithms for other problems.

Chapter 17

Analyzing algorithm speed

Constructing algorithms is easy. But we would like to be able to compare them. In particular, we often want to know how fast they are. How can we determine which of two algorithms is the faster?

Computer scientists have a mathematical approach to answering this question. The answer eliminates the tedium of experimentation and adds much more rigor. This approach is called *asymptotic analysis*. This chapter introduces and explains this concept.

17.1 Comparing algorithms

How can we compare two algorithms' speed? We quickly look at several alternatives.

Implement and test: The most reliable and intuitive approach is to implement both algorithms and to test them. Done correctly, this approach has a definite advantage: It gives strong evidence that the algorithm often works well. There are several problems, though, that lead us to look for other ways. First, implementing algorithms takes a lot of time. Second, the results depend strongly on which computer we use and how well we implement each algorithm. Finally, the algorithm that appears to be better may actually be much slower for many cases not included in the tests. This especially could be a problem if we run our tests on small problems but later, as we work with more powerful computers, we start attempting to use the same algorithm for larger problems.

Extrapolate: The last objection can be met partially by graphing the speed of each algorithm relative to problem size and extrapolating. Unfortunately, as is typical with extrapolating, this can lead to major problems far from the known points (especially if, for example, we fit the points to a line, but actually as problem size increases, the points fit a parabola better).

Create a formula: We can ignore experimentation and just write a formula for the algorithm. For example, we might introduce T_{add} for the time it take our machine to add two numbers together, T_{test} for the amount of time it takes to compare two numbers, and so on. The problem with this approach is that it is quite tedious, and the resulting formula isn't easy to interpret.

Approximate: So what we actually do is called **asymptotic analysis**. The real question, as the problem size gets larger, is: Which term of the formula grows the fastest? As N

increases, does an N term dominate? Or does an N^2 term dominate? (We ignore the constant coefficient for the term, since that makes things more complicated.)

Admittedly, ignoring slower-growing terms and coefficients in this way is extremely crude, but it is an important first cut in deciding which algorithms are worth considering.

We indicate a algorithm's speed using **big-O notation**. For example (as we'll see soon), Prime-Test-All runs in $O(\sqrt{N})$ time (pronounced *order square-root of* N), because as the input number N grows, the dominating term in the time formula is some coefficient times $\sqrt{N}$.

We can define big-O notation explicitly and rigorously. (Don't worry about this definition too much if it's confusing; the intuition is easier to understand and use.) We say a function $f(n)$ is $O(g(n))$ if there are constants c and M so that, for all numbers N past M, we have $f(N) < c \cdot g(N)$.

Before we look at asymptotic analysis of algorithms, we first should get a better feel of the asymptotic bounds of an expression. To do this, you go through each term and determine which term grows fastest for large values, and you ignore the coefficient in this term.

expression		asymptotic bound
$50x^2 + 25x + 40$	$=$	$O(x^2)$
$5096 \log_2 n + 0.02n$	$=$	$O(n)$
$4,236,121$	$=$	$O(1)$
$4 \cdot 2^n \log_2 n + n^2$	$=$	$O(2^n \log_2 n)$

It's important to remember that big-O bounds are *upper bounds*. For example, though $50x^2 + 25x + 40$ is $O(x^2)$, it is also $O(x^3)$ and even $O(2^{2^x})$, since all these grow faster than the fastest-growing term of the expression.

Exercise 17.1: (Solution, 122) Order the following from slowest-growing to fastest-growing as n increases to very large values.

$$\begin{array}{cccc} \sqrt{n} & \log_2 \log_2 n & 2^n & n^2 \\ 1 & \log_2 n & n \log_2 n & n! \end{array}$$

Exercise 17.2: (Solution, 122) Give the best asymptotic bound for each of the following expressions using big-O notation.

a. $3n \log_2 n + 5\sqrt{n}$ **c.** $n! + 8 \cdot 2^n + 5$
b. $8n^2(4 \log_2 n + 3\sqrt{n})$ **d.** $\frac{8+3\log_2 n}{n}$

17.2 Finding big-O bounds

To find the bound for a program, there are some simple rules that you can use. After describing the rules, we go through several examples illustrating these rules at work. (Sometimes these rules are somewhat crude, but don't worry about that for now.)

Constant Rule: All computer actions, except for function calls and iteration statements, take $O(1)$ time.

Sequence Rule: If you do one thing that takes $O(f(n))$ time and then another thing that takes $O(g(n))$ time, then doing both takes $O(f(n)+g(n))$ time.

Iteration Rule: If you go through $O(f(n))$ iterations of a loop, and each iteration takes $O(g(n))$ time, then the time for all iterations is bounded by $O(f(n)\cdot g(n))$.

Function Rule: Calls to functions take as much time as the analysis for that function says. (Recursive calls are more complicated; we defer this issue to Chapter 18.)

In light of these rules, we return to the Prime-Test-All example.

Algorithm Prime-Test-All(N)
1 Let i hold 2.
2 **while** $i^2 \le n$, **do:**
3   **if** i divides n, **then:**
4   **return** `false`.
5   **end of if**
6   Add 1 to i.
7 **end of do**
8 **return** `true`.

When we analyze algorithms like this, we start from the inside and go out. Lines 3, 4, and 6 have no function calls or iteration statements, so each takes $O(1)$ time (Constant Rule). By the Sequence Rule, the total time for lines 3–6 is $O(1+1+1+1)=O(1)$. (Usually we don't get to line 4, but it doesn't hurt to throw it in too.) Now we apply the Iteration Rule to the **while** loop of lines 2–7: We go through $O(\sqrt{N})$ iterations of this loop, and each iteration takes $O(1)$ time, so the total time is $O(\sqrt{N}\cdot 1)=O(\sqrt{N})$. By the Constant Rule, lines 1 and 8 each take $O(1)$ time. We apply the Sequence Rule again: The total amount of time taken by Prime-Test-All is $O(1+\sqrt{N}+1)=O(\sqrt{N})$.

Now let's look at an algorithm for Matrix-Addition. Given two $n\times n$ matrices, we're to find the sum of each corresponding pair of elements.

Algorithm Add-Matrices(A, B)
1 **for each** integer i between 1 and n, **do:**
2   **for each** integer j between 1 and n, **do:**
3   Let $C_{i,j}$ hold $A_{i,j}+B_{i,j}$.
4   **end of do**
5 **end of do**
6 **return** C.

This is slightly more complicated because of the nested loop. Again, the approach is to start with the inside and go out. By the Constant Rule, line 3 takes $O(1)$ time. There are n iterations of the j loop in lines 2–4, and each iteration takes $O(1)$ time, so by the Iteration Rule, lines 2–4 take $O(n\cdot 1)=O(n)$ time. For the i loop in lines 1–5, there are n iterations, and we just saw that each iteration takes $O(n)$ time, so by the Iteration Rule, lines 1–5 take $O(n\cdot n)=O(n^2)$ time. Line 6 takes $O(1)$ time (Constant Rule), so the total amount of time (applying the Sequence Rule to combine lines 1–5 with line 6) is $O(n^2+1)=O(n^2)$. Thus Add-Matrices takes $O(n^2)$ time.

Another example: This C++ function takes a number to a positive integer power. We examined this algorithm in Chapter 10; now we replace the recursion with a loop.

```
 1  double exponentiate(double x, int n) {
 2      double ret = 1.0;
 3      double y = x;
 4      int i = n;
 5      while(i > 0) { // always at this point y^n == x^i * ret
 6          if(i % 2 == 0) { // i is even
 7              i = i / 2;
 8          } else { // i is odd
 9              ret = ret * x;
10              i = (i - 1) / 2;
11          }
12          x = x * x;
13      }
14      return ret; // i must be 0, so y^n = x^0 * ret = ret
15  }
```

By the Constant and Sequence Rules, each iteration of lines 6–12 takes $O(1)$ time. But how many times do we go through the loop? After each iteration, i is at most half of what it was before, so after k iterations, i is at most $n(\frac{1}{2})^k$. Thus if go through $k = \log_2 n$ iterations, i is at most $n(\frac{1}{2})^{\log_2 n} = \frac{n}{n} = 1$. One more iteration brings i to 0, so there are *at most* $1 + \log_2 n$ iterations. (Sometimes we will finish the loop sooner. But it will always stop in at most $1 + \log_2 n$ iterations, and an upper bound is all we need for big-O bounds.) Each iteration takes $O(1)$ time, so by the Iteration Rule, lines 5–13 take $O(\log_2 n)$ time. The amount of time consumed by the statements outside the loop is $O(1)$. So by the Sequence Rule, the amount of time Fast-Exponentiate requires is $O(1 + \log_2 n) = O(\log_2 n)$.

