Computer Science
To Adam, Andrew, Brett, Robbie,

Henry, Iona, Rose, Peter,

and especially Linda

To Jackie, Alex, and Michael
## Contents

Preface ........................................... xiii

1—Elements of Programming .................... 1

1.1 Your First Program .......................... 2
1.2 Built-in Types of Data ......................... 14
1.3 Conditionals and Loops ...................... 50
1.4 Arrays ..................................... 90
1.5 Input and Output ............................. 126
1.6 Case Study: Random Web Surfer .......... 170

2—Functions and Modules ....................... 191

2.1 Defining Functions ......................... 192
2.2 Libraries and Clients ....................... 226
2.3 Recursion .................................. 262
2.4 Case Study: Percolation .................... 300

3—Object-Oriented Programming ................. 329

3.1 Using Data Types ........................... 330
3.2 Creating Data Types ......................... 382
3.3 Designing Data Types ....................... 428
3.4 Case Study: N-Body Simulation ............ 478

4—Algorithms and Data Structures ............... 493

4.1 Performance ................................ 494
4.2 Sorting and Searching ....................... 532
4.3 Stacks and Queues .......................... 566
4.4 Symbol Tables ............................... 624
4.5 Case Study: Small-World Phenomenon .... 670
### Theory of Computing

#### Formal Languages
- 5.1.1 RE recognition . . . . . . . . 729
- 5.1.2 Generalized RE pattern match 736
- 5.1.3 Universal virtual DFA . . . . . 743

#### Turing Machines
- 5.2.1 Virtual Turing machine tape . . 776
- 5.2.2 Universal virtual TM . . . . . 777

#### Universality

#### Computability

#### Intractability
- 5.5.1 SAT solver . . . . . . . . . . 855

### A Computing Machine

#### Representing Information
- 6.1.1 Number conversion . . . . . . 881
- 6.1.2 Floating-point components . . 893

#### TOY Machine
- 6.2.1 Your first TOY program . . . . 915
- 6.2.2 Conditionals and loops . . . . . 921
- 6.2.3 Self-modifying code. . . . . . 923

#### Machine-Language Programming
- 6.3.1 Calling a function . . . . . . . 933
- 6.3.2 Standard output . . . . . . . . . 935
- 6.3.3 Standard input . . . . . . . . . 937
- 6.3.4 Array processing . . . . . . . . 939
- 6.3.5 Linked structures . . . . . . . . 943

#### TOY Virtual Machine
- 6.4.1 TOY virtual machine . . . . . . 967
**Building a Computing Device**

**Boolean Logic**

**Basic Circuit Model**

**Combinational Circuits**
- Basic logic gates . . . . . . . . . . . . . . . . . . . 1014
- Selection multiplexer . . . . . . . . . . . . . . . . . 1024
- Decoder . . . . . . . . . . . . . . . . . . . . . . . . 1021
- Demultiplexer . . . . . . . . . . . . . . . . . . . . 1022
- Multiplexer . . . . . . . . . . . . . . . . . . . . . . 1023
- XOR . . . . . . . . . . . . . . . . . . . . . . . . . . 1024
- Majority . . . . . . . . . . . . . . . . . . . . . . . . 1025
- Odd parity . . . . . . . . . . . . . . . . . . . . . . . 1026
- Adder . . . . . . . . . . . . . . . . . . . . . . . . . . 1029
- ALU . . . . . . . . . . . . . . . . . . . . . . . . . . . 1033
- Bus multiplexer . . . . . . . . . . . . . . . . . . . . 1036

**Sequential Circuits**
- SR flip-flop . . . . . . . . . . . . . . . . . . . . . . . 1050
- Register bit . . . . . . . . . . . . . . . . . . . . . . . 1051
- Register . . . . . . . . . . . . . . . . . . . . . . . . . 1052
- Memory bit . . . . . . . . . . . . . . . . . . . . . . . 1056
- Memory . . . . . . . . . . . . . . . . . . . . . . . . . 1057
- Clock . . . . . . . . . . . . . . . . . . . . . . . . . . 1061

**Digital Devices**
- Program counter . . . . . . . . . . . . . . . . . . . . 1074
- Control . . . . . . . . . . . . . . . . . . . . . . . . . 1081
- CPU . . . . . . . . . . . . . . . . . . . . . . . . . . 1086
The basis for education in the last millennium was “reading, writing, and arithmetic”; now it is reading, writing, and computing. Learning to program is an essential part of the education of every student in the sciences and engineering. Beyond direct applications, it is the first step in understanding the nature of computer science’s undeniable impact on the modern world. This book aims to teach programming to those who need or want to learn it, in a scientific context.

Our primary goal is to empower students by supplying the experience and basic tools necessary to use computation effectively. Our approach is to teach students that composing a program is a natural, satisfying, and creative experience. We progressively introduce essential concepts, embrace classic applications from applied mathematics and the sciences to illustrate the concepts, and provide opportunities for students to write programs to solve engaging problems. We seek also to demystify computation for students and to build awareness about the substantial intellectual underpinnings of the field of computer science.

We use the Java programming language for all of the programs in this book. The first part of the book teaches basic skills for computational problem solving that are applicable in many modern computing environments, and it is a self-contained treatment intended for people with no previous experience in programming. It is about fundamental concepts in programming, not Java per se. The second part of the book demonstrates that there is much more to computer science than programming, but we do often use Java programs to help communicate the main ideas.

This book is an interdisciplinary approach to the traditional CS1 curriculum, in that we highlight the role of computing in other disciplines, from materials science to genomics to astrophysics to network systems. This approach reinforces for students the essential idea that mathematics, science, engineering, and computing are intertwined in the modern world. While it is a CS1 textbook designed for any first-year college student, the book also can be used for self-study.
Coverage The first part of the book is organized around three stages of learning to program: basic elements, functions, object-oriented programming, and algorithms. We provide the basic information that readers need to build confidence in composing programs at each level before moving to the next level. An essential feature of our approach is the use of example programs that solve intriguing problems, supported with exercises ranging from self-study drills to challenging problems that call for creative solutions.

Elements of programming include variables, assignment statements, built-in types of data, flow of control, arrays, and input/output, including graphics and sound.

Functions and modules are the students’ first exposure to modular programming. We build upon students’ familiarity with mathematical functions to introduce Java functions, and then consider the implications of programming with functions, including libraries of functions and recursion. We stress the fundamental idea of dividing a program into components that can be independently debugged, maintained, and reused.

Object-oriented programming is our introduction to data abstraction. We emphasize the concept of a data type and its implementation using Java’s class mechanism. We teach students how to use, create, and design data types. Modularity, encapsulation, and other modern programming paradigms are the central concepts of this stage.

The second part of the book introduces advanced topics in computer science: algorithms and data structures, theory of computing, and machine architecture.

Algorithms and data structures combine these modern programming paradigms with classic methods of organizing and processing data that remain effective for modern applications. We provide an introduction to classical algorithms for sorting and searching as well as fundamental data structures and their application, emphasizing the use of the scientific method to understand performance characteristics of implementations.

Theory of computing helps us address basic questions about computation, using simple abstract models of computers. Not only are the insights gained invaluable, but many of the ideas are also directly useful and relevant in practical computing applications.

Machine architecture provides a path to understanding what computation actually looks like in the real world—a link between the abstract machines of the theory of computing and the real computers that we use. Moreover, the study of
machine architecture provides a link to the past, as the microprocessors found in
today’s computers and mobile devices are not so different from the first computers
that were developed in the middle of the 20th century.

*Applications in science and engineering* are a key feature of the text. We mo-
tivate each programming concept that we address by examining its impact on
specific applications. We draw examples from applied mathematics, the physical
and biological sciences, and computer science itself, and include simulation of
physical systems, numerical methods, data visualization, sound synthesis, image
processing, financial simulation, and information technology. Specific examples
include a treatment in the first chapter of Markov chains for web page ranks and
case studies that address the percolation problem, *n*-body simulation, and the
small-world phenomenon. These applications are an integral part of the text. They
engage students in the material, illustrate the importance of the programming con-
cepts, and provide persuasive evidence of the critical role played by computation in
modern science and engineering.

*Historical context* is emphasized in the later chapters. The fascinating story of
the development and application of fundamental ideas about computation by Alan
Turing, John von Neumann, and many others is an important subtext.

Our primary goal is to teach the specific mechanisms and skills that are
needed to develop effective solutions to any programming problem. We work with
complete Java programs and encourage readers to use them. We focus on program-
mapping by individuals, not programming in the large.

**Use in the Curriculum**  This book is intended for a first-year college course
aimed at teaching computer science to novices in the context of scientific applica-
tions. When such a course is taught from this book, college student will learn to
program in a familiar context. Students completing a course based on this book
will be well prepared to apply their skills in later courses in their chosen major and
to recognize when further education in computer science might be beneficial.

Prospective computer science majors, in particular, can benefit from learning
to program in the context of scientific applications. A computer scientist needs the
same basic background in the scientific method and the same exposure to the role
of computation in science as does a biologist, an engineer, or a physicist.

Indeed, our interdisciplinary approach enables colleges and universities to
teach prospective computer science majors and prospective majors in other fields
in the *same* course. We cover the material prescribed by CS1, but our focus on
applications brings life to the concepts and motivates students to learn them. Our interdisciplinary approach exposes students to problems in many different disciplines, helping them to choose a major more wisely.

Whatever the specific mechanism, the use of this book is best positioned early in the curriculum. First, this positioning allows us to leverage familiar material in high school mathematics and science. Second, students who learn to program early in their college curriculum will then be able to use computers more effectively when moving on to courses in their specialty. Like reading and writing, programming is certain to be an essential skill for any scientist or engineer. Students who have grasped the concepts in this book will continually develop that skill throughout their lifetimes, reaping the benefits of exploiting computation to solve or to better understand the problems and projects that arise in their chosen field.

**Prerequisites**  This book is suitable for typical first-year college students. That is, we do not expect preparation beyond what is typically required for other entry-level science and mathematics courses.

*Mathematical maturity* is important. While we do not dwell on mathematical material, we do refer to the mathematics curriculum that students have taken in high school, including algebra, geometry, and trigonometry. Most students in our target audience automatically meet these requirements. Indeed, we take advantage of their familiarity with the basic curriculum to introduce basic programming concepts.

*Scientific curiosity* is also an essential ingredient. Science and engineering students bring with them a sense of fascination with the ability of scientific inquiry to help explain what goes on in nature. We leverage this predilection with examples of simple programs that speak volumes about the natural world. We do not assume any specific knowledge beyond that provided by typical high school courses in mathematics, physics, biology, or chemistry.

*Programming experience* is not necessary, but also is not harmful. Teaching programming is one of our primary goals, so we assume no prior programming experience. But composing a program to solve a new problem is a challenging intellectual task, so students who have written numerous programs in high school can benefit from taking an introductory programming course based on this book. The book can support teaching students with varying backgrounds because the applications appeal to both novices and experts alike.
Experience using a computer is not necessary, but also is not a problem. College students use computers regularly—for example, to communicate with friends and relatives, listen to music, process photos, and as part of many other activities. The realization that they can harness the power of their own computer in interesting and important ways is an exciting and lasting lesson.

In summary, virtually all college students are prepared to take a course based on this book as a part of their first-semester curriculum.

Goals  What can instructors of upper-level courses in science and engineering expect of students who have completed a course based on this book?

We cover the CS1 curriculum, but anyone who has taught an introductory programming course knows that expectations of instructors in later courses are typically high: each instructor expects all students to be familiar with the computing environment and approach that he or she wants to use. A physics professor might expect some students to design a program over the weekend to run a simulation; an engineering professor might expect other students to use a particular package to numerically solve differential equations; or a computer science professor might expect knowledge of the details of a particular programming environment. Is it realistic for a single entry-level course to meet such diverse expectations? Should there be a different introductory course for each set of students?

Colleges and universities have been wrestling with such questions since computers came into widespread use in the latter part of the 20th century. Our answer to them is found in this common introductory treatment of programming, which is analogous to commonly accepted introductory courses in mathematics, physics, biology, and chemistry. Computer Science strives to provide the basic preparation needed by all students in science and engineering, while sending the clear message that there is much more to understand about computer science than programming. Instructors teaching students who have studied from this book can expect that they will have the knowledge and experience necessary to enable those students to adapt to new computational environments and to effectively exploit computers in diverse applications.

What can students who have completed a course based on this book expect to accomplish in later courses?

Our message is that programming is not difficult to learn and that harnessing the power of the computer is rewarding. Students who master the material in this book are prepared to address computational challenges wherever they might
appear later in their careers. They learn that modern programming environments, such as the one provided by Java, help open the door to any computational problem they might encounter later, and they gain the confidence to learn, evaluate, and use other computational tools. Students interested in computer science will be well prepared to pursue that interest; students in science and engineering will be ready to integrate computation into their studies.

**Online lectures**  A complete set of studio-produced videos that can be used in conjunction with this text are available at

http://www.informit.com/title/9780134493831

As with traditional live lectures, the purpose of these videos is to inform and inspire, motivating students to study and learn from the text. Our experience is that student engagement with the material is significantly better than with live lectures because of the ability to play the lectures at a chosen speed and to replay and review the lectures at any time.

**Booksite**  An extensive amount of other information that supplements this text may be found on the web at

http://introcs.cs.princeton.edu/java

For economy, we refer to this site as the booksite throughout. It contains material for instructors, students, and casual readers of the book. We briefly describe this material here, though, as all web users know, it is best surveyed by browsing. With a few exceptions to support testing, the material is all publicly available.

One of the most important implications of the booksite is that it empowers instructors and students to use their own computers to teach and learn the material. Anyone with a computer and a browser can begin learning to program by following a few instructions on the booksite. The process is no more difficult than downloading a media player or a song. As with any website, our booksite is continually evolving. It is an essential resource for everyone who owns this book. In particular, the supplemental materials are critical to our goal of making computer science an integral component of the education of all scientists and engineers.

For instructors, the booksite contains information about teaching. This information is primarily organized around a teaching style that we have developed over the past decade, where we offer two lectures per week to a large audience, supplemented by two class sessions per week where students meet in small groups
with instructors or teaching assistants. The booksite has presentation slides for the lectures, which set the tone.

For teaching assistants, the booksite contains detailed problem sets and programming projects, which are based on exercises from the book but contain much more detail. Each programming assignment is intended to teach a relevant concept in the context of an interesting application while presenting an inviting and engaging challenge to each student. The progression of assignments embodies our approach to teaching programming. The booksite fully specifies all the assignments and provides detailed, structured information to help students complete them in the allotted time, including descriptions of suggested approaches and outlines for what should be taught in class sessions.

For students, the booksite contains quick access to much of the material in the book, including source code, plus extra material to encourage self-learning. Solutions are provided for many of the book’s exercises, including complete program code and test data. A wealth of information is associated with the programming assignments, including suggested approaches, checklists, FAQs, and test data.

For casual readers, the booksite is a resource for accessing all manner of extra information associated with the book’s content. All of the booksite content provides web links and other routes to pursue more information about the topic under consideration. There is far more information accessible than any individual could fully digest, but our goal is to provide enough to whet any reader’s appetite for more information about the book’s content.

Acknowledgments This project has been under development since 1992, so far too many people have contributed to its success for us to acknowledge them all here. Special thanks are due to Anne Rogers, for helping to start the ball rolling; to Dave Hanson, Andrew Appel, and Chris van Wyk, for their patience in explaining data abstraction; to Lisa Worthington and Donna Gabai, for being the first to truly relish the challenge of teaching this material to first-year students; and to Doug Clark for his patience as we learned about building Turing machines and circuits. We also gratefully acknowledge the efforts of /dev/126; the faculty, graduate students, and teaching staff who have dedicated themselves to teaching this material over the past 25 years here at Princeton University; and the thousands of undergraduates who have dedicated themselves to learning it.

Robert Sedgewick
Kevin Wayne

May 2016
This chapter centers on a construct that has as profound an impact on control flow as do conditionals and loops: the function, which allows us to transfer control back and forth between different pieces of code. Functions (which are known as static methods in Java) are important because they allow us to clearly separate tasks within a program and because they provide a general mechanism that enables us to reuse code.

We group functions together in modules, which we can compile independently. We use modules to break a computational task into subtasks of a reasonable size. You will learn in this chapter how to build modules of your own and how to use them, in a style of programming known as modular programming.

Some modules are developed with the primary intent of providing code that can be reused later by many other programs. We refer to such modules as libraries. In particular, we consider in this chapter libraries for generating random numbers, analyzing data, and providing input/output for arrays. Libraries vastly extend the set of operations that we use in our programs.

We pay special attention to functions that transfer control to themselves—a process known as recursion. At first, recursion may seem counterintuitive, but it allows us to develop simple programs that can address complex tasks that would otherwise be much more difficult to carry out.

Whenever you can clearly separate tasks within programs, you should do so. We repeat this mantra throughout this chapter, and end the chapter with a case study showing how a complex programming task can be handled by breaking it into smaller subtasks, then independently developing modules that interact with one another to address the subtasks.
2.1 Defining Functions

The Java construct for implementing a function is known as the static method. The modifier static distinguishes this kind of method from the kind discussed in Chapter 3—we will apply it consistently for now and discuss the difference then. You have actually been using static methods since the beginning of this book, from mathematical functions such as Math.abs() and Math.sqrt() to all of the methods in StdIn, StdOut, StdDraw, and StdAudio. Indeed, every Java program that you have written has a static method named main(). In this section, you will learn how to define your own static methods.

In mathematics, a function maps an input value of one type (the domain) to an output value of another type (the range). For example, the function \( f(x) = x^2 \) maps 2 to 4, 3 to 9, 4 to 16, and so forth. At first, we work with static methods that implement mathematical functions, because they are so familiar. Many standard mathematical functions are implemented in Java’s Math library, but scientists and engineers work with a broad variety of mathematical functions, which cannot all be included in the library. At the beginning of this section, you will learn how to implement such functions on your own.

Later, you will learn that we can do more with static methods than implement mathematical functions: static methods can have strings and other types as their range or domain, and they can produce side effects such as printing output. We also consider in this section how to use static methods to organize programs and thus to simplify complicated programming tasks.

Static methods support a key concept that will pervade your approach to programming from this point forward: whenever you can clearly separate tasks within programs, you should do so. We will be overemphasizing this point throughout this section and reinforcing it throughout this book. When you write an essay, you break it up into paragraphs; when you write a program, you will break it up into methods. Separating a larger task into smaller ones is much more important in programming than in writing, because it greatly facilitates debugging, maintenance, and reuse, which are all critical in developing good software.
2.1 Defining Functions

Static methods As you know from using Java’s Math library, the use of static methods is easy to understand. For example, when you write `Math.abs(a-b)` in a program, the effect is as if you were to replace that code with the return value that is produced by Java’s `Math.abs()` method when passed the expression `a-b` as an argument. This usage is so intuitive that we have hardly needed to comment on it. If you think about what the system has to do to create this effect, you will see that it involves changing a program’s control flow. The implications of being able to change the control flow in this way are as profound as doing so for conditionals and loops.

You can define static methods other than `main()` in a .java file by specifying a method signature, followed by a sequence of statements that constitute the method. We will consider the details shortly, but we begin with a simple example—Harmonic (Program 2.1.1)—that illustrates how methods affect control flow. It features a static method named `harmonic()` that takes an integer argument `n` and returns the `n`th harmonic number (see Program 1.3.5).

Program 2.1.1 is superior to our original implementation for computing harmonic numbers (Program 1.3.5) because it clearly separates the two primary tasks performed by the program: calculating harmonic numbers and interacting with the user. (For purposes of illustration, Program 2.1.1 takes several command-line arguments instead of just one.) Whenever you can clearly separate tasks within programs, you should do so.

Control flow. While Harmonic appeals to our familiarity with mathematical functions, we will examine it in detail so that you can think carefully about what a static method is and how it operates. Harmonic comprises two static methods: `harmonic()` and `main()`. Even though `harmonic()` appears first in the code, the first statement that Java executes is, as usual, the first statement in `main()`. The next few statements operate as usual, except that the code `harmonic(arg)`, which is known as a call on the static method `harmonic()`, causes a transfer of control to the first line of code in `harmonic()`, each time that it is encountered. Moreover, Java
Program 2.1.1 Harmonic numbers (revisited)

public class Harmonic
{
    public static double harmonic(int n)
    {
        double sum = 0.0;
        for (int i = 1; i <= n; i++)
            sum += 1.0/i;
        return sum;
    }
    
    public static void main(String[] args)
    {
        for (int i = 0; i < args.length; i++)
        {
            int arg = Integer.parseInt(args[i]);
            double value = harmonic(arg);
            StdOut.println(value);
        }
    }
}

This program defines two static methods, one named harmonic() that has integer argument n and computes the nth harmonic numbers (see Program 1.3.5) and one named main(), which tests harmonic() with integer arguments specified on the command line.