Now we look at an example involving the Function Rule. Say we want to count the number of primes between 2 and N. The following would do this.

Algorithm Count-Primes(N)
1 Let $count$ hold 0.
2 **for each** i between 2 and N, **do:**
3   **if** Prime-Test-All$(i) = \texttt{true}$, **do:**
4   Add 1 to $count$.
5   **end of if**
6 **end of do**
7 **return** $count$.

By the Function Rule, line 3 takes $O(\sqrt{i})$ time, and since it is always the case that $i \leq N$, this is $O(\sqrt{N})$. Line 4 takes $O(1)$ item (Constant Rule), so each iteration of lines 3–5 takes $O(\sqrt{N})$ time (Sequence Rule). We go through $N - 1$ iterations of the loop in line 2, so lines 2–6 take a total of $O((N-1)\sqrt{N}) = O(N\sqrt{N})$ time (Iteration Rule). Lines 1 and 7 each take $O(1)$ time, so the total amount of time for Count-Primes is $O(1 + N\sqrt{N} + 1) = O(N\sqrt{N})$.

Exercise 17.3: (Solution, 122) In Exercise 2.2, you invented and compared algorithms for the Square-Root problem. Using big-O notation, analyze the speed of each of the following Square-Root algorithms. Describe the best big-O bound you can find.

```
int squareRootA(int n) {
    int i = 0; // find the least i whose square is less than n
    while(i * i <= n) {
        i = i + 1;
    }
    return i - 1;
}
```

```
int squareRootB(int n) {
    int i = n; // find the greatest i whose square is more than n
    while(i * i > n) {
        i = i - 1;
    }
    return i;
}

int squareRootC(int n) {
    int low = 0;   // this algorithm works by successively halving
    int high = n;  // range (low, high), as dictionary searching
    while(high - low > 0) {
        int mid = (low + high) / 2;
        if(mid * mid < n) {
            low = mid + 1;
        } else if(mid * mid > n) {
            high = mid;
        } else {
            return mid;
        }
    }
    if(low * low <= n) {
        return low;
    } else {
        return low - 1;
    }
}

int squareRootD(int n) { // assumes n perfect square
    // take every other number in the prime factorization
    int ncur = n;
    int sqrt = 1; // always we have (sqrt * sqrt) * ncur == n
    for(int i = 2; ncur != 1; i = i + 1) {
        while(ncur % i == 0) {
            ncur = ncur / i / i;
            sqrt = sqrt * i;
        }
    }
    return sqrt;
}
```

Chapter 18

Divide and conquer

One of the most useful general algorithmic approaches is **divide and conquer**. Algorithms using this approach solve a problem in three steps.

1. Split the problem into smaller, similar subproblems.
2. Solve each of these problems using recursion.
3. Combine the solutions into a solution for the original problem.

We will see how to use the divide-and-conquer technique for two important problems: sorting and multiplication.

18.1 Sorting

One of the classic — and one of the most useful — instances of using the divide-and-conquer approach is in sorting an array.

Problem Sort:
Input: an array A of integers.
Output: an array in increasing order, containing each integer exactly as often as it occurs in A.

For example, given the input array

$$A = \langle 19, 1, 29, 30, 6, 15, 2, 5 \rangle$$

we would want to output

$$\langle 1, 2, 5, 6, 15, 19, 29, 30 \rangle .$$

The Merge-Sort algorithm

Our strategy is to divide the problem in two using the simplest possible method: We split the array down the middle. Recursively we sort both halves. How can we combine these solutions? The two solutions may overlap, but we can combine them into a single sorted list by merging the two sorted solutions in a zipper fashion.

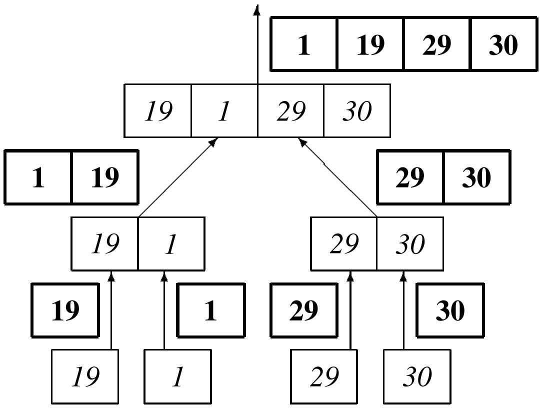


Figure 18.1: Sorting the array $\langle 19, 1, 29, 30\rangle$ using Merge-Sort. (Italic numbers are inputs, Roman numbers are outputs.)

Algorithm Merge(A, B)
// A and B must be already-sorted arrays
Let n_A and n_B be the length of arrays A and B.
Let C hold an array of length $n_A + n_B$. *// C will be the result*
Let a, b, and c hold 0. *// current positions in A, B, and C*
while $a < n_A$ **or** $b < n_B$, **do:**
  if $a < n_A$ **and** $A_a < B_b$, **then:**
    Let $C[c]$ hold $A[a]$.
    Add 1 to a and c.
  else:
    Let $C[c]$ hold $B[b]$.
    Add 1 to b and c.
  end of if
end of loop
return C.

Since every time through the loop we handle one of the items in A or B, we go through the loop exactly $n_A + n_B$ times. Each iteration of the loop, since it involves no loops or function calls, takes $O(1)$ time. So Merge consumes $O(n_A + n_B)$ time.

With Merge in hand we can write our divide-and-conquer sorting algorithm, Merge-Sort.

Algorithm Merge-Sort(A)
if A has only one item, **then:**
  return A.
else:
  Let A_0 hold first half of A.
  Let A_1 hold second half of A.
  return Merge(Merge-Sort(A_0), Merge-Sort(A_1)).
end of if

Figure 18.1 gives an example of using Merge-Sort on the array $\langle 19, 1, 29, 30\rangle$. We divide it into two arrays $\langle 19, 1\rangle$ and $\langle 29, 30\rangle$. Recursively we sort each to get $\langle 1, 19\rangle$ and $\langle 29, 30\rangle$. And finally we merge these two arrays and return $\langle 1, 19, 29, 30\rangle$.

Time analysis of Merge-Sort

To analyze the time this algorithm consumes, we write a recurrence for $T(n)$, the time to sort n numbers. Let us assume for convenience that n is a power of 2. In the base case, when $n = 1$, $T(n)$ is $O(1)$. For other n, we have three steps to analyze. We first divide the array into two pieces of length $n/2$; this takes $O(n)$ time. Then we recursively sort the two subpieces; each recursive call takes $T(n/2)$ time by induction, so sorting both pieces takes $2T(n/2)$ time. And finally we merge the two lists, taking $O(n/2 + n/2) = O(n)$ time. Thus the total amount of time is

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

That is, for some c, $T(n)$ is at most $2T(n/2) + cn$.

To solve this recurrence, we will apply the recurrence to itself until we reach the base case of $T(1)$.

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

In general, after applying the recurrence to itself k times we have

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

Let us take k to be $\log_2 n$, so that $n/2^k$ is $n/n = 1$. In this case we have

$$\begin{aligned} T(n) &\leq 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)cn \\ &= nT(n/n) + cn \log_2 n \\ &= nO(1) + O(n \log_2 n) \\ &= O(n \log_2 n) \end{aligned}$$

Thus we have that the total amount of time for Merge-Sort is $O(n \log_2 n)$.

(We'll not see why in this book, but the general form of Sort cannot be done in less than $O(n \log_2 n)$ time. The Merge-Sort algorithm, then, is optimal within a constant factor.)

18.2 Multiplication

Given two n-digit numbers a and b, the Multiplication problem is to find their product.

Problem Multiplication:
Input: numbers a and b of n digits each.
Output: the product of a and b.

Our goal is find a quick multiplication algorithm.

The grade-school method

The multiplication method that you probably know from grade school works fairly well. It involves going though each digit of b and multiplying that single digit with a, and then adding the

$$
\begin{array}{r}
1215 \\
\times \quad 1998 \\
\hline
9720 \\
10935 \\
10935 \\
+\,1215 \\
\hline
2427570
\end{array}
$$

Figure 18.2: Example of grade-school multiplication.

results in a special way. In case you suffer from calculator-induced forgetfulness, Figure 18.2 illustrates this method.

This is not a divide-and-conquer technique, but we should analyze it first to get a point of comparison: How much time does the grade-school approach take? We have n digits of b to multiply by a; each of these multiplications takes $O(n)$ time, so the first step of writing down the numbers takes $O(n^2)$ time. Then we add together n numbers, each having at most $2n$ digits. Adding two $2n$-digit numbers takes $O(n)$ time, so the addition step takes $O(n^2)$ time. Thus the grade school method takes $O(n^2)$ time.

Karatsuba's method

We now expose the ruse your grade school teacher played on you: There is a better way. Using divide and conquer, we can multiply in $O(n^{1.59})$ time!

Say we divide a into two pieces, a_L and a_R, where a_L has the top $n/2$ digits of a and a_R has the bottom $n/2$ digits. So a is $a_L \times 10^{n/2} + a_R$. Divide b likewise into b_L and b_R. Then $a \times b$ can be written as

$$(a_L \times 10^{n/2} + a_R)(b_L \times 10^{n/2} + b_R) = a_L b_L \times 10^n + (a_L b_R + a_R b_L) \times 10^{n/2} + a_R b_R \, .$$

This already gives us a divide-and-conquer algorithm: Divide a and b into two pieces each, find the four products ($a_L b_L$, $a_L b_R$, $a_R b_L$, and $a_R b_R$), and add them together as the equation tells us.

We can analyze this algorithm by writing another recurrence, letting $T(n)$ represent the amount of time taken to multiply two n-digit numbers. Dividing a and b into two pieces takes $O(n)$ time. Four multiplications of $n/2$-digit numbers will take $4T(n/2)$ time. And adding the results together according to the equation involves adding four numbers, each with at most $2n$ digits. This takes $O(n)$ time. So our recurrence is

$$T(n) \leq 4T(n/2) + O(n) \, .$$

We can solve this similarly to how we solved the Merge-Sort recurrence; if we did this we would find an answer of $T(n) = O(n^2)$. This is not an improvement over the grade-school method.

The problem is that we save nothing by multiplying four numbers each with $n/2$ digits. But, using a bit of cleverness, we can conserve our multiplies. In particular, say we calculate x_1, x_2, and x_3 as follows.

$$x_1 \;=\; a_L b_L$$

$$\begin{aligned} x_2 &= (a_L + a_R)(b_L + b_R) \\ x_3 &= a_R b_R \end{aligned}$$

Each of these three quantities involves multiplying two $n/2$-digit numbers. (Calculating x_2 may involve multiplying numbers of $n/2 + 1$ digits, but for large n the additive 1 is not significant. For convenience we ignore it.) From these we already have $a_L b_L$ and $a_R b_R$. The clever thing, though, is that we also have $a_L b_R + a_R b_L$, because we can obtain it by subtracting x_1 and x_3 from x_2. Thus we can write ab as

$$ab = a_R b_R \times 10^n + (a_L b_R + a_R b_L) \times 10^{n/2} + a_R b_R = x_1 \times 10^n + (x_2 - x_1 - x_3) \times 10^{n/2} + x_3 \,.$$

So we can multiply two n-digit numbers by multiplying only *three* $n/2$-digit numbers!

This gives us a new, faster multiplication algorithm, invented by Karatsuba in 1962.

Algorithm Karatsuba-Multiply(a, b)
if a or b has one digit, **then:**
    return $a \cdot b$.
else:
    Let a_L hold the higher $n/2$ digits of a.
    Let a_R hold the lower $n/2$ digits of a.
    Let b_L hold the higher $n/2$ digits of b.
    Let b_R hold the lower $n/2$ digits of b.
    Let x_1 hold Karatsuba-Multiply(a_L, b_L).
    Let x_2 hold Karatsuba-Multiply$(a_L + a_R, b_L + b_R)$.
    Let x_3 hold Karatsuba-Multiply(a_R, b_R).
    return $x_1 \times 10^n + (x_2 - x_1 - x_3) \times 10^{n/2} + x_3$.
end of if

Figure 18.3 contains an example of running this algorithm on 1215 and 1998.

Time analysis of Karatsuba-Multiply

We now analyze the running time of Karatsuba-Multiply. Each addition or subtraction in the algorithm takes $O(n)$ time, and we have 3 recursive calls, each for multiplying numbers with (about) $n/2$ digits. So we obtain the recurrence

$$T(n) \le 3T(n/2) + cn \,,$$

for some number c. Our approach to solving this recurrence is similar to that for Merge-Sort: We find a general equation for unrolling the recurrence k times, and then we use a value of k for which we get $T(n)$ in terms of the base case, $T(1)$.

$$\begin{aligned} T(n) &\le 3T(n/2) + cn \\ &\le 3(3T(n/4) + cn/2) + cn = 9T(n/4) + (3/2 + 1)cn \\ &\le 9(3T(n/8) + cn/4) + (3 + 1)cn = 27T(n/8) + (9/4 + 3/2 + 1)cn \\ &\;\vdots \\ &\le 3^k T(n/2^k) + \left(\left(\frac{3}{2}\right)^{k-1} + \left(\frac{3}{2}\right)^{k-2} + \cdots + \frac{3}{2} + 1 \right) cn \end{aligned}$$

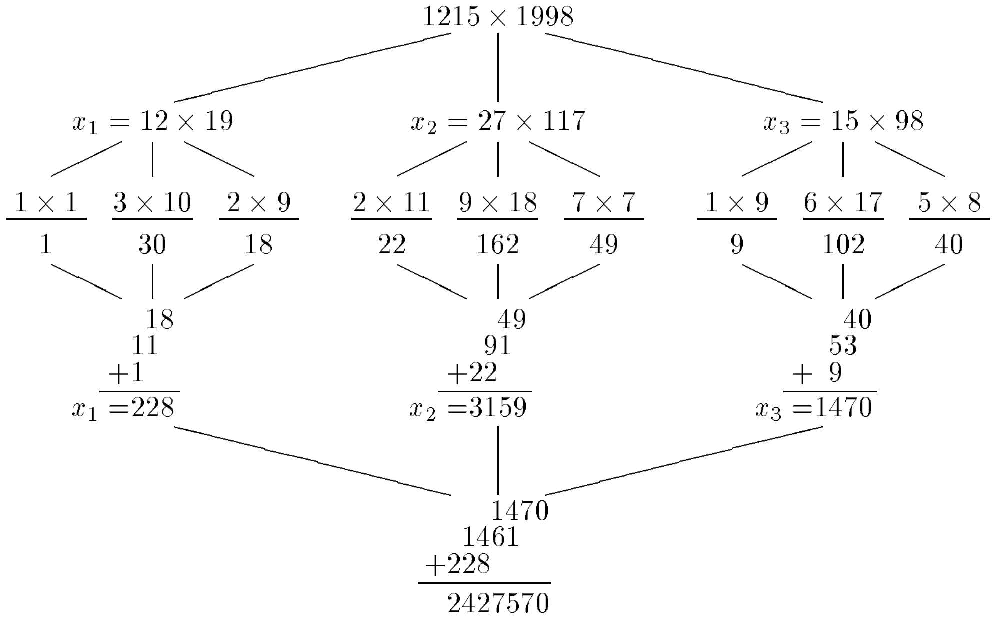


Figure 18.3: Example of multiplying using Karatsuba-Multiply.

To simplify this somewhat, we factor $(3/2)^{k-1}$ from the second term, and we extend the geometrically decreasing series infinitely. (This extension only increases the sum.) It is a fact that for $r < 1$, the sum $1 + r + r^2 + r^3 + \cdots = 1/(1-r)$; in this case, r is $2/3$, so the sum is 3. We apply this fact to our bound too.

$$\begin{aligned}
T(n) &\leq 3^k T(n/2^k) + \left(\left(\frac{3}{2}\right)^{k-1} + \left(\frac{3}{2}\right)^{k-2} + \cdots + \frac{3}{2} + 1\right) cn \\
&= 3^k T(n/2^k) + \left(\frac{3}{2}\right)^{k-1} \left(1 + \frac{2}{3} + \left(\frac{2}{3}\right)^2 + \cdots + \left(\frac{2}{3}\right)^{k-1}\right) cn \\
&\leq 3^k T(n/2^k) + \left(\frac{3}{2}\right)^{k-1} \left(1 + \frac{2}{3} + \left(\frac{2}{3}\right)^2 + \cdots\right) cn \\
&= 3^k T(n/2^k) + 3c \left(\frac{3}{2}\right)^{k-1} n = 3^k T(n/2^k) + 2c \left(\frac{3}{2}\right)^{k} n
\end{aligned}$$

Our bound for $T(n)$ is in terms of $T(1)$ when $k = \log_2 n$. In this case 3^k is

$$3^k = \left(2^{\log_2 3}\right)^{\log_2 n} = 2^{\log_2(3)\cdot\log_2(n)} = 2^{\log_2(n)\cdot\log_2(3)} = \left(2^{\log_2(n)}\right)^{\log_2 3} = n^{\log_2 3}$$

We can use this to simplify our recurrence

$$T(n) \leq 3^k T(n/2^k) + 2c\frac{3^k}{2^k} n = n^{\log_2 3} T(1) + 2c\frac{n^{\log_2 3}}{n} n = O(n^{\log_2 3})\,.$$

Thus the time bound for Karatsuba-Multiply is $O(n^{\log_2 3}) \approx O(n^{1.59})$.