% java Harmonic 1 2 4
1.0
1.5
2.0833333333333333

% java Harmonic 10 100 1000 10000
2.9289682539682538
5.187377517639621
7.485470860550343
9.787606036044348

initializes the parameter variable n in harmonic() to the value of arg in main() at the time of the call. Then, Java executes the statements in harmonic() as usual, until it reaches a return statement, which transfers control back to the statement in main() containing the call on harmonic(). Moreover, the method call harmonic(arg) produces a value—the value specified by the return statement, which is the value of the variable sum in harmonic() at the time that the return
statement is executed. Java then assigns this \textit{return value} to the variable \textit{value}. The end result exactly matches our intuition: The first value assigned to \textit{value} and printed is 1.0—the value computed by code in \textit{harmonic()} when the parameter variable \textit{n} is initialized to 1. The next value assigned to \textit{value} and printed is 1.5—the value computed by \textit{harmonic()} when \textit{n} is initialized to 2. The same process is repeated for each command-line argument, transferring control back and forth between \textit{harmonic()} and \textit{main()}.

\textbf{Function-call trace.} One simple approach to following the control flow through function calls is to imagine that each function prints its name and argument value(s) when it is called and its return value just before returning, with indentation added on calls and subtracted on returns. The result enhances the process of tracing a program by printing the values of its variables, which we have been using since \textsc{Section} 1.2. The added indentation exposes the flow of the control, and helps us check that each function has the effect that we expect. Generally, adding calls on \texttt{StdOut.println()} to trace any program’s control flow in this way is a fine way to begin to understand what it is doing. If the return values match our expectations, we need not trace the function code in detail, saving us a substantial amount of work.

\textbf{For the rest of this chapter, your programming will center on creating and using static methods, so it is worthwhile to consider in more detail their basic properties. Following that, we will study several examples of function implementations and applications.}

\textbf{Terminology.} It is useful to draw a distinction between abstract concepts and Java mechanisms to implement them (the Java \texttt{if} statement implements the conditional, the \texttt{while} statement implements the loop, and so forth). Several concepts are rolled up in the idea of a mathematical function, and there are Java constructs corresponding to each, as summarized in the table at the top of the next page. While these formalisms have served mathematicians well for centuries (and have served programmers well for decades), we will refrain from considering in detail all of the implications of this correspondence and focus on those that will help you learn to program.
When we use a symbolic name in a formula that defines a mathematical function (such as $f(x) = 1 + x + x^2$), the symbol $x$ is a placeholder for some input value that will be substituted into the formula to determine the output value. In Java, we use a *parameter variable* as a symbolic placeholder and we refer to a particular input value where the function is to be evaluated as an *argument*.

**Static method definition.** The first line of a static method definition, known as the *signature*, gives a name to the method and to each parameter variable. It also specifies the type of each parameter variable and the return type of the method. The signature consists of the keyword public; the keyword static; the return type; the method name; and a sequence of zero or more parameter variable types and names, separated by commas and enclosed in parentheses. We will discuss the meaning of the public keyword in the next section and the meaning of the static keyword in *Chapter 3*. (Technically, the signature in Java includes only the method name and parameter types, but we leave that distinction for experts.) Following the signature is the *body* of the method, enclosed in curly braces. The body consists of the kinds of statements we discussed in *Chapter 1*. It also can contain a *return statement*, which transfers control back to the point where the static method was called and returns the result of the computation or return value. The body may declare *local variables*, which are variables that are available only inside the method in which they are declared.
2.1 Defining Functions

**Function calls.** As you have already seen, a static method call in Java is nothing more than the method name followed by its arguments, separated by commas and enclosed in parentheses, in precisely the same form as is customary for mathematical functions. As noted in Section 1.2, a method call is an expression, so you can use it to build up more complicated expressions. Similarly, an argument is an expression—Java evaluates the expression and passes the resulting value to the method. So, you can write code like `Math.exp(-x*x/2) / Math.sqrt(2*Math.PI)` and Java knows what you mean.

**Multiple arguments.** Like a mathematical function, a Java static method can take on more than one argument, and therefore can have more than one parameter variable. For example, the following static method computes the length of the hypotenuse of a right triangle with sides of length `a` and `b`:

```java
public static double hypotenuse(double a, double b) {
    return Math.sqrt(a*a + b*b);
}
```

Although the parameter variables are of the same type in this case, in general they can be of different types. The type and the name of each parameter variable are declared in the function signature, with the declarations for each variable separated by commas.

**Multiple methods.** You can define as many static methods as you want in a .java file. Each method has a body that consists of a sequence of statements enclosed in curly braces. These methods are independent and can appear in any order in the file. A static method can call any other static method in the same file or any static method in a Java library such as `Math`, as illustrated with this pair of methods:

```java
public static double square(double a) {
    return a*a;
}

public static double hypotenuse(double a, double b) {
    return Math.sqrt(square(a) + square(b));
}
```

Also, as we see in the next section, a static method can call static methods in other .java files (provided they are accessible to Java). In Section 2.3, we consider the ramifications of the idea that a static method can even call itself.
Overloading. Static methods with different signatures are different static methods. For example, we often want to define the same operation for values of different numeric types, as in the following static methods for computing absolute values:

```java
public static int abs(int x)
{
    if (x < 0) return -x;
    else       return  x;
}

public static double abs(double x)
{
    if (x < 0.0) return -x;
    else         return  x;
}
```

These are two different methods, but are sufficiently similar so as to justify using the same name (abs). Using the same name for two static methods whose signatures differ is known as overloading, and is a common practice in Java programming. For example, the Java Math library uses this approach to provide implementations of \texttt{Math.abs()}, \texttt{Math.min()}, and \texttt{Math.max()} for all primitive numeric types. Another common use of overloading is to define two different versions of a method: one that takes an argument and another that uses a default value for that argument.

Multiple return statements. You can put return statements in a method wherever you need them: control goes back to the calling program as soon as the first return statement is reached. This primality-testing function is an example of a function that is natural to define using multiple return statements:

```java
public static boolean isPrime(int n)
{
    if (n < 2) return false;
    for (int i = 2; i <= n/i; i++)
        if (n % i == 0) return false;
    return true;
}
```

Even though there may be multiple return statements, any static method returns a single value each time it is invoked: the value following the first return statement encountered. Some programmers insist on having only one return per method, but we are not so strict in this book.
### 2.1 Defining Functions

<table>
<thead>
<tr>
<th>Function Description</th>
<th>Java Code</th>
</tr>
</thead>
</table>
| absolute value of an int value | ```java
public static int abs(int x)
{
    if (x < 0) return -x;
    else return x;
}
``` |
| absolute value of a double value | ```java
public static double abs(double x)
{
    if (x < 0.0) return -x;
    else return x;
}
``` |
| primality test | ```java
public static boolean isPrime(int n)
{
    if (n < 2) return false;
    for (int i = 2; i <= n/i; i++)
        if (n % i == 0) return false;
    return true;
}
``` |
| hypotenuse of a right triangle | ```java
public static double hypotenuse(double a, double b)
{ return Math.sqrt(a*a + b*b); }
``` |
| harmonic number | ```java
public static double harmonic(int n)
{
    double sum = 0.0;
    for (int i = 1; i <= n; i++)
        sum += 1.0 / i;
    return sum;
}
``` |
| uniform random integer in [0, n) | ```java
public static int uniform(int n)
{ return (int) (Math.random() * n); }
``` |
| draw a triangle | ```java
public static void drawTriangle(double x0, double y0, double x1, double y1, double x2, double y2 )
{
    StdDraw.line(x0, y0, x1, y1);
    StdDraw.line(x1, y1, x2, y2);
    StdDraw.line(x2, y2, x0, y0);
}
``` |

*Typical code for implementing functions (static methods)*
Single return value. A Java method provides only one return value to the caller, of the type declared in the method signature. This policy is not as restrictive as it might seem because Java data types can contain more information than the value of a single primitive type. For example, you will see later in this section that you can use arrays as return values.

Scope. The scope of a variable is the part of the program that can refer to that variable by name. The general rule in Java is that the scope of the variables declared in a block of statements is limited to the statements in that block. In particular, the scope of a variable declared in a static method is limited to that method’s body. Therefore, you cannot refer to a variable in one static method that is declared in another. If the method includes smaller blocks—for example, the body of an if or a for statement—the scope of any variables declared in one of those blocks is limited to just the statements within that block. Indeed, it is common practice to use the same variable names in independent blocks of code. When we do so, we are declaring different independent variables. For example, we have been following this practice when we use an index i in two different for loops in the same program. A guiding principle when designing software is that each variable should be declared so that its scope is as small as possible. One of the important reasons that we use static methods is that they ease debugging by limiting variable scope.

```java
public class Harmonic {
    public static double harmonic(int n) {
        double sum = 0.0;
        for (int i = 1; i <= n; i++)
            sum += 1.0/i;
        return sum;
    }

    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            int arg = Integer.parseInt(args[i]);
            double value = harmonic(arg);
            StdOut.println(value);
    }
}
```

Scope of local and parameter variables
2.1 Defining Functions

**Side effects.** In mathematics, a function maps one or more input values to some output value. In computer programming, many functions fit that same model: they accept one or more arguments, and their only purpose is to return a value. A *pure function* is a function that, given the same arguments, always returns the same value, without producing any observable *side effects*, such as consuming input, producing output, or otherwise changing the state of the system. The functions `harmonic()`, `abs()`, `isPrime()`, and `hypotenuse()` are examples of pure functions.

However, in computer programming it is also useful to define functions that do produce side effects. In fact, we often define functions whose *only* purpose is to produce side effects. In Java, a static method may use the keyword `void` as its return type, to indicate that it has no return value. An explicit `return` is not necessary in a `void` static method: control returns to the caller after Java executes the method’s last statement.

For example, the static method `StdOut.println()` has the side effect of printing the given argument to standard output (and has no return value). Similarly, the following static method has the side effect of drawing a triangle to standard drawing (and has no specified return value):

```java
public static void drawTriangle(double x0, double y0,
                                double x1, double y1,
                                double x2, double y2)
{
    StdDraw.line(x0, y0, x1, y1);
    StdDraw.line(x1, y1, x2, y2);
    StdDraw.line(x2, y2, x0, y0);
}
```

It is generally poor style to write a static method that both produces side effects and returns a value. One notable exception arises in functions that read input. For example, `StdIn.readInt()` both returns a value (an integer) and produces a side effect (consuming one integer from standard input). In this book, we use `void` static methods for two primary purposes:

- For I/O, using `StdIn`, `StdOut`, `StdDraw`, and `StdAudio`
- To manipulate the contents of arrays

You have been using `void` static methods for output since `main()` in `HelloWorld`, and we will discuss their use with arrays later in this section. It is possible in Java to write methods that have other side effects, but we will avoid doing so until Chapter 3, where we do so in a specific manner supported by Java.
Implementing mathematical functions  Why not just use the methods that are defined within Java, such as Math.sqrt()? The answer to this question is that we do use such implementations when they are present. Unfortunately, there are an unlimited number of mathematical functions that we may wish to use and only a small set of functions in the library. When you encounter a mathematical function that is not in the library, you need to implement a corresponding static method.

As an example, we consider the kind of code required for a familiar and important application that is of interest to many high school and college students in the United States. In a recent year, more than 1 million students took a standard college entrance examination. Scores range from 400 (lowest) to 1600 (highest) on the multiple-choice parts of the test. These scores play a role in making important decisions: for example, student athletes are required to have a score of at least 820, and the minimum eligibility requirement for certain academic scholarships is 1500. What percentage of test takers are ineligible for athletics? What percentage are eligible for the scholarships?

Two functions from statistics enable us to compute accurate answers to these questions. The Gaussian (normal) probability density function is characterized by the familiar bell-shaped curve and defined by the formula \( \phi(x) = e^{-x^2/2}/\sqrt{2\pi} \). The Gaussian cumulative distribution function \( \Phi(z) \) is defined to be the area under the curve defined by \( \phi(x) \) above the x-axis and to the left of the vertical line \( x = z \). These functions play an important role in science, engineering, and finance because they arise as accurate models throughout the natural world and because they are essential in understanding experimental error.

In particular, these functions are known to accurately describe the distribution of test scores in our example, as a function of the mean (average value of the scores) and the standard deviation (square root of the average of the sum of the squares of the differences between each score and the mean), which are published each year. Given the mean \( \mu \) and the standard deviation \( \sigma \) of the test scores, the percentage of students with scores less than a given value \( z \) is closely approximated by the function \( \Phi((z - \mu)/\sigma) \). Static methods to calculate \( \phi \) and \( \Phi \) are not available in Java’s Math library, so we need to develop our own implementations.
2.1 Defining Functions

Program 2.1.2 Gaussian functions

```java
public class Gaussian
{
   // Implement Gaussian (normal) distribution functions.
   public static double pdf(double x)
   {
      return Math.exp(-x*x/2) / Math.sqrt(2*Math.PI);
   }

   public static double cdf(double z)
   {
      if (z < -8.0) return 0.0;
      if (z >  8.0) return 1.0;
      double sum  = 0.0;
      double term = z;
      for (int i = 3; sum != sum + term; i += 2)
      {
         sum  = sum + term;
         term = term * z * z / i;
      }
      return 0.5 + pdf(z) * sum;
   }

   public static void main(String[] args)
   {
      double z     = Double.parseDouble(args[0]);
      double mu    = Double.parseDouble(args[1]);
      double sigma = Double.parseDouble(args[2]);
      StdOut.printf("%.3f\n", cdf((z - mu) / sigma));
   }
}
```

This code implements the Gaussian probability density function (pdf) and Gaussian cumulative distribution function (cdf), which are not implemented in Java’s Math library. The pdf() implementation follows directly from its definition, and the cdf() implementation uses a Taylor series and also calls pdf() (see accompanying text and Exercise 1.3.38).

% java Gaussian 820 1019 209
0.171
% java Gaussian 1500 1019 209
0.989
% java Gaussian 1500 1025 231
0.980
Functions and Modules

Closed form. In the simplest situation, we have a closed-form mathematical formula defining our function in terms of functions that are implemented in the library. This situation is the case for $\phi$—the Java Math library includes methods to compute the exponential and the square root functions (and a constant value for $\pi$), so a static method `pdf()` corresponding to the mathematical definition is easy to implement (see Program 2.1.2).

No closed form. Otherwise, we may need a more complicated algorithm to compute function values. This situation is the case for $\Phi$—no closed-form expression exists for this function. Such algorithms sometimes follow immediately from Taylor series approximations, but developing reliably accurate implementations of mathematical functions is an art that needs to be addressed carefully, taking advantage of the knowledge built up in mathematics over the past several centuries. Many different approaches have been studied for evaluating $\Phi$. For example, a Taylor series approximation to the ratio of $\Phi$ and $\phi$ turns out to be an effective basis for evaluating the function:

$$\Phi(z) = 1/2 + \phi(z) \left( z + z^3/3 + z^5/(3\cdot5) + z^7/(3\cdot5\cdot7) + \ldots \right)$$

This formula readily translates to the Java code for the static method `cdf()` in Program 2.1.2. For small (respectively large) $z$, the value is extremely close to 0 (respectively 1), so the code directly returns 0 (respectively 1); otherwise, it uses the Taylor series to add terms until the sum converges.

Running Gaussian with the appropriate arguments on the command line tells us that about 17% of the test takers were ineligible for athletics and that only about 1% qualified for the scholarship. In a year when the mean was 1025 and the standard deviation 231, about 2% qualified for the scholarship.

Computing with mathematical functions of all kinds has always played a central role in science and engineering. In a great many applications, the functions that you need are expressed in terms of the functions in Java’s Math library, as we have just seen with `pdf()`, or in terms of Taylor series approximations that are easy to compute, as we have just seen with `cdf()`. Indeed, support for such computations has played a central role throughout the evolution of computing systems and programming languages. You will find many examples on the booksite and throughout this book.
2.1 Defining Functions

Using static methods to organize code  Beyond evaluating mathematical functions, the process of calculating an output value on the basis of an input value is important as a general technique for organizing control flow in any computation. Doing so is a simple example of an extremely important principle that is a prime guiding force for any good programmer: whenever you can clearly separate tasks within programs, you should do so.

Functions are natural and universal for expressing computational tasks. Indeed, the “bird’s-eye view” of a Java program that we began with in Section 1.1 was equivalent to a function: we began by thinking of a Java program as a function that transforms command-line arguments into an output string. This view expresses itself at many different levels of computation. In particular, it is generally the case that a long program is more naturally expressed in terms of functions instead of as a sequence of Java assignment, conditional, and loop statements. With the ability to define functions, we can better organize our programs by defining functions within them when appropriate.

For example, Coupon (Program 2.1.3) is a version of CouponCollector (Program 1.4.2) that better separates the individual components of the computation. If you study Program 1.4.2, you will identify three separate tasks:

• Given \( n \), compute a random coupon value.
• Given \( n \), do the coupon collection experiment.
• Get \( n \) from the command line, and then compute and print the result.

Coupon rearranges the code in CouponCollector to reflect the reality that these three functions underlie the computation. With this organization, we could change getCoupon() (for example, we might want to draw the random numbers from a different distribution) or main() (for example, we might want to take multiple inputs or run multiple experiments) without worrying about the effect of any changes in collectCoupons().

Using static methods isolates the implementation of each component of the collection experiment from others, or encapsulates them. Typically, programs have many independent components, which magnifies the benefits of separating them into different static methods. We will discuss these benefits in further detail after we have seen several other examples, but you certainly can appreciate that it is better to express a computation in a program by breaking it up into functions, just as it is better to express an idea in an essay by breaking it up into paragraphs. Whenever you can clearly separate tasks within programs, you should do so.
Program 2.1.3  Coupon collector (revisited)

```java
public class Coupon {

    public static int getCoupon(int n) {
        // Return a random integer between 0 and n-1.
        return (int) (Math.random() * n);
    }

    public static int collectCoupons(int n) {
        // Collect coupons until getting one of each value
        // and return the number of coupons collected.
        boolean[] isCollected = new boolean[n];
        int count = 0, distinct = 0;
        while (distinct < n) {
            int r = getCoupon(n);
            count++;
            if (!isCollected[r])
                distinct++;
            isCollected[r] = true;
        }
        return count;
    }

    public static void main(String[] args) {
        // Collect n different coupons.
        int n = Integer.parseInt(args[0]);
        int count = collectCoupons(n);
        StdOut.println(count);
    }
}
```

This version of Program 1.4.2 illustrates the style of encapsulating computations in static methods. This code has the same effect as CouponCollector, but better separates the code into its three constituent pieces: generating a random integer between 0 and n-1, running a coupon collection experiment, and managing the I/O.

% java Coupon 1000 6522
% java Coupon 1000 6481
% java Coupon 10000 105798
% java Coupon 1000000 12783771
2.1 Defining Functions

Passing arguments and returning values  Next, we examine the specifics of Java’s mechanisms for passing arguments to and returning values from functions. These mechanisms are conceptually very simple, but it is worthwhile to take the time to understand them fully, as the effects are actually profound. Understanding argument-passing and return-value mechanisms is key to learning any new programming language.

Pass by value. You can use parameter variables anywhere in the code in the body of the function in the same way you use local variables. The only difference between a parameter variable and a local variable is that Java evaluates the argument provided by the calling code and initializes the parameter variable with the resulting value. This approach is known as pass by value. The method works with the value of its arguments, not the arguments themselves. One consequence of this approach is that changing the value of a parameter variable within a static method has no effect on the calling code. (For clarity, we do not change parameter variables in the code in this book.) An alternative approach known as pass by reference, where the method works directly with the calling code’s arguments, is favored in some programming environments.

A static method can take an array as an argument or return an array to the caller. This capability is a special case of Java’s object orientation, which is the subject of Chapter 3. We consider it in the present context because the basic mechanisms are easy to understand and to use, leading us to compact solutions to a number of problems that naturally arise when we use arrays to help us process large amounts of data.

Arrays as arguments. When a static method takes an array as an argument, it implements a function that operates on an arbitrary number of values of the same type. For example, the following static method computes the mean (average) of an array of double values:

```java
public static double mean(double[] a) {
    double sum = 0.0;
    for (int i = 0; i < a.length; i++)
        sum += a[i];
    return sum / a.length;
}
```
We have been using arrays as arguments since our first program. The code

```java
public static void main(String[] args)
```

defines `main()` as a static method that takes an array of strings as an argument and returns nothing. By convention, the Java system collects the strings that you type after the program name in the `java` command into an array and calls `main()` with that array as argument. (Most programmers use the name `args` for the parameter variable, even though any name at all would do.) Within `main()`, we can manipulate that array just like any other array.

**Side effects with arrays.** It is often the case that the purpose of a static method that takes an array as argument is to produce a side effect (change values of array elements). A prototypical example of such a method is one that exchanges the values at two given indices in a given array. We can adapt the code that we examined at the beginning of SECTION 1.4:

```java
public static void exchange(String[] a, int i, int j)
{
    String temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

This implementation stems naturally from the Java array representation. The parameter variable in `exchange()` is a reference to the array, not a copy of the array values: when you pass an array as an argument to a method, the method has an opportunity to reassign values to the elements in that array. A second prototypical example of a static method that takes an array argument and produces side effects is one that randomly shuffles the values in the array, using this version of the algorithm that we examined in SECTION 1.4 (and the `exchange()` and `uniform()` methods considered earlier in this section):

```java
public static void shuffle(String[] a)
{
    int n = a.length;
    for (int i = 0; i < n; i++)
        exchange(a, i, i + uniform(n-i));
}
```
2.1 Defining Functions

find the maximum of the array values

```java
public static double max(double[] a) {
    double max = Double.NEGATIVE_INFINITY;
    for (int i = 0; i < a.length; i++)
        if (a[i] > max) max = a[i];
    return max;
}
```

dot product

```java
public static double dot(double[] a, double[] b) {
    double sum = 0.0;
    for (int i = 0; i < a.length; i++)
        sum += a[i] * b[i];
    return sum;
}
```

exchange the values of two elements in an array

```java
public static void exchange(String[] a, int i, int j) {
    String temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

print a one-dimensional array (and its length)

```java
public static void print(double[] a) {
    StdOut.println(a.length);
    for (int i = 0; i < a.length; i++)
        StdOut.println(a[i]);
}
```

read a 2D array of double values (with dimensions) in row-major order

```java
public static double[][] readDouble2D() {
    int m = StdIn.readInt();
    int n = StdIn.readInt();
    double[][] a = new double[m][n];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            a[i][j] = StdIn.readDouble();
    return a;
}
```

Typical code for implementing functions with array arguments or return values
Similarly, we will consider in Section 4.2 methods that sort an array (rearrange its values so that they are in order). All of these examples highlight the basic fact that the mechanism for passing arrays in Java is call by value with respect to the array reference but call by reference with respect to the array elements. Unlike primitive-type arguments, the changes that a method makes to the elements of an array are reflected in the client program. A method that takes an array as its argument cannot change the array itself—the memory location, length, and type of the array are the same as they were when the array was created—but a method can assign different values to the elements in the array.

**Arrays as return values.** A method that sorts, shuffles, or otherwise modifies an array taken as an argument does not have to return a reference to that array, because it is changing the elements of a client array, not a copy. But there are many situations where it is useful for a static method to provide an array as a return value. Chief among these are static methods that create arrays for the purpose of returning multiple values of the same type to a client. For example, the following static method creates and returns an array of the kind used by `StdAudio` (see Program 1.5.7): it contains values sampled from a sine wave of a given frequency (in hertz) and duration (in seconds), sampled at the standard 44,100 samples per second.

```java
public static double[] tone(double hz, double t)
{
    int SAMPLING_RATE = 44100;
    int n = (int) (SAMPLING_RATE * t);
    double[] a = new double[n+1];
    for (int i = 0; i <= n; i++)
        a[i] = Math.sin(2 * Math.PI * i * hz / SAMPLING_RATE);
    return a;
}
```

In this code, the length of the array returned depends on the duration: if the given duration is `t`, the length of the array is about `44100*t`. With static methods like this one, we can write code that treats a sound wave as a single entity (an array containing sampled values), as we will see next in Program 2.1.4.
Example: superposition of sound waves As discussed in Section 1.5, the simple audio model that we studied there needs to be embellished to create sound that resembles the sound produced by a musical instrument. Many different embellishments are possible; with static methods we can systematically apply them to produce sound waves that are far more complicated than the simple sine waves that we produced in Section 1.5. As an illustration of the effective use of static methods to solve an interesting computational problem, we consider a program that has essentially the same functionality as PlayThatTune (Program 1.5.7), but adds harmonic tones one octave above and one octave below each note to produce a more realistic sound.

Chords and harmonics. Notes like concert A have a pure sound that is not very musical, because the sounds that you are accustomed to hearing have many other components. The sound from the guitar string echoes off the wooden part of the instrument, the walls of the room that you are in, and so forth. You may think of such effects as modifying the basic sine wave. For example, most musical instruments produce harmonics (the same note in different octaves and not as loud), or you might play chords (multiple notes at the same time). To combine multiple sounds, we use superposition: simply add the waves together and rescale to make sure that all values stay between $-1$ and $+1$. As it turns out, when we superpose sine waves of different frequencies in this way, we can get arbitrarily complicated waves. Indeed, one of the triumphs of 19th-century mathematics was the development of the idea that any smooth periodic function can be expressed as a sum of sine and cosine waves, known as a Fourier series. This mathematical idea corresponds to the notion that we can create a large range of sounds with musical instruments or our vocal cords and that all sound consists of a composition of various oscillating curves. Any sound corresponds to a curve and any curve corresponds to a sound, and we can create arbitrarily complex curves with superposition.
**Weighted superposition.** Since we represent sound waves by arrays of numbers that represent their values at the same sample points, superposition is simple to implement: we add together the values at each sample point to produce the combined result and then rescale. For greater control, we specify a relative weight for each of the two waves to be added, with the property that the weights are positive and sum to 1. For example, if we want the first sound to have three times the effect of the second, we would assign the first a weight of 0.75 and the second a weight of 0.25. Now, if one wave is in an array \( a[] \) with relative weight \( a_{wt} \) and the other is in an array \( b[] \) with relative weight \( b_{wt} \), we compute their weighted sum with the following code:

```java
double[] c = new double[a.length];
for (int i = 0; i < a.length; i++)
    c[i] = a[i]*a_{wt} + b[i]*b_{wt};
```

The conditions that the weights are positive and sum to 1 ensure that this operation preserves our convention of keeping the values of all of our waves between \(-1\) and \(+1\).

\[
\text{lo} = \text{tone}(220, 1.0/220.0) \\
\text{hi} = \text{tone}(880, 1.0/220.0) \\
\text{harmonics} = \text{superpose}(\text{lo}, \text{hi}, 0.5, 0.5)
\]

\[
\begin{align*}
\text{harmonics}[44] &= 0.5\times\text{lo}[44] + 0.5\times\text{hi}[44] \\
&= 0.5\times0.982 + 0.5\times(-0.693) \\
&= 0.144
\end{align*}
\]

\[
\begin{align*}
\text{concertA} &= \text{tone}(440, 1.0/220.0) \\
\text{superpose}(\text{harmonics}, \text{concertA}, 0.5, 0.5) \\
&= 0.5\times\text{harmonics}[44] + 0.5\times\text{concertA}[44]) \\
&= 0.5\times0.144 + 0.5\times0.374 \\
&= 0.259
\end{align*}
\]

*Adding harmonics to concert A (1/220 second at 44,100 samples/second)*
2.1 Defining Functions

Program 2.1.4  Play that tune (revisited)

public class PlayThatTuneDeluxe
{
    public static double[] superpose(double[] a, double[] b,
        double awt, double bwt)
    {
        // Weighted superposition of a and b.
        double[] c = new double[a.length];
        for (int i = 0; i < a.length; i++)
            c[i] = a[i]*awt + b[i]*bwt;
        return c;
    }
    public static double[] tone(double hz, double t)
    {
        /* see text */
    }
    public static double[] note(int pitch, double t)
    {
        // Play note of given pitch, with harmonics.
        double hz = 440.0 * Math.pow(2, pitch / 12.0);
        double[] a  = tone(hz, t);
        double[] hi = tone(2*hz, t);
        double[] lo = tone(hz/2, t);
        double[] h  = superpose(hi, lo, 0.5, 0.5);
        return superpose(a, h, 0.5, 0.5);
    }
    public static void main(String[] args)
    {
        // Read and play a tune, with harmonics.
        while (!StdIn.isEmpty())
            {  // Read and play a note, with harmonics.
                int pitch = StdIn.readInt();
                double duration = StdIn.readDouble();
                double[] a = note(pitch, duration);
                StdAudio.play(a);
            }
    }
}

This code embellishes the sounds produced by Program 1.5.7 by using static methods to create harmonics, which results in a more realistic sound than the pure tone.

% java PlayThatTuneDeluxe < elise.txt

% more elise.txt
7 0.25
6 0.25
7 0.25
6 0.25
...
Program 2.1.4 is an implementation that applies these concepts to produce a more realistic sound than that produced by Program 1.5.7. To do so, it makes use of functions to divide the computation into four parts:

- Given a frequency and duration, create a pure tone.
- Given two sound waves and relative weights, superpose them.
- Given a pitch and duration, create a note with harmonics.
- Read and play a sequence of pitch/duration pairs from standard input.

These tasks are each amenable to implementation as a function, with all of the functions then depending on one another. Each function is well defined and straightforward to implement. All of them (and StdAudio) represent sound as a sequence of floating-point numbers kept in an array, corresponding to sampling a sound wave at 44,100 samples per second.

Up to this point, the use of functions has been somewhat of a notational convenience. For example, the control flow in Program 2.1.1–2.1.3 is simple—each function is called in just one place in the code. By contrast, PlayThatTuneDeluxe (Program 2.1.4) is a convincing example of the effectiveness of defining functions to organize a computation because the functions are each called multiple times. For example, the function note() calls the function tone() three times and the function sum() twice. Without functions methods, we would need multiple copies of the code in

```java
public class PlayThatTuneDeluxe {
    public static double[] superpose(double[] a, double[] b, double awt, double bwt) {
        double[] c = new double[a.length];
        for (int i = 0; i < a.length; i++)
            c[i] = a[i]*awt + b[i]*bwt;
        return c;
    }

    public static double[] tone(double hz, double t) {
        int RATE = 44100;
        int n = (int) (RATE * t);
        double[] a = new double[n+1];
        for (int i = 0; i <= n; i++)
            a[i] = Math.sin(2 * Math.PI * i * hz / RATE);
        return a;
    }

    public static double[] note(int pitch, double t) {
        double hz = 440.0 * Math.pow(2, pitch / 12.0);
        double[] a = tone(hz, t);
        double[] hi = tone(2*hz, t);
        double[] lo = tone(hz/2, t);
        double[] h = superpose(hi, lo, .5, .5);
        return superpose(a, h, .5, .5);
    }

    public static void main(String[] args) {
        while (!StdIn.isEmpty()) {
            int pitch = StdIn.readInt();
            double duration = StdIn.readDouble();
            double[] a = note(pitch, duration);
            StdAudio.play(a);
        }
    }
}
```

Flow of control among several static methods
2.1 Defining Functions

tone() and sum(); with functions, we can deal directly with concepts close to the application. Like loops, functions have a simple but profound effect: one sequence of statements (those in the method definition) is executed multiple times during the execution of our program—once for each time the function is called in the control flow in main().

FUNCTIONS (STATIC METHODS) ARE IMPORTANT BECAUSE they give us the ability to extend the Java language within a program. Having implemented and debugged functions such as harmonic(), pdf(), cdf(), mean(), abs(), exchange(), shuffle(), isPrime(), uniform(), superpose(), note(), and tone(), we can use them almost as if they were built into Java. The flexibility to do so opens up a whole new world of programming. Before, you were safe in thinking about a Java program as a sequence of statements. Now you need to think of a Java program as a set of static methods that can call one another. The statement-to-statement control flow to which you have been accustomed is still present within static methods, but programs have a higher-level control flow defined by static method calls and returns. This ability enables you to think in terms of operations called for by the application, not just the simple arithmetic operations on primitive types that are built into Java.

Whenever you can clearly separate tasks within programs, you should do so. The examples in this section (and the programs throughout the rest of the book) clearly illustrate the benefits of adhering to this maxim. With static methods, we can

• Divide a long sequence of statements into independent parts.
• Reuse code without having to copy it.
• Work with higher-level concepts (such as sound waves).

This produces code that is easier to understand, maintain, and debug than a long program composed solely of Java assignment, conditional, and loop statements. In the next section, we discuss the idea of using static methods defined in other programs, which again takes us to another level of programming.
Q. What happens if I leave out the keyword `static` when defining a static method?
A. As usual, the best way to answer a question like this is to try it yourself and see what happens. Here is the result of omitting the `static` modifier from `harmonic()` in `Harmonic`:

```
Harmonic.java:15: error: non-static method harmonic(int)
cannot be referenced from a static context
double value = harmonic(arg);
    ^
1 error
```

Non-static methods are different from static methods. You will learn about the former in Chapter 3.

Q. What happens if I write code after a `return` statement?
A. Once a `return` statement is reached, control immediately returns to the caller, so any code after a `return` statement is useless. Java identifies this situation as a compile-time error, reporting unreachable code.

Q. What happens if I do not include a `return` statement?
A. There is no problem, if the return type is `void`. In this case, control will return to the caller after the last statement. When the return type is not `void`, Java will report a missing `return` statement compile-time error if there is any path through the code that does not end in a `return` statement.

Q. Why do I need to use the return type `void`? Why not just omit the return type?
A. Java requires it; we have to include it. Second-guessing a decision made by a programming-language designer is the first step on the road to becoming one.

Q. Can I return from a `void` function by using `return`? If so, which return value should I use?
A. Yes. Use the statement `return;` with no return value.
Q. This issue with side effects and arrays passed as arguments is confusing. Is it really all that important?

A. Yes. Properly controlling side effects is one of a programmer’s most important tasks in large systems. Taking the time to be sure that you understand the difference between passing a value (when arguments are of a primitive type) and passing a reference (when arguments are arrays) will certainly be worthwhile. The very same mechanism is used for all other types of data, as you will learn in Chapter 3.

Q. So why not just eliminate the possibility of side effects by making all arguments pass by value, including arrays?

A. Think of a huge array with, say, millions of elements. Does it make sense to copy all of those values for a static method that is going to exchange just two of them? For this reason, most programming languages support passing an array to a function without creating a copy of the array elements—Matlab is a notable exception.

Q. In which order does Java evaluate method calls?

A. Regardless of operator precedence or associativity, Java evaluates subexpressions (including method calls) and argument lists from left to right. For example, when evaluating the expression

\[
f_1() + f_2() \times f_3(f_4(), f_5())
\]

Java calls the methods in the order \(f_1(), f_2(), f_4(), f_5(),\) and \(f_3().\) This is most relevant for methods that produce side effects. As a matter of style, we avoid writing code that depends on the order of evaluation.
2.1.1 Write a static method `max3()` that takes three `int` arguments and returns the value of the largest one. Add an overloaded function that does the same thing with three `double` values.

2.1.2 Write a static method `odd()` that takes three `boolean` arguments and returns `true` if an odd number of the argument values are `true`, and `false` otherwise.

2.1.3 Write a static method `majority()` that takes three `boolean` arguments and returns `true` if at least two of the argument values are `true`, and `false` otherwise. Do not use an `if` statement.

2.1.4 Write a static method `eq()` that takes two `int` arrays as arguments and returns `true` if the arrays have the same length and all corresponding pairs of elements are equal, and `false` otherwise.

2.1.5 Write a static method `areTriangular()` that takes three `double` arguments and returns `true` if they could be the sides of a triangle (none of them is greater than or equal to the sum of the other two). See Exercise 1.2.15.

2.1.6 Write a static method `sigmoid()` that takes a `double` argument `x` and returns the `double` value obtained from the formula $1/(1 + e^{-x})$.

2.1.7 Write a static method `sqrt()` that takes a `double` argument and returns the square root of that number. Use Newton’s method (see Program 1.3.6) to compute the result.

2.1.8 Give the function-call trace for `java Harmonic 3 5`.

2.1.9 Write a static method `lg()` that takes a `double` argument `n` and returns the base-2 logarithm of `n`. You may use Java’s `Math` library.

2.1.10 Write a static method `lg()` that takes an `int` argument `n` and returns the largest integer not larger than the base-2 logarithm of `n`. Do not use the `Math` library.

2.1.11 Write a static method `signum()` that takes an `int` argument `n` and returns $-1$ if $n$ is less than 0, 0 if $n$ is equal to 0, and $+1$ if $n$ is greater than 0.
2.1.12 Consider the static method `duplicate()` below.

```java
public static String duplicate(String s)
{
    String t = s + s;
    return t;
}
```

What does the following code fragment do?

```java
String s = "Hello";
s = duplicate(s);
String t = "Bye";
t = duplicate(duplicate(duplicate(t)));
StdOut.println(s + t);
```

2.1.13 Consider the static method `cube()` below.

```java
public static void cube(int i)
{
    i = i * i * i;
}
```

How many times is the following `for` loop iterated?