Performance of Karatsuba-Multiply in practice

In a certain sense the theoretical analysis unsatisfying. It is not really a proof that Karatsuba-Multiply is always faster; it is a proof that is it faster for very large n. Indeed, the relative

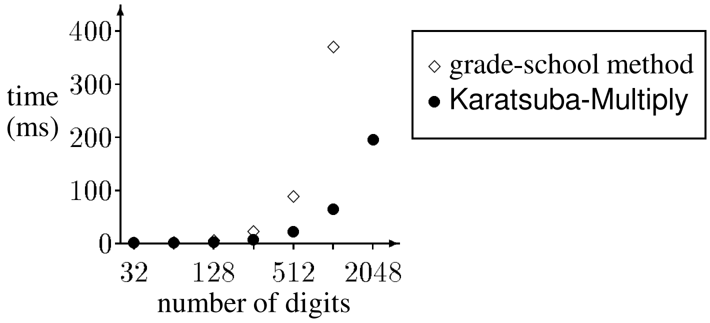


Figure 18.4: Multiplication experiment results. (Note the logarithmic x-axis.)

complexity of Figures 18.2 and 18.3 suggests that perhaps your grade-school teacher was right not to teach you Karatsuba-Multiply. Our analysis shows that Karatsuba-Multiply becomes faster at some point, but that point may be impractically large (hundreds of thousands of digits, maybe).

To see how the algorithms performed with small numbers, the author performed an experiment comparing Karatsuba-Multiply with the grade-school algorithm. Figure 18.4 graphs the results. The graphed measurements are from a Sun SPARCstation 4.

What we see from the results is that the experimental results follow the theoretical analysis very closely. The difference becomes a factor of 2 at 64-digit numbers, and it widens thereafter.

But why, you will ask if you are properly inquisitive, would anybody want to multiply 64-digit numbers? "We never need such precision in real life!" you can object. There are cases when this is important. One case is cryptography. Many cryptographic protocols involve choosing and multiplying keys containing hundreds of digits; this multiplication must be exact. Moreover, the more digits in a key, the more secure the cryptography. A quick multiplication algorithm, then, can help make messages more secure by allowing us to choose larger keys.

In this chapter we have seen and analyzed two divide-and-conquer algorithms: one for sorting, one for multiplication. Both problems are fundamental to computer programs. Divide and conquer applies to many other problems. It is an important technique that is worth considering for nearly any problem you might cross.

Chapter 19

Dynamic programming

In applying **dynamic programming** to a problem, we observe that we could quickly find the problem's solution if we had solutions to some similar but smaller problems. To solve these similar, smaller problems, we can use solutions to similar, yet smaller problems. These will take similar, tiny problems, which require solutions to miniscule problems, and so on. Finally we get to something a problem so small it is trivial.

So far, this description is similar to divide-and-conquer approaches. The difference is that dynamic programming applies when the recursive solution requires recomputation to the same subproblem many times. To apply the technique in an algorithm, we begin with the trivial problems and work our way up until getting to the problem at hand. This allows us to avoid recomputing the same answer to a problem many times.

Dynamic programming algorithms tend to require quite a bit of insight. (Actually, Karatsuba-Multiply required a neat bit of insight itself!) The best way to demonstrate how it can work is to try an example.

19.1 Fibonacci numbers

One of the simplest examples of using dynamic programming is in computing Fibonacci numbers. The Fibonacci numbers turn up in a variety of places that we won't discuss here. Instead, we simply define them: The **Fibonacci sequence** begins with the numbers

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

The first two numbers are 1, and each other number is the sum of the two numbers preceding it. Using this definition, a simple recursive function will compute the nth Fibonacci number.

Algorithm Fibonacci(n)
if $n \leq 1$, **then:**
  return 1.
else:
  return Fibonacci$(n-1)$ + Fibonacci$(n-2)$.
end of if

Look at the recursion tree for Fibonacci(6) in Figure 19.1. You can see that we frequently recompute values. For example, Fibonacci(2) is computed 5 times.

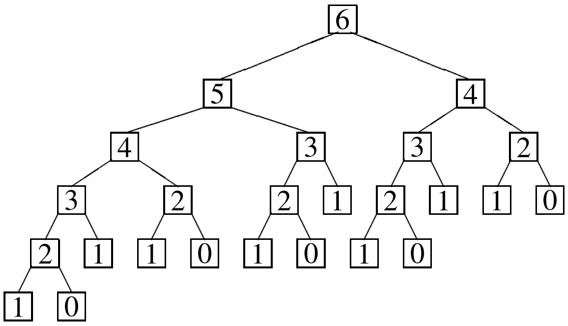


Figure 19.1: Recursion tree for Fibonacci(6).

Because of this recomputation, this algorithm is quite slow. Although we won't discuss why, in big-O notation, Fibonacci takes $O(1.62^n)$ time. In the author's tests, computing Fibonacci(40) took 75.2 seconds. The execution time rapidly increases with n: Computing Fibonacci(70) would take 4.4 years!

The dynamic programming approach stands Fibonacci on its head. Rather than compute starting at n, we will begin at 0 and work up. We store the Fibonacci numbers in an array so that to compute each Fibonacci we can access previous Fibonacci numbers without recomputation.

Algorithm Dynamic-Fibonacci(n)
Let fib_0 hold 1.
Let fib_1 hold 1.
for each i **from** 2 **to** n, **do:**
  Let fib_i hold $fib_{i-1} + fib_{i-2}$.
end of loop
return fib_n.

Each iteration of the loop here takes constant time, and there are $n - 1$ iterations. So this algorithm takes $O(n)$ time. A timing on the same computer as earlier demonstrates how much of an improvement this is. Computing Dynamic-Fibonacci(40) took 2 microseconds, and Dynamic-Fibonacci(70) took 3 microseconds.

19.2 Making change

Now we look at another example where we can apply dynamic programming. Say we're given a huge bag of coins containing an infinite number of coins for each of a set of denominations $\{d_0, d_1, \ldots, d_{n-1}\}$, and we want to make change for some amount amt. The Make-Change problem asks, "How can we use the fewest coins to get exactly amt?"

Problem Make-Change:
Input: n denominations $d_0, d_1, \ldots, d_{n-1}$, amount amt.
Output: the smallest number of coins needed so that their net worth is amt (we can use a denomination as often as we please).

We do this daily using the greedy method, and in American denominations the greedy method works. (The greedy method would take the most valuable coin whose value is at most amt, then the most valuable coin worth at most whatever's left, and so on until we reach 0.) The fact that this works is a result of what American coins are worth. For general denominations this method doesn't always give an exact answer. Indeed, it may not find a solution when one exists!

For example, if we have pennies, four-cent pieces, and nickels, and we want to make change for eight cents, we should choose two four-cent pieces even though the greedy method would recommend a nickel and three pennies. If we don't have pennies available, the greedy method would flail helplessly over the three cents after the nickel!

A dynamic programming approach observes that, if we knew how many coins it takes to handle $amt - d_0$, $amt - d_1, \ldots$, $amt - d_{n-1}$, then making change for amt would take only one coin more than the minimum of these. (If it took i coins to reach $amt - d_j$, we could use these i coins needed to reach $amt - d_j$ and then use a d_j-coin, for a total of $i + 1$ coins worth amt.) Finding the best solution for $amt - d_j$ is just a matter of solving the same way, so we can recurse. The base case here is when the amount is zero; we can change $amt = 0$ with no coins at all.

To program the solution, we start at the trivial case of zero and work our way to amt, remembering in the array $coins$ the number of coins required to change each amount a along the way.