```java
for (int i = 0; i < 1000; i++)
    cube(i);
```

*Answer:* Just 1,000 times. A call to `cube()` has no effect on the client code. It changes the value of its local parameter variable `i`, but that change has no effect on the `i` in the `for` loop, which is a different variable. If you replace the call to `cube(i)` with the statement `i = i * i * i`; (maybe that was what you were thinking), then the loop is iterated five times, with `i` taking on the values 0, 1, 2, 9, and 730 at the beginning of the five iterations.
2.1.14 The following checksum formula is widely used by banks and credit card companies to validate legal account numbers:

\[ d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + \ldots = 0 \pmod{10} \]

The \( d_i \) are the decimal digits of the account number and \( f(d) \) is the sum of the decimal digits of \( 2d \) (for example, \( f(7) = 5 \) because \( 2 \times 7 = 14 \) and \( 1 + 4 = 5 \)). For example, 17,327 is valid because \( 1 + 5 + 3 + 4 + 7 = 20 \), which is a multiple of 10. Implement the function \( f \) and write a program to take a 10-digit integer as a command-line argument and print a valid 11-digit number with the given integer as its first 10 digits and the checksum as the last digit.

2.1.15 Given two stars with angles of declination and right ascension \((d_1, a_1)\) and \((d_2, a_2)\), the angle they subtend is given by the formula

\[ 2 \arcsin\left(\sin^2\left(\frac{d}{2}\right) + \cos(d_1)\cos(d_2)\sin^2\left(\frac{a}{2}\right)\right)^{1/2} \]

where \( a_1 \) and \( a_2 \) are angles between \(-180\) and \(180\) degrees, \( d_1 \) and \( d_2 \) are angles between \(-90\) and \(90\) degrees, \( a = a_2 - a_1 \), and \( d = d_2 - d_1 \). Write a program to take the declination and right ascension of two stars as command-line arguments and print the angle they subtend. *Hint*: Be careful about converting from degrees to radians.

2.1.16 Write a static method `scale()` that takes a `double` array as its argument and has the side effect of scaling the array so that each element is between 0 and 1 (by subtracting the minimum value from each element and then dividing each element by the difference between the minimum and maximum values). Use the `max()` method defined in the table in the text, and write and use a matching `min()` method.

2.1.17 Write a static method `reverse()` that takes an array of strings as its argument and returns a new array with the strings in reverse order. (Do not change the order of the strings in the argument array.) Write a static method `reverseInplace()` that takes an array of strings as its argument and produces the side effect of reversing the order of the strings in the argument array.
2.1.18 Write a static method `readBoolean2D()` that reads a two-dimensional boolean matrix (with dimensions) from standard input and returns the resulting two-dimensional array.

2.1.19 Write a static method `histogram()` that takes an `int` array `a[]` and an integer `m` as arguments and returns an array of length `m` whose `i`th element is the number of times the integer `i` appeared in `a[]`. Assuming the values in `a[]` are all between 0 and `m-1`, the sum of the values in the returned array should equal `a.length`.

2.1.20 Assemble code fragments in this section and in Section 1.4 to develop a program that takes an integer command-line argument `n` and prints `n` five-card hands, separated by blank lines, drawn from a randomly shuffled card deck, one card per line using card names like `Ace of Clubs`.

2.1.21 Write a static method `multiply()` that takes two square matrices of the same dimension as arguments and produces their product (another square matrix of that same dimension). Extra credit: Make your program work whenever the number of columns in the first matrix is equal to the number of rows in the second matrix.

2.1.22 Write a static method `any()` that takes a `boolean` array as its argument and returns `true` if any of the elements in the array is `true`, and `false` otherwise. Write a static method `all()` that takes an array of `boolean` values as its argument and returns `true` if all of the elements in the array are `true`, and `false` otherwise.

2.1.23 Develop a version of `getCoupon()` that better models the situation when one of the coupons is rare: choose one of the `n` values at random, return that value with probability `1/(1,000n)`, and return all other values with equal probability. Extra credit: How does this change affect the expected number of coupons that need to be collected in the coupon collector problem?

2.1.24 Modify `PlayThatTune` to add harmonics two octaves away from each note, with half the weight of the one-octave harmonics.
2.1.25 *Birthday problem.* Develop a class with appropriate static methods for studying the birthday problem (see Exercise 1.4.38).

2.1.26 *Euler’s totient function.* Euler’s totient function is an important function in number theory: \( \varphi(n) \) is defined as the number of positive integers less than or equal to \( n \) that are relatively prime with \( n \) (no factors in common with \( n \) other than 1). Write a class with a static method that takes an integer argument \( n \) and returns \( \varphi(n) \), and a `main()` that takes an integer command-line argument, calls the method with that argument, and prints the resulting value.

2.1.27 *Harmonic numbers.* Write a program `Harmonic` that contains three static methods `harmonic()`, `harmonicSmall()`, and `harmonicLarge()` for computing the harmonic numbers. The `harmonicSmall()` method should just compute the sum (as in Program 1.3.5), the `harmonicLarge()` method should use the approximation

\[
H_n = \log(n) + \gamma + 1/(2n) - 1/(12n^2) + 1/(120n^4) \quad \text{(the number } \gamma = 0.577215664901532... \text{ is known as Euler’s constant,)}
\]

and the `harmonic()` method should call `harmonicSmall()` for \( n < 100 \) and `harmonicLarge()` otherwise.

2.1.28 *Black–Scholes option valuation.* The Black–Scholes formula supplies the theoretical value of a European call option on a stock that pays no dividends, given the current stock price \( s \), the exercise price \( x \), the continuously compounded risk-free interest rate \( r \), the volatility \( \sigma \), and the time (in years) to maturity \( t \). The Black–Scholes value is given by the formula

\[
s \Phi(a) - xe^{-rt} \Phi(b),
\]

where \( \Phi(z) \) is the Gaussian cumulative distribution function, \( a = (\ln(s/x) + (r + \sigma^2/2) t) / (\sigma \sqrt{t}) \), and \( b = a - \sigma \sqrt{t} \). Write a program that takes \( s, r, \sigma, \) and \( t \) from the command line and prints the Black–Scholes value.

2.1.29 *Fourier spikes.* Write a program that takes a command-line argument \( n \) and plots the function

\[
(\cos(t) + \cos(2t) + \cos(3t) + \ldots + \cos(nt)) / n
\]

for 500 equally spaced samples of \( t \) from \(-10\) to \(10\) (in radians). Run your program for \( n = 5 \) and \( n = 500 \). *Note:* You will observe that the sum converges to a spike (0 everywhere except a single value). This property is the basis for a proof that *any* smooth function can be expressed as a sum of sinusoids.
2.1.30 Calendar. Write a program Calendar that takes two integer command-line arguments \( m \) and \( y \) and prints the monthly calendar for month \( m \) of year \( y \), as in this example:

```
% java Calendar 2 2009
February 2009
  S M Tu W Th F S
1  2  3  4  5  6  7
8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

**Hint:** See LeapYear (Program 1.2.4) and Exercise 1.2.29.

2.1.31 Horner’s method. Write a class Horner with a method evaluate() that takes a floating-point number \( x \) and array \( p[] \) as arguments and returns the result of evaluating the polynomial whose coefficients are the elements in \( p[] \) at \( x \):

\[
p(x) = p_0 + p_1 x^1 + p_2 x^2 + \ldots + p_{n-2} x^{n-2} + p_{n-1} x^{n-1}
\]

Use Horner’s method, an efficient way to perform the computations that is suggested by the following parenthesization:

\[
p(x) = p_0 + x (p_1 + x (p_2 + \ldots + x (p_{n-2} + x p_{n-1})\ldots))
\]

Write a test client with a static method exp() that uses evaluate() to compute an approximation to \( e^x \), using the first \( n \) terms of the Taylor series expansion

\[
e^x = 1 + x + x^2/2! + x^3/3! + \ldots
\]

Your client should take a command-line argument \( x \) and compare your result against that computed by `Math.exp(x)`.

2.1.32 Chords. Develop a version of PlayThatTune that can handle songs with chords (including harmonics). Develop an input format that allows you to specify different durations for each chord and different amplitude weights for each note within a chord. Create test files that exercise your program with various chords and harmonics, and create a version of Für Elise that uses them.
2.1.33 Benford’s law. The American astronomer Simon Newcomb observed a quirk in a book that compiled logarithm tables: the beginning pages were much grubbier than the ending pages. He suspected that scientists performed more computations with numbers starting with 1 than with 8 or 9, and postulated that, under general circumstances, the leading digit is much more likely to be 1 (roughly 30%) than the digit 9 (less than 4%). This phenomenon is known as Benford’s law and is now often used as a statistical test. For example, IRS forensic accountants rely on it to discover tax fraud. Write a program that reads in a sequence of integers from standard input and tabulates the number of times each of the digits 1–9 is the leading digit, breaking the computation into a set of appropriate static methods. Use your program to test the law on some tables of information from your computer or from the web. Then, write a program to foil the IRS by generating random amounts from $1.00 to $1,000.00 with the same distribution that you observed.

2.1.34 Binomial distribution. Write a function

```java
public static double binomial(int n, int k, double p)
```

to compute the probability of obtaining exactly $k$ heads in $n$ biased coin flips (heads with probability $p$) using the formula

\[ f(n, k, p) = \frac{p^k (1-p)^{n-k}}{k!(n-k)!} \]

*Hint:* To stave off overflow, compute $x = \ln f(n, k, p)$ and then return $e^x$. In `main()`, take $n$ and $p$ from the command line and check that the sum over all values of $k$ between 0 and $n$ is (approximately) 1. Also, compare every value computed with the normal approximation

\[ f(n, k, p) \approx \phi(np, np(1-p)) \]

(see Exercise 2.2.1).

2.1.35 Coupon collecting from a binomial distribution. Develop a version of `getCoupon()` that uses `binomial()` from the previous exercise to return coupon values according to the binomial distribution with $p = 1/2$. *Hint:* Generate a uniformly random number $x$ between 0 and 1, then return the smallest value of $k$ for which the sum of $f(n, j, p)$ for all $j < k$ exceeds $x$. *Extra credit:* Develop a hypothesis for describing the behavior of the coupon collector function under this assumption.
2.1.36 Postal bar codes. The barcode used by the U.S. Postal System to route mail is defined as follows: Each decimal digit in the ZIP code is encoded using a sequence of three half-height and two full-height bars. The barcode starts and ends with a full-height bar (the guard rail) and includes a checksum digit (after the five-digit ZIP code or ZIP+4), computed by summing up the original digits modulo 10. Implement the following functions

- Draw a half-height or full-height bar on StdDraw.
- Given a digit, draw its sequence of bars.
- Compute the checksum digit.

Also implement a test client that reads in a five- (or nine-) digit ZIP code as the command-line argument and draws the corresponding postal bar code.
Each program that you have written so far consists of Java code that resides in a single .java file. For large programs, keeping all the code in a single file in this way is restrictive and unnecessary. Fortunately, it is very easy in Java to refer to a method in one file that is defined in another. This ability has two important consequences on our style of programming.

First, it enables code reuse. One program can make use of code that is already written and debugged, not by copying the code, but just by referring to it. This ability to define code that can be reused is an essential part of modern programming. It amounts to extending Java—you can define and use your own operations on data.

Second, it enables modular programming. You can not only divide a program up into static methods, as just described in Section 2.1, but also keep those methods in different files, grouped together according to the needs of the application. Modular programming is important because it allows us to independently develop, compile, and debug parts of big programs one piece at a time, leaving each finished piece in its own file for later use without having to worry about its details again. We develop libraries of static methods for use by any other program, keeping each library in its own file and using its methods in any other program. Java’s Math library and our Std* libraries for input/output are examples that you have already used.

More importantly, you will soon see that it is very easy to define libraries of your own. The ability to define libraries and then to use them in multiple programs is a critical aspect of our ability to build programs to address complex tasks.

Having just moved in Section 2.1 from thinking of a Java program as a sequence of statements to thinking of a Java program as a class comprising a set of static methods (one of which is main()), you will be ready after this section to think of a Java program as a set of classes, each of which is an independent module consisting of a set of methods. Since each method can call a method in another class, all of your code can interact as a network of methods that call one another, grouped together in classes. With this capability, you can start to think about managing complexity when programming by breaking up programming tasks into classes that can be implemented and tested independently.
2.2 Libraries and Clients

Using static methods in other programs  To refer to a static method in one class that is defined in another, we use the same mechanism that we have been using to invoke methods such as `Math.sqrt()` and `StdOut.println()`:

- Make both classes accessible to Java (for example, by putting them both in the same directory in your computer).
- To call a method, prepend its class name and a period separator.

For example, we might wish to write a simple client `SAT.java` that takes an SAT score \( z \) from the command line and prints the percentage of students scoring less than \( z \) in a given year (in which the mean score was 1,019 and its standard deviation was 209). To get the job done, `SAT.java` needs to compute \( \Phi((z – 1,019)/209) \), a

```java
% python gaussiantable.py 1019 209
...
%
```

```java
public class SAT
{
    public static void main(String[] args)
    {
        double z = Double.parseDouble(args[0]);
        double v = Gaussian.cdf((z - 1019)/209);
        StdOut.println(v);
    }
}
```

```java
public class Gaussian
{
    public static double cdf(double z)
    {
        if (z < -8.0) return 0.0;
        if (z > 8.0) return 1.0;
        double sum = 0.0;
        double term = z;
        for (int i = 3; sum != sum + term; i += 2)
        {
            sum = sum + term;
            term = term * z * z / i;
        }
        return 0.5 + pdf(z) * sum;
    }
    public static double pdf(double x)
    {
        return Math.exp(-x*x/2) / Math.sqrt(2*Math.PI);
    }
    public static void main(String[] args)
    {
        ...
    }
}
```

```java
public class Math
{
    public static double exp(double x)
    {
        ...
    }
    public static double sqrt(double x)
    {
        ...
    }
    public static void main(String[] args)
    {
        ...
    }
}
```

Flow of control in a modular program
task perfectly suited for the \texttt{cdf()} method in \texttt{Gaussian.java} (Program 2.1.2). All that we need to do is to keep \texttt{Gaussian.java} in the same directory as \texttt{SAT.java} and prepend the class name when calling \texttt{cdf()}. Moreover, any other class in that directory can make use of the static methods defined in Gaussian, by calling \texttt{Gaussian.pdf()} or \texttt{Gaussian.cdf()}. The Math library is always accessible in Java, so any class can call \texttt{Math.sqrt()} and \texttt{Math.exp()}, as usual. The files \texttt{Gaussian.java}, \texttt{SAT.java}, and \texttt{Math.java} implement Java classes that interact with one another: SAT calls a method in Gaussian, which calls another method in Gaussian, which then calls two methods in Math.

The potential effect of programming by defining multiple files, each an independent class with multiple methods, is another profound change in our programming style. Generally, we refer to this approach as modular programming. We independently develop and debug methods for an application and then utilize them at any later time. In this section, we will consider numerous illustrative examples to help you get used to the idea. However, there are several details about the process that we need to discuss before considering more examples.

\textit{The public keyword.} We have been identifying every static method as \texttt{public} since \texttt{HelloWorld}. This modifier identifies the method as available for use by any other program with access to the file. You can also identify methods as \texttt{private} (and there are a few other categories), but you have no reason to do so at this point. We will discuss various options in Section 3.3.

\textit{Each module is a class.} We use the term \texttt{module} to refer to all the code that we keep in a single file. In Java, by convention, each module is a Java \texttt{class} that is kept in a file with the same name of the class but has a .java extension. In this chapter, each \texttt{class} is merely a set of static methods (one of which is main()). You will learn much more about the general structure of the Java \texttt{class} in Chapter 3.

\textit{The .class file.} When you compile the program (by typing \texttt{javac} followed by the class name), the Java compiler makes a file with the class name followed by a .class extension that has the code of your program in a language more suited to your computer. If you have a .class file, you can use the module’s methods in another program even without having the source code in the corresponding .java file (but you are on your own if you discover a bug!).
2.2 Libraries and Clients

Compile when necessary. When you compile a program, Java typically compiles everything that needs to be compiled in order to run that program. If you call Gaussian.cdf() in SAT, then, when you type javac SAT.java, the compiler will also check whether you modified Gaussian.java since the last time it was compiled (by checking the time it was last changed against the time Gaussian.class was created). If so, it will also compile Gaussian.java! If you think about this approach, you will agree that it is actually quite helpful. After all, if you find a bug in Gaussian.java (and fix it), you want all the classes that call methods in Gaussian to use the new version.

Multiple main() methods. Another subtle point is to note that more than one class might have a main() method. In our example, both SAT and Gaussian have their own main() method. If you recall the rule for executing a program, you will see that there is no confusion: when you type java followed by a class name, Java transfers control to the machine code corresponding to the main() method defined in that class. Typically, we include a main() method in every class, to test and debug its methods. When we want to run SAT, we type java SAT; when we want to debug Gaussian, we type java Gaussian (with appropriate command-line arguments).

If you think of each program that you write as something that you might want to make use of later, you will soon find yourself with all sorts of useful tools. Modular programming allows us to view every solution to a computational problem that we may develop as adding value to our computational environment.

For example, suppose that you need to evaluate $\Phi$ for some future application. Why not just cut and paste the code that implements cdf() from Gaussian? That would work, but would leave you with two copies of the code, making it more difficult to maintain. If you later want to fix or improve this code, you would need to do so in both copies. Instead, you can just call Gaussian.cdf(). Our implementations and uses of our methods are soon going to proliferate, so having just one copy of each is a worthy goal.

From this point forward, you should write every program by identifying a reasonable way to divide the computation into separate parts of a manageable size and implementing each part as if someone will want to use it later. Most frequently, that someone will be you, and you will have yourself to thank for saving the effort of rewriting and re-debugging code.
Libraries  We refer to a module whose methods are primarily intended for use by many other programs as a **library**. One of the most important characteristics of programming in Java is that thousands of libraries have been predefined for your use. We reveal information about those that might be of interest to you throughout the book, but we will postpone a detailed discussion of the scope of Java libraries, because many of them are designed for use by experienced programmers. Instead, we focus in this chapter on the even more important idea that we can build **user-defined libraries**, which are nothing more than classes that contain a set of related methods for use by other programs. No Java library can contain all the methods that we might need for a given computation, so this ability to create our own library of methods is a crucial step in addressing complex programming applications.

**Clients.** We use the term **client** to refer to a program that calls a given library method. When a class contains a method that is a client of a method in another class, we say that the first class is a client of the second class. In our example, **Gaussian** is a client of **SAT**. A given class might have multiple clients. For example, all of the programs that you have written that call **Math.sqrt()** or **Math.random()** are clients of **Math**.

**APIs.** Programmers normally think in terms of a **contract** between the client and the implementation that is a clear specification of what the method is to do. When you are writing both clients and implementations, you are making contracts with yourself, which by itself is helpful because it provides extra help in debugging. More important, this approach enables code reuse. You have been able to write programs that are clients of **Std** and **Math** and other built-in Java classes because of an informal contract (an English-
language description of what they are supposed to do) along with a precise specification of the signatures of the methods that are available for use. Collectively, this information is known as an application programming interface (API). This same mechanism is effective for user-defined libraries. The API allows any client to use the library without having to examine the code in the implementation, as you have been doing for Math and Std*. The guiding principle in API design is to provide to clients the methods they need and no others. An API with a huge number of methods may be a burden to implement; an API that is lacking important methods may be unnecessarily inconvenient for clients.

**Implementations.** We use the term implementation to describe the Java code that implements the methods in an API, kept by convention in a file with the library name and a .java extension. Every Java program is an implementation of some API, and no API is of any use without some implementation. Our goal when developing an implementation is to honor the terms of the contract. Often, there are many ways to do so, and separating client code from implementation code gives us the freedom to substitute new and improved implementations.

**FOR EXAMPLE, CONSIDER THE GAUSSIAN DISTRIBUTION functions.** These do not appear in Java’s Math library but are important in applications, so it is worthwhile for us to put them in a library where they can be accessed by future client programs and to articulate this API:

```java
public class Gaussian
{
    double pdf(double x) {
        // Implementation
    }
    double pdf(double x, double mu, double sigma) {
        // Implementation
    }
    double cdf(double z) {
        // Implementation
    }
    double cdf(double z, double mu, double sigma) {
        // Implementation
    }
}
```

API for our library of static methods for Gaussian distribution functions

The API includes not only the one-argument Gaussian distribution functions that we have previously considered (see Program 2.1.2) but also three-argument versions (in which the client specifies the mean and standard deviation of the distribution) that arise in many statistical applications. Implementing the three-argument Gaussian distribution functions is straightforward (see Exercise 2.2.1).
How much information should an API contain? This is a gray area and a hotly debated issue among programmers and computer-science educators. We might try to put as much information as possible in the API, but (as with any contract!) there are limits to the amount of information that we can productively include. In this book, we stick to a principle that parallels our guiding design principle: *provide to client programmers the information they need and no more*. Doing so gives us vastly more flexibility than the alternative of providing detailed information about implementations. Indeed, any extra information amounts to implicitly extending the contract, which is undesirable. Many programmers fall into the bad habit of checking implementation code to try to understand what it does. Doing so might lead to client code that depends on behavior not specified in the API, which would not work with a new implementation. Implementations change more often than you might think. For example, each new release of Java contains many new implementations of library functions.

Often, the implementation comes first. You might have a working module that you later decide would be useful for some task, and you can just start using its methods in other programs. In such a situation, it is wise to carefully articulate the API at some point. The methods may not have been designed for reuse, so it is worthwhile to use an API to do such a design (as we did for Gaussian).

The remainder of this section is devoted to several examples of libraries and clients. Our purpose in considering these libraries is twofold. First, they provide a richer programming environment for your use as you develop increasingly sophisticated client programs of your own. Second, they serve as examples for you to study as you begin to develop libraries for your own use.

**Random numbers** We have written several programs that use `Math.random()`, but our code often uses particular idioms that convert the random `double` values between 0 and 1 that `Math.random()` provides to the type of random numbers that we want to use (random boolean values or random `int` values in a specified range, for example). To effectively reuse our code that implements these idioms, we will, from now on, use the `StdRandom` library in Program 2.2.1. `StdRandom` uses overloading to generate random numbers from various distributions. You can use any of them in the same way that you use our standard I/O libraries (see the first Q&A at the end of Section 2.1). As usual, we summarize the methods in our `StdRandom` library with an API:
2.2 Libraries and Clients

public class StdRandom

void setSeed(long seed)  
set the seed for reproducible results

int uniform(int n)  
integer between 0 and n-1

double uniform(double lo, double hi)  
floating-point number between lo and hi

boolean bernoulli(double p)  
true with probability p, false otherwise

double gaussian()  
Gaussian, mean 0, standard deviation 1

double gaussian(double mu, double sigma)  
Gaussian, mean mu, standard deviation sigma

int discrete(double[] p)  
i with probability p[i]

void shuffle(double[] a)  
randomly shuffle the array a[]

API for our library of static methods for random numbers

These methods are sufficiently familiar that the short descriptions in the API suffice to specify what they do. By collecting all of these methods that use Math.random() to generate random numbers of various types in one file (StdRandom.java), we concentrate our attention on generating random numbers to this one file (and reuse the code in that file) instead of spreading them through every program that uses these methods. Moreover, each program that uses one of these methods is clearer than code that calls Math.random() directly, because its purpose for using Math.random() is clearly articulated by the choice of method from StdRandom.