Algorithm Dynamic-Make-Change$(amt, d_0, d_1, \ldots d_{n-1})$
Let $coins_0$ hold 0.
for each a **from** 1 **to** amt, **do:**
    Let $coins_a$ hold ∞.   *// an upper bound on* $coins_a$
    for each j **from** 0 **to** $n-1$, **do:**
        if $d_j \le a$ **and** $coins_{a-d_j} + 1 < coins_a$, **then:**
            Let $coins_a$ hold $coins_{a-d_j} + 1$.
        end of if
    end of loop
end of loop
return $coins_{amt}$.

How much time does this algorithm take? An iteration of the inner loop takes $O(1)$ time, and there are n iterations of it, so the inner loop takes $O(n)$ time. Each iteration of the outer loop, then, is $O(n)$, and it goes amt times. Thus the total time taken is $O(amt \cdot n)$.

19.3 All-pairs shortest paths

Now we'll consider shortest paths in a graph.

Problem All-Pair-Paths:
Input: an undirected graph with nonnegative edge lengths.
Output: for each pair of vertices, the length of the shortest path between them.

For simplicity, we number the vertices 1 to n, and define a distance matrix d so that $d_{i,j}$ is the length of the edge between i and j if it exists and infinite (∞) otherwise. We want to compute the matrix of $p_{i,j}$, where $p_{i,j}$ is the length of the shortest path from i to j.

How can we approach this with dynamic programming? This requires quite a bit of insight. Here's the observation. If we want the shortest path between s and t in an n-vertex graph, we can find it quickly if we already know the shortest paths *that only involve the first* $n-1$ *vertices* (except the endpoints themselves). That is, if we know the shortest path between every i and j

that does not pass through vertex n along the way, then we can find the shortest path over all the n vertices. We use the following method. Say that $p_{i,j}^{(n-1)}$ is the length of the shortest path between i and j that has only the first $n-1$ vertices as intermediate steps. Then the shortest path from s to t on all n vertices will be the minimum of $p_{s,t}^{(n-1)}$ and $p_{s,n}^{(n-1)} + p_{n,t}^{(n-1)}$. The first alternative ($p_{s,t}^{(n-1)}$) represents the case that the shortest path from s to t does not go through n. The second ($p_{s,n}^{(n-1)} + p_{n,t}^{(n-1)}$) represents the case that it does. In this case, it will only go through n once. (Can you prove this?) The path will first extend from s to n, with length $p_{s,n}^{(n-1)}$. Then it will extend from n to t, with length $p_{n,t}^{(n-1)}$. So its total length in this case is $p_{s,n}^{(n-1)} + p_{n,t}^{(n-1)}$.

We can likewise compute the $p_{i,j}^{(n-1)}$ if we know the $p_{i,j}^{(n-2)}$, which we can compute if we know the $p_{i,j}^{(n-3)}$, and so on until we get to the $p_{i,j}^{(0)}$. This is the trivial case: The shortest path between i and j going through *no* intermediate vertices is $d_{i,j}$.

As before, the dynamic programming algorithm will stand the computation on its head. We begin with finding the $p^{(0)}$ matrix; gradually, we work up to the $p^{(n)}$ matrix, at which point we are done.

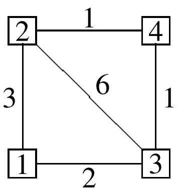
Algorithm Dynamic-Paths(n, d)
// Initialize $p^{(0)}$
for each s **from** 1 **to** n, **do:**
    for each t **from** 1 **to** n, **do:**
        if there is an edge between s and t, **then:**
            Let $p_{s,t}^{(0)}$ hold length of edge between s and t.
        else:
            Let $p_{s,t}^{(0)}$ hold ∞.
        end of if
    end of loop
end of loop
// Now work up to $p^{(n)}$ *by computing successive* $p^{(k)}$.
for each k **from** 1 **to** n, **do:**
    for each s **from** 1 **to** n, **do:**
        for each t **from** 1 **to** n, **do:**
            Let $p_{s,t}^{(k)}$ hold $\min\{p_{s,t}^{(k-1)}, p_{s,k}^{(k-1)} + p_{k,t}^{(k-1)}\}$.
        end of loop
    end of loop
end of loop
return $p^{(n)}$.

This algorithm takes $O(n^3)$ time. Figure 19.2 gives an example of how this problem would work on a specific graph.

Summary

In this chapter we have investigated how dynamic programming helps in three very different problems: computing Fibonacci numbers, making change, and finding all the shortest paths in a graph. Dynamic programming shows up in many other problems, too.

The problems where it applies have this characteristic: One can phrase the solution to a problem in terms of recursive solutions to smaller problems from the same problem, and these



$$p^{(0)} \quad \begin{matrix} 0 & 3 & 2 & \infty \\ 3 & 0 & 6 & 1 \\ 2 & 6 & 0 & 1 \\ \infty & 1 & 1 & 0 \end{matrix} \qquad p^{(3)} \quad \begin{matrix} 0 & 3 & 2 & \mathbf{3} \\ 3 & 0 & 5 & 1 \\ 2 & 5 & 0 & 1 \\ \mathbf{3} & 1 & 1 & 0 \end{matrix}$$

$$p^{(1)} \quad \begin{matrix} 0 & 3 & 2 & \infty \\ 3 & 0 & \mathbf{5} & 1 \\ 2 & \mathbf{5} & 0 & 1 \\ \infty & 1 & 1 & 0 \end{matrix} \qquad p^{(4)} \quad \begin{matrix} 0 & 3 & 2 & 3 \\ 3 & 0 & \mathbf{2} & 1 \\ 2 & \mathbf{2} & 0 & 1 \\ 3 & 1 & 1 & 0 \end{matrix}$$

$$p^{(2)} \quad \begin{matrix} 0 & 3 & 2 & \mathbf{4} \\ 3 & 0 & 5 & 1 \\ 2 & 5 & 0 & 1 \\ \mathbf{4} & 1 & 1 & 0 \end{matrix}$$

Figure 19.2: An example of using Dynamic-Paths.

smaller problems overlap. When you see this, it is time to consider dynamic programming. The dynamic programming algorithm will begin with the small — or easy — problems and build on previous solutions, remembering solutions to more and more complicated problems until finally reaching the actually asked problem.

Sometimes the dynamic programming solution is obvious enough that only after you see the algorithm do you realize it uses dynamic programming. But sometimes its application is less obvious. When you encounter a problem, it is often worthwhile to consider how dynamic programming might be applied; the thought could easily pay off in a good solution.

SIXTH UNIT

Appendices

This textbook includes four appendices.

Appendix A is a quick-reference listing the fundamentals of C++ syntax.

Appendix B lists miscellaneous keyboard symbols, their names, and how they are used in C++.

Appendix C describes some fundamental math concepts that are used elsewhere in the book.

Appendix D gives solutions to many of the exercises appearing in the book.

Appendix A

C++ syntax reference

Program structure

```
#include <iostream>
#include <string>

int main() {
    ⟨programBody⟩
}
```

Function definition

```
⟨returnValueType⟩ ⟨functionName⟩(⟨parameterList⟩) {
    ⟨functionBody⟩
}
```

Exiting a function

```
return ⟨expressionWhoseValueToReturn⟩;
```

Variable declaration

```
⟨typeOfVariable⟩ ⟨variableToDefine⟩;
vector<⟨elementType⟩> ⟨arrayToDefine⟩(⟨lengthOfArray⟩);
```

Variable assignment

```
⟨variableToChange⟩ = ⟨valueToGiveIt⟩;
```

Conditional statements

```
if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfTrue⟩
}

if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfTrue⟩
} else {
    ⟨statementsToDoIfFalse⟩
}
```

```
if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfThisTrue⟩
} else if (⟨thatIsTrue⟩) {
    ⟨statementsToDoIfThisFalseButThatTrue⟩
} else {
    ⟨statementsToDoIfBothFalse⟩
}
```

Iteration statements

```
while(⟨thisIsTrue⟩) {
    ⟨statementsToRepeat⟩
}

for(⟨initialAssignment⟩; ⟨thisIsTrue⟩; ⟨updateAssignment⟩) {
    ⟨statementsToRepeat⟩
}
```

Types Types included in C++:

`int`	integer
`double`	number
`char`	character
`string`	string

The array type is `vector<`⟨***elementType***⟩`>`, and a structure type is declared as follows.

```
struct ⟨structureTypeName⟩ {
    ⟨elementDeclarations⟩;
};
```

Expression operators The following table lists the operators we saw in the book. Operators listed higher have a higher order of precedence; operators on the same line have the same priority.

rank	operator	meaning
1	`::`	object specifier
2	`(···)`, `[···]`, `.`	function call, array index, structure member
3	−, !	negation, logical not
4	*, /, %	multiplication, division, remainder
5	+, −	addition, subtraction
6	<<, >>	output, input
7	<, <=, >=, >	comparison
8	==, !=	equals, not equals
9	&&	logical and
10	||	logical or
11	=	assignment

Reserved words C++ uses a variety of **keywords** (also called **reserved words**) for special purposes. These cannot be used as names for other things. Some of these we have seen in this textbook.

keyword	page	usage in C++
`char`	21	the type for characters
`class`	45	define a new object type
`const`	38	for constant-reference parameters
`double`	20	the type for real numbers
`else`	27	for an “otherwise” clause in `if` statements
`for`	30	loop through statements *for* items in a set
`if`	25	do some statements *if* a condition holds
`int`	20	the type for integers
`private`	47	for indicating an object’s hidden members
`public`	47	for indicating an object’s public members
`return`	36	exit from a function with a given value
`struct`	42	define a new structure type
`void`	35	the return type for functions returning nothing
`while`	29	loop through statements *while* a condition holds

Many more keywords we have not seen at all in this book.

```
asm        enum       protected  this
auto       extern     register   throw
break      float      short      try
case       friend     signed     typedef
catch      goto       sizeof     union
continue   inline     static     unsigned
default    long       switch     virtual
delete     new        template   volatile
do         operator
```

Appendix B

Symbols

symbol	name	C++ meaning seen in this book
!	exclamation point, bang, not	not operator `!`; inequality operator `!=`
@	at sign	
#	sharp, pound, hash mark	used in `#include` statements
$	dollar sign	
%	percent, mod	remainder operator `%`
^	caret, exponentiation	
&	ampersand, and	and operator `&&`; indicates reference parameter
*	asterisk, star, multiplication	multiplication operator `*`
()	parentheses	orders expression evaluation; function call operator `()`; delimits condition in `if`, `for`, `while`
−	minus, dash	negation operator −; subtraction operator −
_	underscore	part of a name of variable or function
=	equal sign	assignment operator =; comparison operators == `!=` <= >=
+	plus	addition operator +
\	backslash	quote next character literally (as in `'\''` or `"\\"`)
|	vertical bar, or	or operator `||`
`	backquote	
~	tilde, twiddle	
[]	brackets	array indexing operator `[]`
{}	braces	delimits statement block
;	semicolon	terminates statement
:	colon	qualifies object (in `::`); ends `private` and `public`
'	single-quote	delimits character constant
"	double-quote	delimits string constant, delimits file in `#include`
,	comma	separates function parameters
.	period, dot	structure member operator `.`, decimal
<>	angle brackets, less/greater than	comparison operators < > <= >=; input/output operators << >>; delimits file in `#include`; delimits vector element type
/	slash, virgule, division	division operator `/`; begins comments `//`
?	question mark	

Appendix C

Mathematical concepts

You may not be familiar with all of the mathematical concepts found in this book; or maybe you are. At any rate, in this appendix we discuss some of the more obscure mathematical concepts we assume. Go through it quickly to make sure you understand it; and, that which you don't, learn.

C.1 Logarithms

A **logarithm** is the inverse of exponentiation. The logarithm base b of x, written as $\log_b x$, is the value of y for which $b^y = x$.

Logarithms have a number of important properties. Here are a few of them.

$$\begin{aligned} \log_b b^x &= x \\ b^{\log_b x} &= x \\ \log_b(xy) &= \log_b x + \log_b y \\ \log_b(x^y) &= y \log_b x \\ \log_y x &= \frac{\log_z x}{\log_z y} \end{aligned}$$

It's worth proving each of these identities on your own, to get a better feel for how logarithms behave.

C.2 Induction

Computer scientists often use **mathematical induction** to prove things. Mathematical induction is a method for proving that something holds for all integers at least some number n_0. A proof by induction consists of two steps. The first is the **base case**. Here we show that the hypothesis is true for n_0. (This is usually trivial.) The second step is the **induction step**, wherein it is shown that, if the statement is true for all i between n_0 and n, then it is also true for n. These two steps imply that the statement is true for all $n \geq n_0$.

As an example, we will show that, for any integer $n \geq 1$, the sum of the first n positive odd numbers is n^2. As the base case, we observe that the first odd number is 1, which is 1^2. So the statement is true for $n = 1$. Now, for the inductive step, we assume it holds for all $i < n$. In particular, it holds for $n - 1$. The sum of the first n positive odds is the sum of the first

$n-1$ positive odds (which we assumed is $(n-1)^2$) and the nth odd number, $2n-1$. Since $(n-1)^2+2n-1=n^2-2n+1+2n-1=n^2$, we know that the sum of the first n positive odds is n^2. So our statement is proven.

Exercise C.1: Induction can often go awry. What's wrong with the following "proof" that all cows are the same color? For our base case, consider one cow. Obviously the cow is the same color as itself. Now say it holds for all $i<n$. Take away a cow named Alfalfa. We have $n-1$ cows left, and by the inductive hypothesis they all have the same color. Put Alfalfa back, and take away another cow, Bessie. Again, we know that they all must have the same color. This means that if we have n cows, they must all have the same color. Since there are only a finite number of cows in the world, they must all be identically colored.

C.3 Geometric series

You can also use induction to show facts about geometric series. A **geometric series** is a summation of numbers $1+r+r^2+\cdots+r^n$, where each number is a factor r times the last. You can show by induction that the sum of this series is $(1-r^{n+1})/(1-r)$.

An **infinite geometric series** extends the previous series out to infinity: $1+r+r^2+\cdots$. If $r<1$, then this infinite sum is $1/(1-r)$.

C.4 Recurrences

Recurrences turn up frequently when we use an inductive method. A recurrence is a function defined in terms of itself. Like induction (and recursion), every recurrence must have a base case.

We can represent the sum of the first n odd numbers using a recurrence as the following: Let $S(n)$ represent the sum of the first n odd numbers. Obviously we have $S(1)=1$; this is the base case. For other values in the parentheses, we have $S(n)=S(n-1)+(2n-1)$, since $2n-1$ is the nth odd number, and $S(n-1)$ is recursively defined as the sum of the first $n-1$ odd numbers.

We proved in Section C.2 that $S(n)$ has a simple **closed form** — that is, a form using no recurrences or extended summations. The closed form of $S(n)$ is n^2.

C.5 Graphs

A **graph** is a pair of sets V and E, denoted (V,E). The first set, V, is a set of **vertices**. This is an arbitrary set. The second set, E, is a set of **edges**. Each edge is a defined by a pair of vertices. One graph is a square, drawn in Figure C.1. In this graph, the set of vertices is

$$V=\{a,b,c,d\}\,,$$

and the set of edges is

$$E=\{(a,b),(b,c),(c,d),(d,a)\}\,.$$

Graphs are useful because they are very general structures that model a wide variety of problems. The edges can represent roads between cities, friendships between people, wires between computers, or many other things. This makes them interesting to computer scientists.

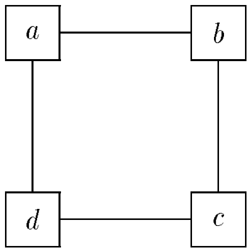


Figure C.1: A simple graph.

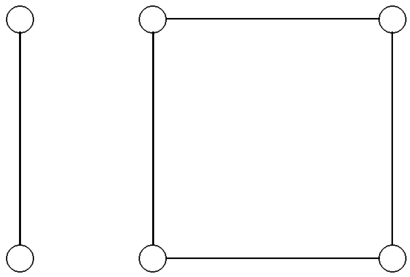


Figure C.2: A disconnected graph.

If they find how to model a problem as a graph problem (which often occurs), there's a good chance that somebody has already solved the graph problem.

An **undirected graph** is one in which the order of vertices in each pair is not significant (edge (a, d) is considered the same as edge (d, a)). In a **directed graph**, order matters. This book deals exclusively with undirected graphs.

Notice that an undirected graph of n vertices has at most $n(n-1)/2$ edges, since there are that many pairs of vertices, and the set of edges is a set of vertex pairs.