API design. We make certain assumptions about the values passed to each method in StdRandom. For example, we assume that clients will call uniform(n) only for positive integers n, bernoulli(p) only for p between 0 and 1, and discrete() only for an array whose elements are between 0 and 1 and sum to 1. All of these assumptions are part of the contract between the client and the implementation. We strive to design libraries such that the contract is clear and unambiguous and to avoid getting bogged down with details. As with many tasks in programming, a good API design is often the result of several iterations of trying and living with various possibilities. We always take special care in designing APIs, because when we change an API we might have to change all clients and all implementations. Our goal is to articulate what clients can expect separate from the code in the API. This practice frees us to change the code, and perhaps to use an implementation that achieves the desired effect more efficiently or with more accuracy.
Functions and Modules

Program 2.2.1 Random number library

```java
public class StdRandom {
    public static int uniform(int n) {
        return (int) (Math.random() * n);  }
    public static double uniform(double lo, double hi) {
        return lo + Math.random() * (hi - lo);  }
    public static boolean bernoulli(double p) {
        return Math.random() < p;  }
    public static double gaussian() {
        /* See Exercise 2.2.17. */  }
    public static double gaussian(double mu, double sigma) {
        return mu + sigma * gaussian();  }
    public static int discrete(double[] probabilities) {
        /* See Program 1.6.2. */  }
    public static void shuffle(double[] a) {
        /* See Exercise 2.2.4. */  }
    public static void main(String[] args) {
        /* See text. */  }
}
```

The methods in this library compute various types of random numbers: random nonnegative integer less than a given value, uniformly distributed in a given range, random bit (Bernoulli), standard Gaussian, Gaussian with given mean and standard deviation, and distributed according to a given discrete distribution.

```bash
% java StdRandom 5
90 26.36076 false 8.79269 0
13 18.02210 false 9.03992 1
58 56.41176 true 8.80501 0
29 16.68454 false 8.90827 0
85 86.24712 true 8.95228 0
```
Unit testing. Even though we implement StdRandom without reference to any particular client, it is good programming practice to include a test client main() that, although not used when a client class uses the library, is helpful when debugging and testing the methods in the library. Whenever you create a library, you should include a main() method for unit testing and debugging. Proper unit testing can be a significant programming challenge in itself (for example, the best way of testing whether the methods in StdRandom produce numbers that have the same characteristics as truly random numbers is still debated by experts). At a minimum, you should always include a main() method that

- Exercises all the code
- Provides some assurance that the code is working
- Takes an argument from the command line to allow more testing

Then, you should refine that main() method to do more exhaustive testing as you use the library more extensively. For example, we might start with the following code for StdRandom (leaving the testing of shuffle() for an exercise):

```java
public static void main(String[] args)
{
    int n = Integer.parseInt(args[0]);
    double[] probabilities = { 0.5, 0.3, 0.1, 0.1 };
    for (int i = 0; i < n; i++)
    {
        StdOut.printf(" %2d " , uniform(100));
        StdOut.printf("%8.5f " , uniform(10.0, 99.0));
        StdOut.printf("%5b " , bernoulli(0.5));
        StdOut.printf("%7.5f ", gaussian(9.0, 0.2));
        StdOut.printf("%2d " , discrete(probabilities));
        StdOut.println();
    }
}
```

When we include this code in StdRandom.java and invoke this method as illustrated in Program 2.2.1, the output includes no surprises: the integers in the first column might be equally likely to be any value from 0 to 99; the numbers in the second column might be uniformly spread between 10.0 and 99.0; about half of the values in the third column are true; the numbers in the fourth column seem to average about 9.0, and seem unlikely to be too far from 9.0; and the last column seems to be not far from 50% 0s, 30% 1s, 10% 2s, and 10% 3s. If something seems
amiss in one of the columns, we can type `java StdRandom 10` or `100` to see many more results. In this particular case, we can (and should) do far more extensive testing in a separate client to check that the numbers have many of the same properties as truly random numbers drawn from the cited distributions (see Exercise 2.2.3). One effective approach is to write test clients that use `StdDraw`, as data visualization can be a quick indication that a program is behaving as intended. For example, a plot of a large number of points whose $x$- and $y$-coordinates are both drawn from various distributions often produces a pattern that gives direct insight into the important properties of the distribution. More important, a bug in the random number generation code is likely to show up immediately in such a plot.

Stress testing. An extensively used library such as `StdRandom` should also be subjected to **stress testing**, where we make sure that it does not crash when the client does not follow the contract or makes some assumption that is not explicitly covered. Java libraries have already been subjected to such stress testing, which requires carefully examining each line of code and questioning whether some condition might cause a problem. What should `discrete()` do if the array elements do not sum to exactly 1? What if the argument is an array of length 0? What should the two-argument `uniform()` do if one or both of its arguments is NaN? Infinity? Any question that you can think of is fair game. Such cases are sometimes referred to as **corner cases**. You are certain to encounter a teacher or a supervisor who is a stickler about corner cases. With experience, most programmers learn to address them early, to avoid an unpleasant bout of debugging later. Again, a reasonable approach is to implement a stress test as a separate client.
2.2 Libraries and Clients

Input and output for arrays We have seen—and will continue to see—many examples where we wish to keep data in arrays for processing. Accordingly, it is useful to build a library that complements StdIn and StdOut by providing static methods for reading arrays of primitive types from standard input and printing them to standard output. The following API provides these methods:

```
public class StdArrayIO

double[] readDouble1D()  // read a one-dimensional array of double values

double[][] readDouble2D() // read a two-dimensional array of double values

void print(double[] a)     // print a one-dimensional array of double values

void print(double[][] a)    // print a two-dimensional array of double values
```

Note 1. 1D format is an integer n followed by n values.
Note 2. 2D format is two integers m and n followed by m × n values in row-major order.
Note 3. Methods for int and boolean are also included.

API for our library of static methods for array input and output

The first two notes at the bottom of the table reflect the idea that we need to settle on a file format. For simplicity and harmony, we adopt the convention that all values appearing in standard input include the dimension(s) and appear in the order indicated. The read*() methods expect input in this format; the print() methods produce output in this format. The third note at the bottom of the table indicates that StdArrayIO actually contains 12 methods—four each for int, double, and boolean. The print() methods are overloaded (they all have the same name print() but different types of arguments), but the read*() methods need different names, formed by adding the type name (capitalized, as in StdIn) followed by 1D or 2D.

Implementing these methods is straightforward from the array-processing code that we have considered in Section 1.4 and in Section 2.1, as shown in StdArrayIO (PROGRAM 2.2.2). Packaging up all of these static methods into one file—StdArrayIO.java—allows us to easily reuse the code and saves us from having to worry about the details of reading and printing arrays when writing client programs later on.
**Program 2.2.2  Array I/O library**

```java
public class StdArrayIO {
    public static double[] readDouble1D() {
        /* See Exercise 2.2.11. */
    }

    public static double[][] readDouble2D() {
        int m = StdIn.readInt();
        int n = StdIn.readInt();
        double[][] a = new double[m][n];
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                a[i][j] = StdIn.readDouble();
        return a;
    }

    public static void print(double[] a) {
        /* See Exercise 2.2.11. */
    }

    public static void print(double[][] a) {
        int m = a.length;
        int n = a[0].length;
        System.out.println(m + " " + n);
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                StdOut.printf("%9.5f ", a[i][j]);
        StdOut.println();
    }

    // Methods for other types are similar (see booksite).
    public static void main(String[] args) {
        print(readDouble2D());
    }
}
```

This library of static methods facilitates reading one-dimensional and two-dimensional arrays from standard input and printing them to standard output. The file format includes the dimensions (see accompanying text). Numbers in the output in the example are truncated.
2.2 Libraries and Clients

**Iterated function systems** Scientists have discovered that complex visual images can arise unexpectedly from simple computational processes. With StdRandom, StdDraw, and StdArrayIO, we can study the behavior of such systems.

**Sierpinski triangle.** As a first example, consider the following simple process: Start by plotting a point at one of the vertices of a given equilateral triangle. Then pick one of the three vertices at random and plot a new point halfway between the point just plotted and that vertex. Continue performing this same operation. Each time, we are pick a random vertex from the triangle to establish the line whose midpoint will be the next point plotted. Since we make random choices, the set of points should have some of the characteristics of random points, and that does seem to be the case after the first few iterations:

We can study the process for a large number of iterations by writing a program to plot trials points according to the rules:

```java
double[] cx = { 0.000, 1.000, 0.500 };
double[] cy = { 0.000, 0.000, 0.866 };
double x = 0.0, y = 0.0;
for (int t = 0; t < trials; t++)
{
    int r = StdRandom.uniform(3);
    x = (x + cx[r]) / 2.0;
    y = (y + cy[r]) / 2.0;
    StdDraw.point(x, y);
}
```

We keep the $x$- and $y$-coordinates of the triangle vertices in the arrays $cx[]$ and $cy[]$, respectively. We use StdRandom.uniform() to choose a random index $r$ into
these arrays—the coordinates of the chosen vertex are \((cx[r], cy[r])\). The \(x\)-coordinate of the midpoint of the line from \((x, y)\) to that vertex is given by the expression \((x + cx[r])/2.0\), and a similar calculation gives the \(y\)-coordinate. Adding a call to `StdDraw.point()` and putting this code in a loop completes the implementation. Remarkably, despite the randomness, the same figure always emerges after a large number of iterations! This figure is known as the Sierpinski triangle (see Exercise 2.3.27). Understanding why such a regular figure should arise from such a random process is a fascinating question.

![A random process?](image)

**Barnsley fern.** To add to the mystery, we can produce pictures of remarkable diversity by playing the same game with different rules. One striking example is known as the Barnsley fern. To generate it, we use the same process, but this time driven by the following table of formulas. At each step, we choose the formulas to use to update \(x\) and \(y\) with the indicated probability (1% of the time we use the first pair of formulas, 85% of the time we use the second pair of formulas, and so forth).

<table>
<thead>
<tr>
<th>probability</th>
<th>(x)-update</th>
<th>(y)-update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>(x = 0.500)</td>
<td>(y = 0.16y)</td>
</tr>
<tr>
<td>85%</td>
<td>(x = 0.85x + 0.04y + 0.075)</td>
<td>(y = -0.04x + 0.85y + 0.180)</td>
</tr>
<tr>
<td>7%</td>
<td>(x = 0.20x - 0.26y + 0.400)</td>
<td>(y = 0.23x + 0.22y + 0.045)</td>
</tr>
<tr>
<td>7%</td>
<td>(x = -0.15x + 0.28y + 0.575)</td>
<td>(y = 0.26x + 0.24y - 0.086)</td>
</tr>
</tbody>
</table>
2.2 Libraries and Clients

Program 2.2.3 Iterated function systems

```java
public class IFS {
    public static void main(String[] args) {
        // Plot trials iterations of IFS on StdIn.
        int trials = Integer.parseInt(args[0]);
        double[] dist = StdArrayIO.readDouble1D();
        double[][] cx = StdArrayIO.readDouble2D();
        double[][] cy = StdArrayIO.readDouble2D();
        double x = 0.0, y = 0.0;
        for (int t = 0; t < trials; t++) {
            // Plot 1 iteration.
            int r = StdRandom.discrete(dist);
            double x0 = cx[r][0]*x + cx[r][1]*y + cx[r][2];
            double y0 = cy[r][0]*x + cy[r][1]*y + cy[r][2];
            x = x0;
            y = y0;
            StdDraw.point(x, y);
        }
    }
}
```

This data-driven client of `StdArrayIO`, `StdRandom`, and `StdDraw` iterates the function system defined by a 1-by-m vector (probabilities) and two m-by-3 matrices (coefficients for updating x and y, respectively) on standard input, plotting the result as a set of points on standard drawing. Curiously, this code does not need to know the value of m, as it uses separate methods to create and process the matrices.

```bash
% more sierpinski.txt
3
  33 33 34
3 3
  .50 .00 .00
  .50 .00 .50
  .50 .00 .25
3 3
  .00 .50 .00
  .00 .50 .00
  .00 .50 .433

% java IFS 10000 < sierpinski.txt
```

![Image of the fractal generated by the program]
Examples of iterated function systems

% more barnsley.txt
4
0.01 0.85 0.07 0.07
4 3
0.00 0.00 0.500
0.85 0.04 0.075
0.20 -0.26 0.400
-0.15 0.28 0.575
4 3
0.00 0.16 0.000
-0.04 0.85 0.180
0.23 0.22 0.045
0.26 0.24 -0.086

% java IFS 20000 < barnsley.txt

% more tree.txt
6
0.1 0.1 0.2 0.2 0.2 0.2
6 3
0.00 0.00 0.550
-0.05 0.00 0.525
0.46 -0.15 0.270
0.47 -0.15 0.265
0.43 0.26 0.290
0.42 0.26 0.290
6 3
0.00 0.60 0.000
-0.50 0.00 0.750
0.39 0.38 0.105
0.17 0.42 0.465
-0.25 0.45 0.625
-0.35 0.31 0.525

% java IFS 20000 < tree.txt

% more coral.txt
3
0.40 0.15 0.45
3 3
0.3077 -0.5315 0.8863
0.3077 -0.0769 0.2166
0.0000 0.5455 0.0106
3 3
-0.4615 -0.2937 1.0962
-0.1538 -0.4476 0.3384
0.6923 -0.1958 0.3808

% java IFS 20000 < coral.txt
We could write code just like the code we just wrote for the Sierpinski triangle to iterate these rules, but matrix processing provides a uniform way to generalize that code to handle any set of rules. We have \( m \) different transformations, chosen from a 1-by-\( m \) vector with \texttt{StdRandom.discrete()}.

For each transformation, we have an equation for updating \( x \) and an equation for updating \( y \), so we use two \( m \)-by-3 matrices for the equation coefficients, one for \( x \) and one for \( y \). \texttt{IFS} (Program 2.2.3) implements this data-driven version of the computation.

This program enables limitless exploration: it performs the iteration for any input containing a vector that defines the probability distribution and the two matrices that define the coefficients, one for updating \( x \) and the other for updating \( y \). For the coefficients just given, again, even though we choose a random equation at each step, the same figure emerges every time that we do this computation: an image that looks remarkably similar to a fern that you might see in the woods, not something generated by a random process on a computer.

That the same short program that takes a few numbers from standard input and plots points on standard drawing can (given different data) produce both the Sierpinski triangle and the Barnsley fern (and many, many other images) is truly remarkable. Because of its simplicity and the appeal of the results, this sort of calculation is useful in making synthetic images that have a realistic appearance in computer-generated movies and games.

Perhaps more significantly, the ability to produce such realistic diagrams so easily suggests intriguing scientific questions: What does computation tell us about nature? What does nature tell us about computation?
Statistics. Next, we consider a library for a set of mathematical calculations and basic visualization tools that arise in all sorts of applications in science and engineering and are not all implemented in standard Java libraries. These calculations relate to the task of understanding the statistical properties of a set of numbers. Such a library is useful, for example, when we perform a series of scientific experiments that yield measurements of a quantity. One of the most important challenges facing modern scientists is proper analysis of such data, and computation is playing an increasingly important role in such analysis. These basic data analysis methods that we will consider are summarized in the following API:

```java
class StdStats {
    public double max(double[] a) { /* largest value */ }
    public double min(double[] a) { /* smallest value */ }
    public double mean(double[] a) { /* average */ }
    public double var(double[] a) { /* sample variance */ }
    public double stddev(double[] a) { /* sample standard deviation */ }
    public double median(double[] a) { /* median */ }
    public void plotPoints(double[] a) { /* plot points at (i, a[i]) */ }
    public void plotLines(double[] a) { /* plot lines connecting points at (i, a[i]) */ }
    public void plotBars(double[] a) { /* plot bars to points at (i, a[i]) */ }
}
```

Note: Overloaded implementations are included for other numeric types.

API for our library of static methods for data analysis

Basic statistics. Suppose that we have \( n \) measurements \( x_0, x_1, \ldots, x_{n-1} \). The average value of those measurements, otherwise known as the mean, is given by the formula

\[
\mu = \frac{x_0 + x_1 + \ldots + x_{n-1}}{n}
\]

and is an estimate of the value of the quantity. The minimum and maximum values are also of interest, as is the median (the value that is smaller than and larger than half the values). Also of interest is the sample variance, which is given by the formula

\[
\sigma^2 = \frac{(x_0 - \mu)^2 + (x_1 - \mu)^2 + \ldots + (x_{n-1} - \mu)^2}{n-1}
\]
2.2 Libraries and Clients

Program 2.2.4  Data analysis library

```java
public class StdStats {

  public static double max(double[] a)
  {  // Compute maximum value in a[].
      double max = Double.NEGATIVE_INFINITY;
      for (int i = 0; i < a.length; i++)
        if (a[i] > max) max = a[i];
      return max;
  }

  public static double mean(double[] a)
  {  // Compute the average of the values in a[].
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
        sum += a[i];
      return sum / a.length;
  }

  public static double var(double[] a)
  {  // Compute the sample variance of the values in a[].
      double avg = mean(a);
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
        sum += (a[i] - avg) * (a[i] - avg);
      return sum / (a.length - 1);
  }

  public static double stddev(double[] a)
  { return Math.sqrt(var(a));  }
  // See Program 2.2.5 for plotting methods.

  public static void main(String[] args)
  {  /* See text. */  }
}
```

This code implements methods to compute the maximum, mean, variance, and standard deviation of numbers in a client array. The method for computing the minimum is omitted; plotting methods are in Program 2.2.5; see Exercise 4.2.20 for median().

```bash
% java StdStats < tiny1D.txt
  min 1.000
  mean 3.000
  max 5.000
  std dev 1.581
```
and the sample standard deviation, the square root of the sample variance. StdStats (Program 2.2.4) shows implementations of static methods for computing these basic statistics (the median is more difficult to compute than the others—we will consider the implementation of median() in Section 4.2). The main() test client for StdStats reads numbers from standard input into an array and calls each of the methods to print the minimum, mean, maximum, and standard deviation, as follows:

```java
public static void main(String[] args) {
    double[] a = StdArrayIO.readDouble1D();
    StdOut.printf(" min %7.3f\n", min(a));
    StdOut.printf(" mean %7.3f\n", mean(a));
    StdOut.printf(" max %7.3f\n", max(a));
    StdOut.printf(" std dev %7.3f\n", stddev(a));
}
```

As with StdRandom, a more extensive test of the calculations is called for (see Exercise 2.2.3). Typically, as we debug or test new methods in the library, we adjust the unit testing code accordingly, testing the methods one at a time. A mature and widely used library like StdStats also deserves a stress-testing client for extensively testing everything after any change. If you are interested in seeing what such a client might look like, you can find one for StdStats on the booksite. Most experienced programmers will advise you that any time spent doing unit testing and stress testing will more than pay for itself later.

**Plotting.** One important use of StdDraw is to help us visualize data rather than relying on tables of numbers. In a typical situation, we perform experiments, save the experimental data in an array, and then compare the results against a model, perhaps a mathematical function that describes the data. To expedite this process for the typical case where values of one variable are equally spaced, our StdStats library contains static methods that you can use for plotting data in an array. Program 2.2.5 is an implementation of the plotPoints(), plotLines(), and plotBars() methods for StdStats. These methods display the values in the argument array at evenly spaced intervals in the drawing window, either connected together by line segments (lines), filled circles at each value (points), or bars from the x-axis to the value (bars). They all plot the points with x-coordinate i and y-coordinate a[i] using filled circles, lines through the points, and bars, respectively. In addition,
2.2 Libraries and Clients

Program 2.2.5 Plotting data values in an array

```java
public static void plotPoints(double[] a)
{  // Plot points at (i, a[i]).
    int n = a.length;
    StdDraw.setXscale(-1, n);
    StdDraw.setPenRadius(1/(3.0*n));
    for (int i = 0; i < n; i++)
        StdDraw.point(i, a[i]);
}

public static void plotLines(double[] a)
{  // Plot lines through points at (i, a[i]).
    int n = a.length;
    StdDraw.setXscale(-1, n);
    StdDraw.setPenRadius();
    for (int i = 1; i < n; i++)
        StdDraw.line(i-1, a[i-1], i, a[i]);
}

public static void plotBars(double[] a)
{  // Plot bars from (0, a[i]) to (i, a[i]).
    int n = a.length;
    StdDraw.setXscale(-1, n);
    for (int i = 0; i < n; i++)
        StdDraw.filledRectangle(i, a[i]/2, 0.25, a[i]/2);
}
```

This code implements three methods in StdStats (Program 2.2.4) for plotting data. They plot the points (i, a[i]) with filled circles, connecting line segments, and bars, respectively.

```java
int n = 20;
double[] a = new double[n];
for (int i = 0; i < n; i++)
a[i] = 1.0/(i+1);
plotPoints(a); plotLines(a); plotBars(a);
```
they all rescale $x$ to fill the drawing window (so that the points are evenly spaced along the $x$-coordinate) and leave to the client scaling of the $y$-coordinates.

These methods are not intended to be a general-purpose plotting package, but you can certainly think of all sorts of things that you might want to add: different types of spots, labeled axes, color, and many other artifacts are commonly found in modern systems that can plot data. Some situations might call for more complicated methods than these.

Our intent with `StdStats` is to introduce you to data analysis while showing you how easy it is to define a library to take care of useful tasks. Indeed, this library has already proved useful—we use these plotting methods to produce the figures in this book that depict function graphs, sound waves, and experimental results. Next, we consider several examples of their use.

**Plotting function graphs.** You can use the `StdStats.plot*()` methods to draw a plot of the function graph for any function at all: choose an $x$-interval where you want to plot the function, compute function values evenly spaced through that interval and store them in an array, determine and set the $y$-scale, and then call `StdStats.plotLines()` or another `plot*()` method. For example, to plot a sine function, rescale the $y$-axis to cover values between $-1$ and $1$. Scaling the $x$-axis is automatically handled by the `StdStats` methods. If you do not know the range, you can handle the situation by calling:

```java
int n = 50;
double[] a = new double[n+1];
for (int i = 0; i <= n; i++)
    a[i] = Gaussian.pdf(-4.0 + 8.0*i/n);
StdStats.plotPoints(a);
StdStats.plotLines(a);
```

![Plotting a function graph](image)

The smoothness of the curve is determined by properties of the function and by the number of points plotted. As we discussed when first considering `StdDraw`, you have to be careful to sample enough points to catch fluctuations in the function. We will consider another approach to plotting functions based on sampling values that are not equally spaced in **Section 2.4.**
2.2 Libraries and Clients

Plotting sound waves. Both the StdAudio library and the StdStats plot methods work with arrays that contain sampled values at regular intervals. The diagrams of sound waves in Section 1.5 and at the beginning of this section were all produced by first scaling the y-axis with StdDraw.setYscale(-1, 1), then plotting the points with StdStats.plotPoints(). As you have seen, such plots give direct insight into processing audio. You can also produce interesting effects by plotting sound waves as you play them with StdAudio, although this task is a bit challenging because of the huge amount of data involved (see Exercise 1.5.23).

Plotting experimental results. You can put multiple plots on the same drawing. One typical reason to do so is to compare experimental results with a theoretical model. For example, Bernoulli (Program 2.2.6) counts the number of heads found when a fair coin is flipped \( n \) times and compares the result with the predicted Gaussian probability density function. A famous result from probability theory is that the distribution of this quantity is the binomial distribution, which is extremely well approximated by the Gaussian distribution with mean \( n/2 \) and standard deviation \( \sqrt{n}/2 \). The more trials we perform, the more accurate the approximation. The drawing produced by Bernoulli is a succinct summary of the results of the experiment and a convincing validation of the theory. This example is prototypical of a scientific approach to applications programming that we use often throughout this book and that you should use whenever you run an experiment. If a theoretical model that can explain your results is available, a visual plot comparing the experiment to the theory can validate both.

These few examples are intended to suggest what is possible with a well-designed library of static methods for data analysis. Several extensions and other ideas are explored in the exercises. You will find StdStats to be useful for basic plots, and you are encouraged to experiment with these implementations and to modify them or to add methods to make your own library that can draw plots of your own design. As you continue to address an ever-widening circle of programming tasks, you will naturally be drawn to the idea of developing tools like these for your own use.
Program 2.2.6  Bernoulli trials

public class Bernoulli
{
    public static int binomial(int n)
    {  // Simulate flipping a coin n times; return # heads.
        int heads = 0;
        for (int i = 0; i < n; i++)
            if (StdRandom.bernoulli(0.5)) heads++;
        return heads;
    }
    public static void main(String[] args)
    {  // Perform Bernoulli trials, plot results and model.
        int n = Integer.parseInt(args[0]);
        int trials = Integer.parseInt(args[1]);
        int[] freq = new int[n+1];
        for (int t = 0; t < trials; t++)
            freq[binomial(n)]++;
        double[] norm = new double[n+1];
        for (int i = 0; i <= n; i++)
            norm[i] = (double) freq[i] / trials;
        StdStats.plotBars(norm);
        double mean = n / 2.0;
        double stddev = Math.sqrt(n) / 2.0;
        double[] phi = new double[n+1];
        for (int i = 0; i <= n; i++)
            phi[i] = Gaussian.pdf(i, mean, stddev);
        StdStats.plotLines(phi);
    }
}

This StdStats, StdRandom, and Gaussian client provides visual evidence that the number of heads observed when a fair coin is flipped n times obeys a Gaussian distribution.

% java Bernoulli 20 100000
Modular programming  The library implementations that we have developed illustrate a programming style known as modular programming. Instead of writing a new program that is self-contained within its own file to address a new problem, we break up each task into smaller, more manageable subtasks, then implement and independently debug code that addresses each subtask. Good libraries facilitate modular programming by allowing us to define and provide solutions for important subtasks for future clients. Whenever you can clearly separate tasks within a program, you should do so. Java supports such separation by allowing us to independently debug and later use classes in separate files. Traditionally, programmers use the term module to refer to code that can be compiled and run independently; in Java, each class is a module.

IFS (Program 2.2.3) exemplifies modular programming. This relatively sophisticated computation is implemented with several relatively small modules, developed independently. It uses StdRandom and StdArrayIO, as well as the methods from Integer and StdDraw that we are accustomed to using. If we were to put all of the code required for IFS in a single file, we would have a large amount of code on our hands to maintain and debug; with modular programming, we can study iterated function systems with some confidence that the arrays are read properly and that the random number generator will produce properly distributed values, because we already implemented and tested the code for these tasks in separate modules.

Similarly, Bernoulli (Program 2.2.6) exemplifies modular programming. It is a client of Gaussian, Integer, Math, StdRandom, and StdStats. Again, we can have some confidence that the methods in these modules produce the expected results because they are system libraries or libraries that we have tested, debugged, and used before.

<table>
<thead>
<tr>
<th>API</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian</td>
<td>Gaussian distribution functions</td>
</tr>
<tr>
<td>StdRandom</td>
<td>random numbers</td>
</tr>
<tr>
<td>StdArrayIO</td>
<td>input and output for arrays</td>
</tr>
<tr>
<td>IFS</td>
<td>client for iterated function systems</td>
</tr>
<tr>
<td>StdStats</td>
<td>functions for data analysis</td>
</tr>
<tr>
<td>Bernoulli</td>
<td>client for Bernoulli trials</td>
</tr>
</tbody>
</table>

Summary of classes in this section
To describe the relationships among modules in a modular program, we often draw a *dependency graph*, where we connect two class names with an arrow labeled with the name of a method if the first class contains a method call and the second class contains the definition of the method. Such diagrams play an important role because understanding the relationships among modules is necessary for proper development and maintenance.

We emphasize modular programming throughout this book because it has many important advantages that have come to be accepted as essential in modern programming, including the following:

- We can have programs of a reasonable size, even in large systems.
- Debugging is restricted to small pieces of code.
- We can reuse code without having to re-implement it.
- Maintaining (and improving) code is much simpler.

The importance of these advantages is difficult to overstate, so we will expand upon each of them.

**Programs of a reasonable size.** No large task is so complex that it cannot be divided into smaller subtasks. If you find yourself with a program that stretches to more than a few pages of code, you must ask yourself the following questions: Are there subtasks that could be implemented separately? Could some of these subtasks be
logically grouped together in a separate library? Could other clients use this code in the future? At the other end of the range, if you find yourself with a huge number of tiny modules, you must ask yourself questions such as these: Is there some group of subtasks that logically belong in the same module? Is each module likely to be used by multiple clients? There is no hard-and-fast rule on module size: one implementation of a critically important abstraction might properly be a few lines of code, whereas another library with a large number of overloaded methods might properly stretch to hundreds of lines of code.

**Debugging.** Tracing a program rapidly becomes more difficult as the number of statements and interacting variables increases. Tracing a program with hundreds of variables requires keeping track of hundreds of values, as any statement might affect or be affected by any variable. To do so for hundreds or thousands of statements or more is untenable. With modular programming and our guiding principle of keeping the scope of variables local to the extent possible, we severely restrict the number of possibilities that we have to consider when debugging. Equally important is the idea of a contract between client and implementation. Once we are satisfied that an implementation is meeting its end of the bargain, we can debug all its clients under that assumption.

**Code reuse.** Once we have implemented libraries such as StdStats and StdRandom, we do not have to worry about writing code to compute averages or standard deviations or to generate random numbers again—we can simply reuse the code that we have written. Moreover, we do not need to make copies of the code: any module can just refer to any public method in any other module.

**Maintenance.** Like a good piece of writing, a good program can always be improved, and modular programming facilitates the process of continually improving your Java programs because improving a module improves all of its clients. For example, it is normally the case that there are several different approaches to solving a particular problem. With modular programming, you can implement more than one and try them independently. More importantly, suppose that while developing a new client, you find a bug in some module. With modular programming, fixing that bug essentially fixes bugs in all of the module’s clients.
If you encounter an old program (or a new program written by an old programmer!), you are likely to find one huge module—a long sequence of statements, stretching to several pages or more, where any statement can refer to any variable in the program. Old programs of this kind are found in critical parts of our computational infrastructure (for example, some nuclear power plants and some banks) precisely because the programmers charged with maintaining them cannot even understand them well enough to rewrite them in a modern language! With support for modular programming, modern languages like Java help us avoid such situations by separately developing libraries of methods in independent classes.

The ability to share static methods among different files fundamentally extends our programming model in two different ways. First, it allows us to reuse code without having to maintain multiple copies of it. Second, by allowing us to organize a program into files of manageable size that can be independently debugged and compiled, it strongly supports our basic message: whenever you can clearly separate tasks within a program, you should do so.

In this section, we have supplemented the Std* libraries of Section 1.5 with several other libraries that you can use: Gaussian, StdArrayIO, StdRandom, and StdStats. Furthermore, we have illustrated their use with several client programs. These tools are centered on basic mathematical concepts that arise in any scientific project or engineering task. Our intent is not just to provide tools, but also to illustrate that it is easy to create your own tools. The first question that most modern programmers ask when addressing a complex task is “Which tools do I need?” When the needed tools are not conveniently available, the second question is “How difficult would it be to implement them?” To be a good programmer, you need to have the confidence to build a software tool when you need it and the wisdom to know when it might be better to seek a solution in a library.

After libraries and modular programming, you have one more step to learn a complete modern programming model: object-oriented programming, the topic of Chapter 3. With object-oriented programming, you can build libraries of functions that use side effects (in a tightly controlled manner) to vastly extend the Java programming model. Before moving to object-oriented programming, we consider in this chapter the profound ramifications of the idea that any method can call itself (in Section 2.3) and a more extensive case study (in Section 2.4) of modular programming than the small clients in this section.
2.2 Libraries and Clients

Q&A

Q. I tried to use StdRandom, but got the error message Exception in thread "main" java.lang.NoClassDefFoundError: StdRandom. What's wrong?

A. You need to make StdRandom accessible to Java. See the first Q&A at the end of Section 1.5.

Q. Is there a keyword that identifies a class as a library?

A. No, any set of public methods will do. There is a bit of a conceptual leap in this viewpoint because it is one thing to sit down to create a .java file that you will compile and run, quite another thing to create a .java file that you will rely on much later in the future, and still another thing to create a .java file for someone else to use in the future. You need to develop some libraries for your own use before engaging in this sort of activity, which is the province of experienced systems programmers.

Q. How do I develop a new version of a library that I have been using for a while?

A. With care. Any change to the API might break any client program, so it is best to work in a separate directory. When you use this approach, you are working with a copy of the code. If you are changing a library that has a lot of clients, you can appreciate the problems faced by companies putting out new versions of their software. If you just want to add a few methods to a library, go ahead: that is usually not too dangerous, though you should realize that you might find yourself in a situation where you have to support that library for years!

Q. How do I know that an implementation behaves properly? Why not automatically check that it satisfies the API?

A. We use informal specifications because writing a detailed specification is not much different from writing a program. Moreover, a fundamental tenet of theoretical computer science says that doing so does not even solve the basic problem, because generally there is no way to check that two different programs perform the same computation.
2.2.1 Add to Gaussian (Program 2.1.2) an implementation of the three-argument static method \( \text{pdf}(x, \mu, \sigma) \) specified in the API that computes the Gaussian probability density function with a given mean \( \mu \) and standard deviation \( \sigma \), based on the formula \( \Phi(x, \mu, \sigma) = \Phi((x - \mu) / \sigma) \). Also add an implementation of the associated cumulative distribution function \( \text{cdf}(z, \mu, \sigma) \), based on the formula \( \Phi(z, \mu, \sigma) = \Phi((z - \mu) / \sigma) \).

2.2.2 Write a library of static methods that implements the hyperbolic functions based on the definitions \( \sinh(x) = (e^x - e^{-x}) / 2 \) and \( \cosh(x) = (e^x + e^{-x}) / 2 \), with \( \tanh(x), \coth(x), \sech(x), \text{and csch}(x) \) defined in a manner analogous to standard trigonometric functions.

2.2.3 Write a test client for both StdStats and StdRandom that checks that the methods in both libraries operate as expected. Take a command-line argument \( n \), generate \( n \) random numbers using each of the methods in StdRandom, and print their statistics. Extra credit: Defend the results that you get by comparing them to those that are to be expected from analysis.

2.2.4 Add to StdRandom a method \( \text{shuffle()} \) that takes an array of double values as argument and rearranges them in random order. Implement a test client that checks that each permutation of the array is produced about the same number of times. Add overloaded methods that take arrays of integers and strings.

2.2.5 Develop a client that does stress testing for StdRandom. Pay particular attention to \( \text{discrete()} \). For example, do the probabilities sum to 1?

2.2.6 Write a static method that takes double values \( y_{\text{min}} \) and \( y_{\text{max}} \) (with \( y_{\text{min}} \) strictly less than \( y_{\text{max}} \)), and a double array \( a[] \) as arguments and uses the StdStats library to linearly scale the values in \( a[] \) so that they are all between \( y_{\text{min}} \) and \( y_{\text{max}} \).

2.2.7 Write a Gaussian and StdStats client that explores the effects of changing the mean and standard deviation for the Gaussian probability density function. Create one plot with the Gaussian distributions having a fixed mean and various standard deviations and another with Gaussian distributions having a fixed standard deviation and various means.
2.2.8 Add a method `exp()` to `StdRandom` that takes an argument \( \lambda \) and returns a random number drawn from the exponential distribution with rate \( \lambda \). **Hint:** If \( x \) is a random number uniformly distributed between 0 and 1, then \(-\ln x / \lambda\) is a random number from the exponential distribution with rate \( \lambda \).

2.2.9 Add to `StdRandom` a static method `maxwellBoltzmann()` that returns a random value drawn from a Maxwell–Boltzmann distribution with parameter \( \sigma \). To produce such a value, return the square root of the sum of the squares of three random numbers drawn from the Gaussian distribution with mean 0 and standard deviation \( \sigma \). The speeds of molecules in an ideal gas obey a Maxwell–Boltzmann distribution.

2.2.10 Modify `Bernoulli` (Program 2.2.6) to animate the bar graph, reploting it after each experiment, so that you can watch it converge to the Gaussian distribution. Then add a command-line argument and an overloaded `binomial()` implementation to allow you to specify the probability \( p \) that a biased coin comes up heads, and run experiments to get a feeling for the distribution corresponding to a biased coin. Be sure to try values of \( p \) that are close to 0 and close to 1.

2.2.11 Develop a full implementation of `StdArrayIO` (implement all 12 methods indicated in the API).

2.2.12 Write a library `Matrix` that implements the following API:

```java
public class Matrix {
    double dot(double[] a, double[] b) // vector dot product
    double[][] multiply(double[][] a, double[][] b) // matrix–matrix product
    double[][] transpose(double[][] a) // transpose
    double[] multiply(double[][] a, double[] x) // matrix–vector product
    double[] multiply(double[] x, double[][] a) // vector–matrix product
}
```

(See Section 1.4.) As a test client, use the following code, which performs the same calculation as `Markov` (Program 1.6.3):
public static void main(String[] args)
{
    int trials = Integer.parseInt(args[0]);
    double[][] p = StdArrayIO.readDouble2D();
    double[] ranks = new double[p.length];
    rank[0] = 1.0;
    for (int t = 0; t < trials; t++)
        ranks = Matrix.multiply(ranks, p);
    StdArrayIO.print(ranks);
}

Mathematicians and scientists use mature libraries or special-purpose matrix-processing languages for such tasks. See the booksite for details on using such libraries.

2.2.13 Write a Matrix client that implements the version of Markov described in Section 1.6 but is based on squaring the matrix, instead of iterating the vector–matrix multiplication.

2.2.14 Rewrite RandomSurfer (Program 1.6.2) using the StdArrayIO and StdRandom libraries.

Partial solution.

... 
    double[][] p = StdArrayIO.readDouble2D();
    int page = 0; // Start at page 0.
    int[] freq = new int[n];
    for (int t = 0; t < trials; t++)
    {
        page = StdRandom.discrete(p[page]);
        freq[page]++;
    }
...
2.2.15 Sicherman dice. Suppose that you have two six-sided dice, one with faces labeled 1, 3, 4, 5, 6, and 8 and the other with faces labeled 1, 2, 2, 3, 3, and 4. Compare the probabilities of occurrence of each of the values of the sum of the dice with those for a standard pair of dice. Use StdRandom and StdStats.

2.2.16 Craps. The following are the rules for a pass bet in the game of craps. Roll two six-sided dice, and let \( x \) be their sum.
- If \( x \) is 7 or 11, you win.
- If \( x \) is 2, 3, or 12, you lose.
Otherwise, repeatedly roll the two dice until their sum is either \( x \) or 7.
- If their sum is \( x \), you win.
- If their sum is 7, you lose.

Write a modular program to estimate the probability of winning a pass bet. Modify your program to handle loaded dice, where the probability of a die landing on 1 is taken from the command line, the probability of landing on 6 is 1/6 minus that probability, and 2–5 are assumed equally likely. Hint: Use StdRandom.discrete().

2.2.17 Gaussian random values. Implement the no-argument gaussian() function in StdRandom (PROGRAM 2.2.1) using the Box–Muller formula (see EXERCISE 1.2.27). Next, consider an alternative approach, known as Marsaglia’s method, which is based on generating a random point in the unit circle and using a form of the Box–Muller formula (see the discussion of do-while at the end of SECTION 1.3).

```java
public static double gaussian()
{
    double r, x, y;
    do {
        x = uniform(-1.0, 1.0);
        y = uniform(-1.0, 1.0);
        r = x*x + y*y;
    } while (r >= 1 || r == 0);
    return x * Math.sqrt(-2 * Math.log(r) / r);
}
```

For each approach, generate 10 million random values from the Gaussian distribution, and measure which is faster.
2.2.18 Dynamic histogram. Suppose that the standard input stream is a sequence of double values. Write a program that takes an integer \( n \) and two double values \( l_0 \) and \( h_i \) from the command line and uses StdStats to plot a histogram of the count of the numbers in the standard input stream that fall in each of the \( n \) intervals defined by dividing \((l_0, h_i)\) into \( n \) equal-sized intervals. Use your program to add code to your solution to Exercise 2.2.3 to plot a histogram of the distribution of the numbers produced by each method, taking \( n \) from the command line.

2.2.19 Stress test. Develop a client that does stress testing for StdStats. Work with a classmate, with one person writing code and the other testing it.

2.2.20 Gambler’s ruin. Develop a StdRandom client to study the gambler’s ruin problem (see Program 1.3.8 and Exercise 1.3.24–25). Note: Defining a static method for the experiment is more difficult than for Bernoulli because you cannot return two values.