A graph defined by sets V' and E' is a **subgraph** of graph defined by sets V and E if $V' \subseteq V$ and $E' \subseteq E$. A single point ($V' = \{c\}$,$E' = \emptyset$) is a subgraph of the graph of Figure C.1. Note that if $V' = \{a\}$ and $E' = \{(a, b)\}$, we do not have a subgraph, even though the vertices and edges are subsets of the graph: This is not a graph itself. (The edge includes something not in V'.)

A **weighted graph** is a graph with a function w from edges to real numbers. Such a graph is often notated as (V, E, w). The **weight** of an edge is the value of the function for that edge. These weights can be understood as costs, capacities, or distances of the edge, according to what is natural for the problem.

A weighted graph defines a **distance** between vertices. A **path** between two vertices u and u' is a sequence of vertices $\langle u = v_0, v_1, v_2, \ldots v_{k-1}, v_k = u' \rangle$ such that, for every $1 \leq i \leq k$, (v_{i-1}, v_i) is an edge in the graph. The **length** of a path is the sum of the weights of the edges on it. There can be many paths between two vertices. The **distance** between u and v is the minimum length among all the paths between u and v.

We call a graph **connected** if, for every pair of vertices, there is a path in the graph connecting the vertices. The graph of Figure C.1 is connected; the graph of Figure C.2 is not.

C.6 Mathematical notation

This section defines most of the more unusual mathematical notation you'll find in in this book.

$\langle \ldots \rangle$ an ordered sequence of elements.

$\{ \ldots \}$ a set of elements.

$| \ldots |$ if a number is enclosed, the **absolute value** (x if x is positive, otherwise $-x$); if a set is enclosed, the number of items in the set.

∞ theoretically, infinity; in practice, a number that larger than anything that might occur.

$\sum_i f(i)$ the sum of $f(i)$ for each value of i. If i can be any number between 0 and n, this is $f(0) + f(1) + f(2) + \cdots + f(n)$.

$n!$ the **factorial** of a number; $n!$ is the product of the positive integers at most n ($3! = 3 \times 2 \times 1 = 6$)

e Euler's constant, an irrational number (like π) whose value is approximately 2.718281828.

$\log_b x$ the base-b logarithm; see Section C.1.

$O(\ldots)$ big-O notation; see Chapter 17.

$\Pr[\cdots]$ the probability that the event in brackets occurs.

$x_{(b)}$ the number x should be interpreted in base b. (If $b = 2$, then x is in binary; if $b = 10$, then x is in decimal.)

That's all the mathematical concepts we need. If you find a concept in the book not covered in this appendix, try referring to the index. If you don't find it listed, your instructor should be happy to help.

Appendix D

Exercise solutions

(The `#include` lines are omitted from the program listings in these solutions.)

Exercise 2.1: (page 6)

> **Problem** Search:
> **Input:** A list L of numbers and a number x.
> **Output:** `true` if x occurs in L, and `false` otherwise.

Exercise 2.2: (page 7) There are many plausible answers here. The following lists just a few of them.

Square-Root-Up Start with 1 and continue counting upward until you reach a number whose square is more than n. Output the number just before that one.

This algorithm's primary virtue is its simplicity. This simplicity makes it easy to understand and easy to program. It is not particularly quick, however.

Square-Root-Down Start at n and count downward until you reach a number whose square is less than n.

This is a minor variation on the previous algorithm. It is also very simple (although counting downwards is a little harder to understand), but it is much slower than before. Consider, for example, if n were $1,000,000$. With the previous algorithm, we count up from 1 to $1,000$. But with this algorithm we count down from $1,000,000$ to $1,000$, which is a much longer distance.

Square-Root-Half Always maintain a range where the square root might be. Initially, the range is $[1, n]$, but then we successively refine this range by repeatedly choosing the midpoint of the range (the first time, for example, the midpoint is $(1 + n)/2$), squaring it, and seeing whether the square is more or less than n. If it is more, then we know the square root is less than the midpoint and so we take our range to be the lower half of the range. It it is less, then the square root is above the midpoint, and so we take the range to be the upper half instead. We can stop once the range includes only one integer k; then we know the answer is either k or $k - 1$, and we can easily determine which it by seeing how k^2 compares to n. (By the way, this method of searching in a range by successively halving the range where it might be is called **binary search**.)

For example, if we sought the square root of 18, we would start with the range $[1, 18]$. The midpoint is 9.5, whose square is 90.25, so we restrict our range to $[1, 9.5]$. Now the midpoint is 5.25, whose square is 25.56; we restrict the range to $[1, 5.25]$. Now the midpoint is 3.13, and the square is 9.77; we restrict the range to $[3.13, 5.25]$. The midpoint is 4.19, and the square is 17.54; we restrict the range to $[4.19, 5.25]$. This range contains only one integer, 5. We test to see whether $5^2 > 18$; it is, so the answer is 4.

This algorithm is considerably more complicated than either of the first two algorithms, but it is also considerably faster. Consider again the case when n is $1,000,000$. The range is initially $999,999$ wide, but it halves each time we try something out. After 20 tries, then, the range is $999,999(1/2)^{20} \approx 0.95$ wide. At this point it contains at most one integer, and so the algorithm will stop after another try, for a total of 21 tries. This is much better than the $1,000$ done for the first algorithm. Unfortunately, the complexity of the algorithm does make it rather error-prone.

Square-Root-Factor If we knew n were a perfect square, then we could might take the prime factorization of n and then take every other prime factor. For example, to find the square root of 3600, we find the prime factorization of $2^4 \times 3^2 \times 5^2$. Every other factor of this is $2^2 \times 3 \times 5 = 60$, which is the square root of 3600.

This procedure has the fatal flaw that it doesn't accomplish what the problem states. It assumes that the input is a perfect square, and the problem statement did not include this restriction. But if the problem were for perfectly square inputs, it would be a reasonable algorithm. (In most cases it would be much faster than the first (and certainly the second) algorithm, and in some cases it may be somewhat faster than the third.)

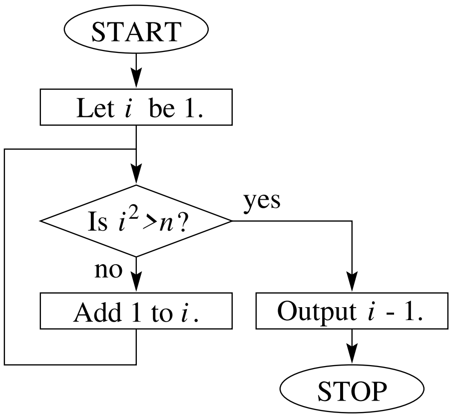
Exercise 3.1: (page 10)

Algorithm Square-Root-Up(n)
Let i hold 0.
while $i^2 \leq n$, **do:**
    Add 1 to i.
end of loop
output $i - 1$.

Algorithm Square-Root-Down(n)
Let i hold n.
while $i^2 > n$, **do:**
    Subtract 1 from i.
end of loop
output i.

Algorithm Square-Root-Half(n)
Initialize $range$ to $[1, n]$.
while $range$ contains more than 1 integer, **do:**
    Let k be the middle number in the range.
    if $k^2 \leq n$, **then** narrow $range$ to $[range$'s bottom, $k]$.
    else narrow $range$ to $[k, range$'s top$]$.
    end of if
end of loop
Let k be the first integer after $range$'s bottom.
if $k^2 \leq n$, **then output** k.
else output $k - 1$.
end of if

Exercise 3.2: (page 11) The following is a flowchart for Square-Root-Up.



Exercise 5.1: (page 21)

a.	`"3.4"`	`string`
b.	`0`	`int`
c.	`45.0`	`double`
d.	`"a"`	`string`
e.	`-1e10`	`double`

Exercise 5.2: (page 21)

a.	`name`	Yes; a `string` is probably the best choice.
b.	`num_points`	Yes; an `int` is probably the best choice.
c.	`letter`	Yes; a `char` is probably the best choice.
d.	`char`	No; the word `char` is reserved for other uses in C++.
e.	`#students`	No; the ‘`#`’ character cannot appear in names.
f.	`temperature`	Yes; a `double` is probably the best choice.
g.	`r2d2`	Yes; but it is a poor variable name: It doesn’t indicate content.
h.	`2i`	No; names cannot begin with digits.