2.2.21 IFS. Experiment with various inputs to IFS to create patterns of your own design like the Sierpinski triangle, the Barnsley fern, or the other examples in the table in the text. You might begin by experimenting with minor modifications to the given inputs.

2.2.22 IFS matrix implementation. Write a version of IFS that uses the static method multiply() from Matrix (see Exercise 2.2.12) instead of the equations that compute the new values of \( x_0 \) and \( y_0 \).

2.2.23 Library for properties of integers. Develop a library based on the functions that we have considered in this book for computing properties of integers. Include functions for determining whether a given integer is prime; determining whether two integers are relatively prime; computing all the factors of a given integer; computing the greatest common divisor and least common multiple of two integers; Euler’s totient function (Exercise 2.1.26); and any other functions that you think might be useful. Include overloaded implementations for long values. Create an API, a client that performs stress testing, and clients that solve several of the exercises earlier in this book.
2.2.24 Music library. Develop a library based on the functions in PlayThatTune (Program 2.1.4) that you can use to write client programs to create and manipulate songs.

2.2.25 Voting machines. Develop a StdRandom client (with appropriate static methods of its own) to study the following problem: Suppose that in a population of 100 million voters, 51% vote for candidate A and 49% vote for candidate B. However, the voting machines are prone to make mistakes, and 5% of the time they produce the wrong answer. Assuming the errors are made independently and at random, is a 5% error rate enough to invalidate the results of a close election? What error rate can be tolerated?

2.2.26 Poker analysis. Write a StdRandom and StdStats client (with appropriate static methods of its own) to estimate the probabilities of getting one pair, two pair, three of a kind, a full house, and a flush in a five-card poker hand via simulation. Divide your program into appropriate static methods and defend your design decisions. Extra credit: Add straight and straight flush to the list of possibilities.

2.2.27 Animated plots. Write a program that takes a command-line argument m and produces a bar graph of the m most recent double values on standard input. Use the same animation technique that we used for BouncingBall (Program 1.5.6): erase, redraw, show, and wait briefly. Each time your program reads a new number, it should redraw the whole bar graph. Since most of the picture does not change as it is redrawn slightly to the left, your program will produce the effect of a fixed-size window dynamically sliding over the input values. Use your program to plot a huge time-variant data file, such as stock prices.

2.2.28 Array plot library. Develop your own plot methods that improve upon those in StdStats. Be creative! Try to make a plotting library that you think will be useful for some application in the future.
2.3 Recursion

The idea of calling one function from another immediately suggests the possibility of a function calling itself. The function-call mechanism in Java and most modern programming languages supports this possibility, which is known as recursion. In this section, we will study examples of elegant and efficient recursive solutions to a variety of problems. Recursion is a powerful programming technique that we use often in this book. Recursive programs are often more compact and easier to understand than their nonrecursive counterparts. Few programmers become sufficiently comfortable with recursion to use it in everyday code, but solving a problem with an elegantly crafted recursive program is a satisfying experience that is certainly accessible to every programmer (even you!).

Recursion is much more than a programming technique. In many settings, it is a useful way to describe the natural world. For example, the recursive tree (to the left) resembles a real tree, and has a natural recursive description. Many, many phenomena are well explained by recursive models. In particular, recursion plays a central role in computer science. It provides a simple computational model that embraces everything that can be computed with any computer; it helps us to organize and to analyze programs; and it is the key to numerous critically important computational applications, ranging from combinatorial search to tree data structures that support information processing to the fast Fourier transform for signal processing.

One important reason to embrace recursion is that it provides a straightforward way to build simple mathematical models that we can use to prove important facts about our programs. The proof technique that we use to do so is known as mathematical induction. Generally, we avoid going into the details of mathematical proofs in this book, but you will see in this section that it is worthwhile to understand that point of view and make the effort to convince yourself that recursive programs have the intended effect.
2.3 Recursion

The idea of recursion is very seductive from another immediately suggests the possibility of a function calling itself. The function-call mechanism in Python and in most modern programming languages supports this possibility, which is known as recursion. In this section, we will study examples of elegant and efficient recursive solutions to a variety of problems. Once you get used to the idea, you will see that recursion is a powerful general-purpose programming technique with many attractive properties. It is a tool that we use often in this book. Recursive programs are often more compact and easier to understand than their nonrecursive counterparts. Few programmers become sufficiently comfortable with recursion to use it in everyday code, but solving a problem with an elegantly crafted recursive program is a satisfying experience that is certainly accessible to every programmer (even you!).

A recursive image

Recursion is much more than a programming technique. In many settings, it is a useful way to describe the natural world. For example, the recursive tree (to the left) resembles a real tree, and has a natural recursive description. Many, many phenomena are well explained by recursive descriptions. In particular, recursion plays a central role in computer science. It provides a simple computational model that showcases recursion that can be computed with any computer, such as recursive search and analysis programs, and it is the key to numerous critically important computational applications, ranging from combinatorial search to tree data structures that support information processing in the fast fashion. We now turn to the fundamentals of recursion.

One important reason to embrace recursion is that it provides a straightforward way to build simple mathematical models that we can use to prove important facts about our programs. The proof technique that we use to do so is known as mathematical induction. Generally, we avoid going into the details of mathematical induction in this book, but you will see in this section that it is worthwhile to understand the point of it, and to make the effort to continue yourself that recursive programs have the intended effect.
**Your first recursive program** The “Hello, World” for recursion is the factorial function, defined for positive integers \( n \) by the equation

\[
 n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1
\]

In other words, \( n! \) is the product of the positive integers less than or equal to \( n \). Now, \( n! \) is easy to compute with a for loop, but an even easier method is to use the following recursive function:

```java
public static long factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

This function calls itself. The implementation clearly produces the desired effect. You can persuade yourself that it does so by noting that \( \text{factorial()} \) returns \( 1 = 1! \) when \( n \) is 1 and that if it properly computes the value

\[
(n-1)! = (n-1) \times (n-2) \times \ldots \times 2 \times 1
\]

then it properly computes the value

\[
n! = n \times (n-1)!
= n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1
\]

To compute \( \text{factorial(5)} \), the recursive function multiplies 5 by \( \text{factorial(4)} \); to compute \( \text{factorial(4)} \), it multiplies 4 by \( \text{factorial(3)} \); and so forth. This process is repeated until calling \( \text{factorial(1)} \), which directly returns the value 1. We can trace this computation in precisely the same way that we trace any sequence of function calls. Since we treat all of the calls as being independent copies of the code, the fact that they are recursive is immaterial.

Our \( \text{factorial()} \) implementation exhibits the two main components that are required for every recursive function. First, the base case returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For \( \text{factorial()} \), the base case is \( n = 1 \). Second, the reduction step is the central part of a recursive function-call trace for \( \text{factorial(5)} \)

```
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
    return 1
    return 2*1 = 2
    return 3*2 = 6
    return 4*6 = 24
    return 5*24 = 120
```

Function-call trace for \( \text{factorial(5)} \)
function. It relates the function at one (or more) arguments to
the function evaluated at one (or more) other arguments. For
factorial(), the reduction step is $n \times \text{factorial}(n-1)$. All re-
cursive functions must have these two components. Furthermore,
the sequence of argument values must converge to the base case.
For factorial(), the value of $n$ decreases by 1 for each call, so
the sequence of argument values converges to the base case $n = 1$.

Tiny programs such as factorial() perhaps become
slightly clearer if we put the reduction step in an else clause.
However, adopting this convention for every recursive program
would unnecessarily complicate larger programs because it
would involve putting most of the code (for the reduction step)
within curly braces after the else. Instead, we adopt the conven-
tion of always putting the base case as the first statement, end-
ing with a return, and then devoting the rest of the code to the
reduction step.

The factorial() implementation itself is not particularly
useful in practice because $n!$ grows so quickly that the multiplica-
tion will overflow a long and produce incorrect answers for $n > 20$. But the same
technique is effective for computing all sorts of functions. For example, the recur-
sive function

public static double harmonic(int n)
{
    if (n == 1) return 1.0;
    return harmonic(n-1) + 1.0/n;
}

computes the $n$th harmonic numbers (see Program 1.3.5) when $n$ is small, based
on the following equations:

\[
H_n = 1 + 1/2 + \ldots + 1/n \\
= (1 + 1/2 + \ldots + 1/(n-1)) + 1/n \\
= H_{n-1} + 1/n
\]

Indeed, this same approach is effective for computing, with only a few lines of code,
the value of any finite sum (or product) for which you have a compact formula.
Recursive functions like these are just loops in disguise, but recursion can help us
better understand the underlying computation.

\[
\begin{align*}
1 & \quad 1 \\
2 & \quad 2 \\
3 & \quad 6 \\
4 & \quad 24 \\
5 & \quad 120 \\
6 & \quad 720 \\
7 & \quad 5040 \\
8 & \quad 40320 \\
9 & \quad 362880 \\
10 & \quad 3628800 \\
11 & \quad 39916800 \\
12 & \quad 479001600 \\
13 & \quad 6227020800 \\
14 & \quad 87178291200 \\
15 & \quad 1307674368000 \\
16 & \quad 20922789888000 \\
17 & \quad 355687428096000 \\
18 & \quad 6402373705728000 \\
19 & \quad 12164510408832000 \\
20 & \quad 2432902008176640000
\end{align*}
\]
Mathematical induction  Recursive programming is directly related to *mathematical induction*, a technique that is widely used for proving facts about the natural numbers.

Proving that a statement involving an integer \( n \) is true for infinitely many values of \( n \) by mathematical induction involves the following two steps:

- The **base case**: prove the statement true for some specific value or values of \( n \) (usually 0 or 1).
- The **induction step** (the central part of the proof): assume the statement to be true for all positive integers less than \( n \), then use that fact to prove it true for \( n \).

Such a proof suffices to show that the statement is true for infinitely many values of \( n \): we can start at the base case, and use our proof to establish that the statement is true for each larger value of \( n \), one by one.

Everyone’s first induction proof is to demonstrate that the sum of the positive integers less than or equal to \( n \) is given by the formula \( n \frac{(n + 1)}{2} \). That is, we wish to prove that the following equation is valid for all \( n \geq 1 \):

\[
1 + 2 + 3 \ldots + (n-1) + n = n \frac{(n + 1)}{2}
\]

The equation is certainly true for \( n = 1 \) (base case) because \( 1 = 1 \frac{(1 + 1)}{2} \). If we assume it to be true for all positive integers less than \( n \), then, in particular, it is true for \( n-1 \), so

\[
1 + 2 + 3 \ldots + (n-1) = (n-1) \frac{n}{2}
\]

and we can add \( n \) to both sides of this equation and simplify to get the desired equation (induction step).

Every time we write a recursive program, we need mathematical induction to be convinced that the program has the desired effect. The correspondence between induction and recursion is self-evident. The difference in nomenclature indicates a difference in outlook: in a recursive program, our outlook is to get a computation done by reducing to a smaller problem, so we use the term *reduction step*; in an induction proof, our outlook is to establish the truth of the statement for larger problems, so we use the term *induction step*.

When we write recursive programs we usually do not write down a full formal proof that they produce the desired result, but we are always dependent upon the existence of such a proof. We often appeal to an informal induction proof to convince ourselves that a recursive program operates as expected. For example, we just discussed an informal proof to become convinced that `factorial()` computes the product of the positive integers less than or equal to \( n \).
The greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68. You may recall learning about the greatest common divisor when you learned to reduce fractions. For example, we can simplify 68/102 to 2/3 by dividing both numerator and denominator by 34, their gcd. Finding the gcd of huge numbers is an important problem that arises in many commercial applications, including the famous RSA cryptosystem.

We can efficiently compute the gcd using the following property, which holds for positive integers \( p \) and \( q \):

If \( p > q \), the gcd of \( p \) and \( q \) is the same as the gcd of \( q \) and \( p \mod q \).
To convince yourself of this fact, first note that the gcd of \( p \) and \( q \) is the same as the gcd of \( q \) and \( p - q \), because a number divides both \( p \) and \( q \) if and only if it divides both \( q \) and \( p - q \). By the same argument, \( q \) and \( p - 2q, q \) and \( p - 3q \), and so forth have the same gcd, and one way to compute \( p \% q \) is to subtract \( q \) from \( p \) until getting a number less than \( q \).

The static method \( \text{gcd()} \) in \texttt{Euclid} (Program 2.3.1) is a compact recursive function whose reduction step is based on this property. The base case is when \( q \) is 0, with \( \text{gcd}(p, 0) = p \). To see that the reduction step converges to the base case, observe that the second argument value strictly decreases in each recursive call since \( p \% q < q \). If \( p < q \), the first recursive call effectively switches the order of the two arguments. In fact, the second argument value decreases by at least a factor of 2 for every second recursive call, so the sequence of argument values quickly converges to the base case (see Exercise 2.3.11). This recursive solution to the problem of computing the greatest common divisor is known as \textit{Euclid's algorithm} and is one of the oldest known algorithms—it is more than 2,000 years old.

\textbf{Towers of Hanoi} No discussion of recursion would be complete without the ancient \textit{towers of Hanoi} problem. In this problem, we have three poles and \( n \) discs that fit onto the poles. The discs differ in size and are initially stacked on one of the poles, in order from largest (disc \( n \)) at the bottom to smallest (disc 1) at the top. The task is to move all \( n \) discs to another pole, while obeying the following rules:

- Move only one disc at a time.
- Never place a larger disc on a smaller one.

One legend says that the world will end when a certain group of monks accomplishes this task in a temple with 64 golden discs on three diamond needles. But how can the monks accomplish the task at all, playing by the rules?

To solve the problem, our goal is to issue a sequence of instructions for moving the discs. We assume that the poles are arranged in a row, and that each instruction to move a disc specifies its number and whether to move it left or right. If a disc is on the left pole, an instruction to move left means to wrap to the right pole; if a disc is on the right pole, an instruction to move right means to wrap to the left pole. When the discs are all on one pole, there are two possible moves (move the smallest disc left or right); otherwise, there are three possible moves...
(move the smallest disc left or right, or make the one legal move involving the other two poles). Choosing among these possibilities on each move to achieve the goal is a challenge that requires a plan. Recursion provides just the plan that we need, based on the following idea: first we move the top \( n-1 \) discs to an empty pole, then we move the largest disc to the other empty pole (where it does not interfere with the smaller ones), and then we complete the job by moving the \( n-1 \) discs onto the largest disc.

\( \text{TowersOfHanoi} \) (Program 2.3.2) is a direct implementation of this recursive strategy. It takes a command-line argument \( n \) and prints the solution to the towers of Hanoi problem on \( n \) discs. The recursive function \( \text{moves()} \) prints the sequence of moves to move the stack of discs to the left (if the argument \( \text{left} \) is \( \text{true} \)) or to the right (if \( \text{left} \) is \( \text{false} \)). It does so exactly according to the plan just described.

**Function-call trees** To better understand the behavior of modular programs that have multiple recursive calls (such as \( \text{TowersOfHanoi} \)), we use a visual representation known as a function-call tree. Specifically, we represent each method call as a tree node, depicted as a circle labeled with the values of the arguments for that call. Below each tree node, we draw the tree nodes corresponding to each call in that use of the method (in order from left to right) and lines connecting to them. This diagram contains all the information we need to understand the behavior of the program. It contains a tree node for each function call.

We can use function-call trees to understand the behavior of any modular program, but they are particularly useful in exposing the behavior of recursive programs. For example, the tree corresponding to a call to \( \text{move()} \) in \( \text{TowersOfHanoi} \) is easy to construct. Start by drawing a tree node labeled with the values of the command-line arguments. The first argument is the number

\[ \text{Function-call tree for moves(4, true) in TowersOfHanoi} \]
Program 2.3.2  Towers of Hanoi

public class TowersOfHanoi
{
    public static void moves(int n, boolean left)
    {
        if (n == 0) return;
        moves(n-1, !left);
        if (left) StdOut.println(n + " left");
        else      StdOut.println(n + " right");
        moves(n-1, !left);
    }
    public static void main(String[] args)
    {  // Read n, print moves to move n discs left.
       int n = Integer.parseInt(args[0]);
       moves(n, true);
    }
}

The recursive method moves() prints the moves needed to move n discs to the left (if left is true) or to the right (if left is false).
2.3 Recursion

of discs in the pile to be moved (and the label of the disc to actually be moved); the second is the direction to move the disc. For clarity, we depict the direction (a boolean value) as an arrow that points left or right, since that is our interpretation of the value—the direction to move the piece. Then draw two tree nodes below with the number of discs decremented by 1 and the direction switched, and continue doing so until only nodes with labels corresponding to a first argument value 1 have no nodes below them. These nodes correspond to calls on moves() that do not lead to further recursive calls.

Take a moment to study the function-call tree depicted earlier in this section and to compare it with the corresponding function-call trace depicted at right. When you do so, you will see that the recursion tree is just a compact representation of the trace. In particular, reading the node labels from left to right gives the moves needed to solve the problem.

Moreover, when you study the tree, you probably notice several patterns, including the following two:

• Alternate moves involve the smallest disc.
• That disc always moves in the same direction.

These observations are relevant because they give a solution to the problem that does not require recursion (or even a computer): every other move involves the smallest disc (including the first and last), and each intervening move is the only legal move at the time not involving the smallest disc. We can prove that this approach produces the same outcome as the recursive program, using induction. Having started centuries ago without the benefit of a computer, perhaps our monks are using this approach.

Trees are relevant and important in understanding recursion because the tree is a quintessential recursive object. As an abstract mathematical model, trees play an essential role in many applications, and in Chapter 4, we will consider the use of trees as a computational model to structure data for efficient processing.
Exponential time  One advantage of using recursion is that often we can develop mathematical models that allow us to prove important facts about the behavior of recursive programs. For the towers of Hanoi problem, we can estimate the amount of time until the end of the world (assuming that the legend is true). This exercise is important not just because it tells us that the end of the world is quite far off (even if the legend is true), but also because it provides insight that can help us avoid writing programs that will not finish until then.

The mathematical model for the towers of Hanoi problem is simple: if we define the function $T(n)$ to be the number of discs moved by TowersOfHanoi to solve an $n$-disc problem, then the recursive code implies that $T(n)$ must satisfy the following equation:

$$T(n) = 2 \cdot T(n-1) + 1 \text{ for } n > 1, \text{ with } T(1) = 1$$

Such an equation is known in discrete mathematics as a recurrence relation. Recurrence relations naturally arise in the study of recursive programs. We can often use them to derive a closed-form expression for the quantity of interest. For $T(n)$, you may have already guessed from the initial values $T(1) = 1$, $T(2) = 3$, $T(3) = 7$, and $T(4) = 15$ that $T(n) = 2^n - 1$. The recurrence relation provides a way to prove this to be true, by mathematical induction:

- **Base case**: $T(1) = 2^1 - 1 = 1$
- **Induction step**: if $T(n-1) = 2^{n-1} - 1$, $T(n) = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Therefore, by induction, $T(n) = 2^n - 1$ for all $n > 0$. The minimum possible number of moves also satisfies the same recurrence (see Exercise 2.3.11).

Knowing the value of $T(n)$, we can estimate the amount of time required to perform all the moves. If the monks move discs at the rate of one per second, it would take more than one week for them to finish a 20-disc problem, more than 34 years to finish a 30-disc problem, and more than 348 centuries for them to finish a 40-disc problem (assuming that they do not make a mistake). The 64-disc problem would take more than 5.8 billion centuries. The end of the world is likely to be even further off than that because those monks presumably never have had the benefit of using Program 2.3.2, and might not be able to move the discs so rapidly or to figure out so quickly which disc to move next.

Even computers are no match for exponential growth. A computer that can do a billion operations per second will still take centuries to do $2^{64}$ operations, and no computer will ever do $2^{1,000}$ operations, say. The lesson is profound: with recursion, you can easily write simple short programs...
that take exponential time, but they simply will not run to completion when you try to run them for large \(n\). Novices are often skeptical of this basic fact, so it is worth your while to pause now to think about it. To convince yourself that it is true, take the print statements out of `TowersOfHanoi` and run it for increasing values of \(n\) starting at 20. You can easily verify that each time you increase the value of \(n\) by 1, the running time doubles, and you will quickly lose patience waiting for it to finish. If you wait for an hour for some value of \(n\), you will wait more than a day for \(n + 5\), more than a month for \(n + 10\), and more than a century for \(n + 20\) (no one has that much patience). Your computer is just not fast enough to run every short Java program that you write, no matter how simple the program might seem! Beware of programs that might require exponential time.

We are often interested in predicting the running time of our programs. In Section 4.1, we will discuss the use of the same process that we just used to help estimate the running time of other programs.

**Gray codes** The towers of Hanoi problem is no toy. It is intimately related to basic algorithms for manipulating numbers and discrete objects. As an example, we consider Gray codes, a mathematical abstraction with numerous applications.

The playwright Samuel Beckett, perhaps best known for *Waiting for Godot*, wrote a play called *Quad* that had the following property: starting with an empty stage, characters enter and exit one at a time so that each subset of characters on the stage appears exactly once. How did Beckett generate the stage directions for this play?

One way to represent a subset of \(n\) discrete objects is to use a string of \(n\) bits. For Beckett’s problem, we use a 4-bit string, with bits numbered from right to left and a bit value of 1 indicating the character onstage. For example, the string 0 1 0 1 corresponds to the scene with characters 3 and 1 onstage. This representation gives a quick proof of a basic fact: the number of different subsets of \(n\) objects is exactly \(2^n\). *Quad* has four characters, so there are \(2^4 = 16\) different scenes. Our task is to generate the stage directions.

An \(n\)-bit Gray code is a list of the \(2^n\) different \(n\)-bit binary numbers such that each element in the list differs in precisely one bit from its predecessor. Gray codes directly apply to Beckett’s problem because changing the value of a bit from 0 to 1...
corresponds to a character entering the subset onstage; changing a bit from 1 to 0 corresponds to a character exiting the subset.

How do we generate a Gray code? A recursive plan that is very similar to the one that we used for the towers of Hanoi problem is effective. The \( n \)-bit binary-reflected Gray code is defined recursively as follows:

- The \((n-1)\) bit code, with 0 prepended to each word, followed by
- The \((n-1)\) bit code in reverse order, with 1 prepended to each word

The 0-bit code is defined to be empty, so the 1-bit code is 0 followed by 1. From this recursive definition, we can verify by induction that the \( n \)-bit binary reflected Gray code has the required property: adjacent codewords differ in one bit position. It is true by the inductive hypothesis, except possibly for the last codeword in the first half and the first codeword in the second half: this pair differs only in their first bit.

The recursive definition leads, after some careful thought, to the implementation in Beckett (Program 2.3.3) for printing Beckett's stage directions. This program is remarkably similar to TowersOfHanoi. Indeed, except for nomenclature, the only difference is in the values of the second arguments in the recursive calls!

As with the directions in TowersOfHanoi, the enter and exit directions are redundant in Beckett, since exit is issued only when an actor is onstage, and enter is issued only when an actor is not onstage. Indeed, both Beckett and TowersOfHanoi directly involve the ruler function that we considered in one of our first programs (Program 1.2.1).

Without the printing instructions, they both implement a simple recursive function that could allow Ruler to print the values of the ruler function for any value given as a command-line argument.

Gray codes have many applications, ranging from analog-to-digital converters to experimental design. They have been used in pulse code communication, the minimization of logic circuits, and hypercube architectures, and were even proposed to organize books on library shelves.
2.3 Recursion

Program 2.3.3 Gray code