Exercise 5.3: (page 23) `x` holds the value 98.6 and `k` holds the value 42.

a.	`k % 8`	value is 2
b.	`x - k * 2`	value is 14.6
c.	`k / 9`	value is 4 (remember integer division!)
d.	`-x / 2`	value is -49.3
e.	`2 k + 5`	invalid (must use `*` for multiplication)

Exercise 5.4: (page 24)

```
int main() {
    int year;
    cout << "It is January 1 of which year? ";
    cin >> year;
    int age = year - 1974;
    cout << "You are " << age << " years old." << endl;
}
```

Exercise 6.1: (page 28)

a. *true* always
b. *true* if x is 1 or 0
c. *true* if score is more than 90, or if bonus is nonzero and score is more than 80
d. *true* only if k is 0 (! has higher rank than ==)

Exercise 6.2: (page 29) If year is a multiple of 4 but not 100, it is a leap year. It is also a leap year if year is a multiple of 400. The following encodes both these cases.

```
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

Exercise 6.3: (page 29) This fragment lets ch hold a question mark if k holds 2. One minor fault is that ch doesn't hold anything meaningful otherwise, so we can't really use the value of ch later on.

```
char ch = ' '; // ch should holds something meaningful if k != 2.
if(k == 2) {  // condition must be in parentheses, and = must be doubled
    ch = '?'; // "?" is a string, but ch is a char. It should be '?'.
}
```

Exercise 6.4: (page 29) Notice how the following tests whether the denominator is 0 before performing the division. This is a feature of good programs: It is very careful with user input when the user might type something causing a run-time error. It is much better to print something meaningful in these cases.

```
int main() {
    int num; // numerator
    int den; // denominator
    cout << "What is the numerator? ";
    cin >> num;
    cout << "What is the denominator? ";
    cin >> den;

    if(den == 0) {
        cout << "Cannot divide by 0." << endl;
    } else if(num % den == 0) {
        cout << den << " divides " << num << "." << endl;
    } else {
        cout << den << " does not divide " << num << "." << endl;
    }
}
```

Exercise 6.5: (page 32) This fragment prints the powers of two from 1 to 16.

```
double total = 1;
i = 30;
while(i > 0) {
    cout << total << endl;
    total = 2 * total;
    i = i / 2;
}
```

Exercise 6.6: (page 33) This is probably meant to read in 30 numbers from the user and output the product of all the numbers. (Since it begins i at 30 and increases it for each iteration, the loop does not actually stop.)

```
for(i = 0; i < 30; i = i + 1) { // commas become semicolons, == becomes =
    cin >> num;                 // >> becomes <<, semicolon added
    product = product * num;    // semicolon added
}
```

Exercise 8.1: (page 41)

```
int removeDuplicates(vector<int> &arr, int arr_len) {
    int j = 1; // position in array with duplicates removed
    int i;     // position in initial array
    for(i = 1; i < arr_len; i = i + 1) {
        if(arr[i] != arr[i - 1]) { // this is not a duplicate
            arr[j] = arr[i]; // put it in array with duplicates removed
            j = j + 1;       // increase our position
        }
    }
    return j;
}
```

Exercise 8.2: (page 41) Probably the most intuitive way to do this is to read the numbers into an array and sort the numbers. This is an alternative answer working like Mode-Tally. Both are good approaches.

```
int main() {
    // create the tally boxes, initially 0.
    int tally[101];
    int i;
    for(i = 0; i <= 100; i++) {
        tally[i] = 0;
    }

    // how many scores are there?
    int num_scores;
    cout << "How many numbers? ";
    cin >> num_scores;

    // tally up the scores
    for(i = 0; i < num_scores; i = i + 1) {
        cout << "#" << (i + 1) << ": ";
        int x;
        cin >> x;
        tally[x] = tally[x] + 1;
    }

    // now find the median
    int total = 0; // number of marks found so far
    for(i = 0; i <= 100; i = i + 1) {
        total = total + tally[i];
        if(total > num_scores / 2) {
            cout << "median = " << i << endl;
            return 0;
        }
    }
}
```

Exercise 8.3: (page 42) It's important that you be careful with how you select the indices of the two letters you're comparing. (It's easy to be off by one here.)

```
int main() {
    string to_test;
    cout << "Which word? ";
    cin >> to_test;

    int i;
    for(i = 0; i < to_test.length() / 2; i = i + 1) {
        if(to_test[i] != to_test[to_test.length() - i - 1]) {
            cout << to_test << " is not a palindrome." << endl;
            return 0;
        }
    }
    cout << to_test << " is a palindrome." << endl;
    return 0;
}
```

Exercise 8.4: (page 43)

```
struct Date {
    int year;
    int month;
    int day;
};

struct LibraryBook {
    string name;
    int id;
    double price;
    Date due;
};
```

Exercise 10.1: (page 59)

```
void allSubsets(int depth, vector<int> &chosen, int n) {
    if(depth == n) { // we've made all choices; print this subset
        int i;
        for(i = 0; i < depth; i++) {
            if(chosen[i]) {
                cout << " " << (i + 1);
            }
        }
        cout << endl;
    } else {
        chosen[depth] = 0; // choose #depth to not be in the subset
        allSubsets(depth + 1, chosen, n);
        chosen[depth] = 1; // now choose #depth to be in the subset
        allSubsets(depth + 1, chosen, n);
    }
}

int main() {
    int n;
    cout << "Choose from how many? ";
    cin >> n;
    vector<int> subset(n);
    allSubsets(0, subset, n);
}
```

Exercise 16.1: (page 87) This comes from Lewis Carroll's poem "The Jabberwocky."

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

Exercise 17.1: (page 91) $1, \log_2 \log_2 n, \log_2 n, \sqrt{n}, n \log_2 n, n^2, 2^n, n!$

Exercise 17.2: (page 91)

a.	$3n \log_2 n + 5\sqrt{n}$	$= O(n \log_2 n)$
b.	$8n^2(4 \log_2 n + 3\sqrt{n})$	$= O(n^{5/2})$
c.	$n! + 8 \cdot 2^n + 5$	$= O(n!)$
d.	$\frac{8+3\log_2 n}{n}$	$= O(\frac{\log_2 n}{n})$

Exercise 17.3: (page 93)

`squareRootA()` $O(\sqrt{\mathtt{n}})$. Each time we go through the loop, we increase i by one. We start at zero and end at at most $1 + \sqrt{\mathtt{n}}$ so we go through $O(\sqrt{\mathtt{n}})$ iterations. Each iteration through O(1) time (Constant Rule) so the total is the product of these, $O(\sqrt{\mathtt{n}})$.

`squareRootB()` $O(\mathtt{n})$. The program begins `i` at `n` and subtracts 1. It stops by the time `i` becomes $\sqrt{\mathtt{n}} - 1$. So the computer goes through the loop at most $\mathtt{n} - \sqrt{\mathtt{n}} + 2$ iterations. The fastest-growing term here is the `n` term, so we go through $O(\mathtt{n})$ iterations. Each iteration takes O(1) time (Constant Rule), so by the Iteration Rule, the total time is $O(\mathtt{n})$.

`squareRootC()` $O(\log_2 \mathtt{n})$. The distance between `low` and `high` is initially `n`, and each time through the loop this distance at least halves. After halving the distance $\log_2 \mathtt{n}$ times, the range's size is at most 1, and so we have at most $\log_2 \mathtt{n}$ iterations. Each iteration involves no loops or function calls, so each takes O(1) time. By the Iteration Rule, the time required is therefore is $O(\log_2 \mathtt{n})$.

`squareRootD()` It's not the best answer, but one reasonable answer is $O(\sqrt{\mathtt{n}} \log_4 \mathtt{n})$. The outer loop will go through until `i` is the largest prime factor of `n`; $\sqrt{\mathtt{n}}$ is the largest possible prime factor of a perfect square, since every prime factor occurs twice in the factorization. So the outer loop has at most $\sqrt{\mathtt{n}}$ iterations. The inner loop can happen at most $\log_4 \mathtt{n}$ times, since each iteration divides `n` by $\mathtt{i}^2$, and `i` is always at least 2. Each iteration of the inner loop take $O(1)$ time, and the inner loop overall takes $O(\log_4 \mathtt{n})$ time. Thus each iteration of the outer loop takes $O(\log_4 \mathtt{n})$ time, for a total of $O(\sqrt{\mathtt{n}} \log_4 \mathtt{n})$.

A slightly better answer is $O(\sqrt{\mathtt{n}})$. This comes from observing that the inner loop can occur at most $\log_4 \mathtt{n}$ times during the entire execution of the function. Thus the total time is $O(\sqrt{\mathtt{n}} + \log_4 \mathtt{n}) = O(\sqrt{\mathtt{n}})$. Since this reasoning is a little subtle, you may reasonably not have found it.

Index