```java
public class Beckett {
    public static void moves(int n, boolean enter) {
        if (n == 0) return;
        moves(n-1, true);
        if (enter) StdOut.println("enter " + n);
        else StdOut.println("exit  " + n);
        moves(n-1, false);
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        moves(n, true);
    }
}
```

This recursive program gives Beckett’s stage instructions (the bit positions that change in a binary-reflected Gray code). The bit position that changes is precisely described by the ruler function, and (of course) each actor alternately enters and exits.

% java Beckett
enter 1
% java Beckett 2
enter 1
enter 2
exit 1
% java Beckett 3
enter 1
enter 2
exit 1
enter 3
enter 1
exit 2
exit 1
% java Beckett 4
enter 1
enter 2
exit 1
enter 3
enter 1
enter 2
exit 1
exit 3
enter 1
exit 2
exit 1
Recursive graphics  Simple recursive drawing schemes can lead to pictures that are remarkably intricate. Recursive drawings not only relate to numerous applications, but also provide an appealing platform for developing a better understanding of properties of recursive functions, because we can watch the process of a recursive figure taking shape.

As a first simple example, consider Htree (Program 2.3.4), which, given a command-line argument \( n \), draws an H-tree of order \( n \), defined as follows: The base case is to draw nothing for \( n = 0 \). The reduction step is to draw, within the unit square

- three lines in the shape of the letter H
- four H-trees of order \( n - 1 \), one centered at each tip of the H

with the additional proviso that the H-trees of order \( n - 1 \) are halved in size.

Drawings like these have many practical applications. For example, consider a cable company that needs to run cable to all of the homes distributed throughout its region. A reasonable strategy is to use an H-tree to get the signal to a suitable number of centers distributed throughout the region, then run cables connecting each home to the nearest center. The same problem is faced by computer designers who want to distribute power or signal throughout an integrated circuit chip.

Though every drawing is in a fixed-size window, H-trees certainly exhibit exponential growth. An H-tree of order \( n \) connects \( 4^n \) centers, so you would be trying to plot more than a million lines with \( n = 10 \), and more than a billion with \( n = 15 \). The program will certainly not finish the drawing with \( n = 30 \).

If you take a moment to run Htree on your computer for a drawing that takes a minute or so to complete, you will, just by watching the drawing progress, have the opportunity to gain substantial insight into the nature of recursive programs, because you can see the order in which the H figures appear and how they form into H-trees. An even more instructive exercise, which derives from the fact that the same drawing results no matter in which order the recursive draw() calls and the StdDraw.line() calls appear, is to observe the effect of rearranging the order of these calls on the order in which the lines appear in the emerging drawing (see Exercise 2.3.14).
2.3 Recursion

Program 2.3.4 Recursive graphics

```java
class Htree {
    public static void draw(int n, double size, double x, double y) {
        // Draw an H-tree centered at x, y of depth n and given size.
        if (n == 0) return;
        double x0 = x - size/2, x1 = x + size/2;
        double y0 = y - size/2, y1 = y + size/2;
        StdDraw.line(x0, y, x1, y);
        StdDraw.line(x0, y0, x0, y1);
        StdDraw.line(x1, y0, x1, y1);
        draw(n-1, size/2, x0, y0);
        draw(n-1, size/2, x0, y1);
        draw(n-1, size/2, x1, y0);
        draw(n-1, size/2, x1, y1);
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        draw(n, 0.5, 0.5, 0.5);
    }
}
```

The function `draw()` draws three lines, each of length `size`, in the shape of the letter H, centered at `(x, y)`. Then, it calls itself recursively for each of the four tips, halving the `size` argument in each call and using an integer argument `n` to control the depth of the recursion.
Brownian bridge  An H-tree is a simple example of a fractal: a geometric shape that can be divided into parts, each of which is (approximately) a reduced-size copy of the original. Fractals are easy to produce with recursive programs, although scientists, mathematicians, and programmers study them from many different points of view. We have already encountered fractals several times in this book—for example, IFS (Program 2.2.3).

The study of fractals plays an important and lasting role in artistic expression, economic analysis, and scientific discovery. Artists and scientists use fractals to build compact models of complex shapes that arise in nature and resist description using conventional geometry, such as clouds, plants, mountains, riverbeds, human skin, and many others. Economists use fractals to model function graphs of economic indicators.

Fractional Brownian motion is a mathematical model for creating realistic fractal models for many naturally rugged shapes. It is used in computational finance and in the study of many natural phenomena, including ocean flows and nerve membranes. Computing the exact fractals specified by the model can be a difficult challenge, but it is not difficult to compute approximations with recursive programs.

Brownian (Program 2.3.5) produces a function graph that approximates a simple example of fractional Brownian motion known as a Brownian bridge and closely related functions. You can think of this graph as a random walk that connects the two points \((x_0, y_0)\) and \((x_1, y_1)\), controlled by a few parameters. The implementation is based on the midpoint displacement method, which is a recursive plan for drawing the plot within the \(x\)-interval \([x_0, x_1]\). The base case (when the length of the interval is smaller than a given tolerance) is to draw a straight line connecting the two endpoints. The reduction case is to divide the interval into two halves, proceeding as follows:

- Compute the midpoint \((x_m, y_m)\) of the interval.
- Add to the \(y\)-coordinate \(y_m\) of the midpoint a random value \(\delta\), drawn from the Gaussian distribution with mean 0 and a given variance.
- Recur on the subintervals, dividing the variance by a given scaling factor \(s\).

The shape of the curve is controlled by two parameters: the volatility (initial value of the variance) controls the distance the function graph strays from the straight
2.3 Recursion

Program 2.3.5  Brownian bridge

```java
public class Brownian {
    public static void curve(double x0, double y0,
                               double x1, double y1,
                               double var, double s)
    {
        if (x1 - x0 < 0.01)
        {
            StdDraw.line(x0, y0, x1, y1);
            return;
        }
        double xm = (x0 + x1) / 2;
        double ym = (y0 + y1) / 2;
        double delta = StdRandom.gaussian(0, Math.sqrt(var));
        curve(x0, y0, xm, ym+delta, var/s, s);
        curve(xm, ym+delta, x1, y1, var/s, s);
    }
    public static void main(String[] args)
    {
        double hurst = Double.parseDouble(args[0]);
        double s = Math.pow(2, 2*hurst);
        curve(0, 0.5, 1.0, 0.5, 0.01, s);
    }
}
```

By adding a small, random Gaussian to a recursive program that would otherwise plot a straight line, we get fractal curves. The command-line argument `hurst`, known as the Hurst exponent, controls the smoothness of the curves.

```text
% java Brownian 1
% java Brownian 0.5
% java Brownian 0.05
```
line connecting the points, and the Hurst exponent controls the smoothness of the curve. We denote the Hurst exponent by $H$ and divide the variance by $2^{2H}$ at each recursive level. When $H$ is $1/2$ (halved at each level), the curve is a Brownian bridge—a continuous version of the gambler’s ruin problem (see Program 1.3.8). When $0 < H < 1/2$, the displacements tend to increase, resulting in a rougher curve. Finally, when $2 > H > 1/2$, the displacements tend to decrease, resulting in a smoother curve. The value $2 - H$ is known as the fractal dimension of the curve.

The volatility and initial endpoints of the interval have to do with scale and positioning. The main() test client in Brownian allows you to experiment with the Hurst exponent. With values larger than $1/2$, you get plots that look something like the horizon in a mountainous landscape; with values smaller than $1/2$, you get plots similar to those you might see for the value of a stock index.

Extending the midpoint displacement method to two dimensions yields fractals known as plasma clouds. To draw a rectangular plasma cloud, we use a recursive plan where the base case is to draw a rectangle of a given color and the reduction step is to draw a plasma cloud in each of the four quadrants with colors that are perturbed from the average with a random Gaussian. Using the same volatility and smoothness controls as in Brownian, we can produce synthetic clouds that are remarkably realistic. We can use the same code to produce synthetic terrain, by interpreting the color value as the altitude. Variants of this scheme are widely used in the entertainment industry to generate background scenery for movies and games.

![Plasma clouds](image.png)
2.3 Recursion

Pitfalls of recursion  By now, you are perhaps persuaded that recursion can help you to write compact and elegant programs. As you begin to craft your own recursive programs, you need to be aware of several common pitfalls that can arise. We have already discussed one of them in some detail (the running time of your program might grow exponentially). Once identified, these problems are generally not difficult to overcome, but you will learn to be very careful to avoid them when writing recursive programs.

Missing base case.  Consider the following recursive function, which is supposed to compute harmonic numbers, but is missing a base case:

```java
public static double harmonic(int n) {
    return harmonic(n-1) + 1.0/n;
}
```

If you run a client that calls this function, it will repeatedly call itself and never return, so your program will never terminate. You probably already have encountered infinite loops, where you invoke your program and nothing happens (or perhaps you get an unending sequence of printed output). With infinite recursion, however, the result is different because the system keeps track of each recursive call (using a mechanism that we will discuss in Section 4.3, based on a data structure known as a stack) and eventually runs out of memory trying to do so. Eventually, Java reports a StackOverflowError at run time. When you write a recursive program, you should always try to convince yourself that it has the desired effect by an informal argument based on mathematical induction. Doing so might uncover a missing base case.

No guarantee of convergence.  Another common problem is to include within a recursive function a recursive call to solve a subproblem that is not smaller than the original problem. For example, the following method goes into an infinite recursive loop for any value of its argument (except 1) because the sequence of argument values does not converge to the base case:

```java
public static double harmonic(int n) {
    if (n == 1) return 1.0;
    return harmonic(n) + 1.0/n;
}
```
Bugs like this one are easy to spot, but subtle versions of the same problem can be harder to identify. You may find several examples in the exercises at the end of this section.

**Excessive memory requirements.** If a function calls itself recursively an excessive number of times before returning, the memory required by Java to keep track of the recursive calls may be prohibitive, resulting in a `StackOverflowError`. To get an idea of how much memory is involved, run a small set of experiments using our recursive function for computing the harmonic numbers for increasing values of $n$:

```java
public static double harmonic(int n)
{
    if (n == 1) return 1.0;
    return harmonic(n-1) + 1.0/n;
}
```

The point at which you get `StackOverflowError` will give you some idea of how much memory Java uses to implement recursion. By contrast, you can run `PROGRAM 1.3.5` to compute $H_n$ for huge $n$ using only a tiny bit of memory.

**Excessive recomputation.** The temptation to write a simple recursive function to solve a problem must always be tempered by the understanding that a function might take exponential time (unnecessarily) due to excessive recomputation. This effect is possible even in the simplest recursive functions, and you certainly need to learn to avoid it. For example, the *Fibonacci sequence*

\[
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots
\]

is defined by the recurrence $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$ with $F_0 = 0$ and $F_1 = 1$. The Fibonacci sequence has many interesting properties and arise in numerous applications. A novice programmer might implement this recursive function to compute numbers in the Fibonacci sequence:

```java
// Warning: this function is spectacularly inefficient.
public static long fibonacci(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```
2.3 Recursion

However, this function is spectacularly inefficient! Novice programmers often refuse to believe this fact, and run code like this expecting that the computer is certainly fast enough to crank out an answer. Go ahead; see if your computer is fast enough to use this function to compute fibonacci(50). To see why it is futile to do so, consider what the function does to compute fibonacci(8) = 21. It first computes fibonacci(7) = 13 and fibonacci(6) = 8. To compute fibonacci(7), it recursively computes fibonacci(6) = 8 again and fibonacci(5) = 5. Things rapidly get worse because both times it computes fibonacci(6), it ignores the fact that it already computed fibonacci(5), and so forth. In fact, the number of times this program computes fibonacci(1) when computing fibonacci(n) is precisely \( F_n \) (see Exercise 2.3.12). The mistake of recomputation is compounded exponentially. As an example, fibonacci(200) makes \( F_{200} > 10^{43} \) recursive calls to fibonacci(1)! No imaginable computer will ever be able to do this many calculations. Beware of programs that might require exponential time. Many calculations that arise and find natural expression as recursive functions fall into this category. Do not fall into the trap of implementing and trying to run them.

Next, we consider a systematic technique known as dynamic programming, an elegant technique for avoiding such problems. The idea is to avoid the excessive recomputation inherent in some recursive functions by saving away the previously computed values for later reuse, instead of constantly recomputing them.

Wrong way to compute Fibonacci numbers
Functions and Modules

Dynamic programming  A general approach to implementing recursive programs, known as dynamic programming, provides effective and elegant solutions to a wide class of problems. The basic idea is to recursively divide a complex problem into a number of simpler subproblems; store the answer to each of these subproblems; and, ultimately, use the stored answers to solve the original problem. By solving each subproblem only once (instead of over and over), this technique avoids a potential exponential blow-up in the running time.

For example, if our original problem is to compute the $n$th Fibonacci number, then it is natural to define $n + 1$ subproblems, where subproblem $i$ is to compute the $i$th Fibonacci number for each $0 \leq i \leq n$. We can solve subproblem $i$ easily if we already know the solutions to smaller subproblems—specifically, subproblems $i-1$ and $i-2$. Moreover, the solution to our original problem is simply the solution to one of the subproblems—subproblem $n$.

Top-down dynamic programming. In top-down dynamic programming, we store or cache the result of each subproblem that we solve, so that the next time we need to solve the same subproblem, we can use the cached values instead of solving the subproblem from scratch. For our Fibonacci example, we use an array $f[]$ to store the Fibonacci numbers that have already been computed. We accomplish this in Java by using a static variable, also known as a class variable or global variable, that is declared outside of any method. This allows us to save information from one function call to the next.

```
public class TopDownFibonacci {
    private static long[] f = new long[92];
    public static long fibonacci(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;
        if (f[n] > 0) return f[n];
        f[n] = fibonacci(n-1) + fibonacci(n-2);
        return f[n];
    }
}
```

Top-down dynamic programming approach for computing Fibonacci numbers

Top-down dynamic programming is also known as memoization because it avoids duplicating work by remembering the results of function calls.
2.3 Recursion

**Bottom-up dynamic programming.** In bottom-up dynamic programming, we compute solutions to all of the subproblems, starting with the “simplest” subproblems and gradually building up solutions to more and more complicated subproblems. To apply bottom-up dynamic programming, we must order the subproblems so that each subsequent subproblem can be solved by combining solutions to subproblems earlier in the order (which have already been solved). For our Fibonacci example, this is easy: solve the subproblems in the order 0, 1, and 2, and so forth. By the time we need to solve subproblem $i$, we have already solved all smaller subproblems—in particular, subproblems $i-1$ and $i-2$.

```java
public static long fibonacci(int n) {
    int[] f = new int[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

When the ordering of the subproblems is clear, and space is available to store all the solutions, bottom-up dynamic programming is a very effective approach.

Next, we consider a more sophisticated application of dynamic programming, where the order of solving the subproblems is not so clear (until you see it). Unlike the problem of computing Fibonacci numbers, this problem would be much more difficult to solve without thinking recursively and also applying a bottom-up dynamic programming approach.

**Longest common subsequence problem.** We consider a fundamental string-processing problem that arises in computational biology and other domains. Given two strings $x$ and $y$, we wish to determine how similar they are. Some examples include comparing two DNA sequences for homology, two English words for spelling, or two Java files for repeated code. One measure of similarity is the length of the longest common subsequence (LCS). If we delete some characters from $x$ and some characters from $y$, and the resulting two strings are equal, we call the resulting string a common subsequence. The LCS problem is to find a common subsequence of two strings that is as long as possible. For example, the LCS of $\text{GGCACCCG}$ and $\text{ACGGCGGATACG}$ is $\text{GGCAACG}$, a string of length 7.
Algorithms to compute the LCS are used in data comparison programs like the `diff` command in Unix, which has been used for decades by programmers wanting to understand differences and similarities in their text files. Similar algorithms play important roles in scientific applications, such as the Smith–Waterman algorithm in computational biology and the Viterbi algorithm in digital communications theory.

**LCS recurrence.** Now we describe a recursive formulation that enables us to find the LCS of two given strings $s$ and $t$. Let $m$ and $n$ be the lengths of $s$ and $t$, respectively. We use the notation $s[i..m)$ to denote the suffix of $s$ starting at index $i$, and $t[j..n)$ to denote the suffix of $t$ starting at index $j$. On the one hand, if $s$ and $t$ begin with the same character, then the LCS of $x$ and $y$ contains that first character. Thus, our problem reduces to finding the LCS of the suffixes $s[i..m)$ and $t[1..n)$. On the other hand, if $s$ and $t$ begin with different characters, both characters cannot be part of a common subsequence, so we can safely discard one or the other. In either case, the problem reduces to finding the LCS of two strings—either $s[0..m)$ and $t[1..n)$ or $s[i..m)$ and $t[0..n)$—one of which is strictly shorter. In general, if we let $opt[i][j]$ denote the length of the LCS of the suffixes $s[i..m)$ and $t[j..n)$, then the following recurrence expresses $opt[i][j]$ in terms of the length of the LCS for shorter suffixes.

$$
\begin{align*}
0 & \quad \text{if } i = m \text{ or } j = n \\
opt[i][j] &= \begin{cases}
\opt[i+1][j+1] + 1 & \text{if } s[i] = t[j] \\
\max(\opt[i][j+1], \opt[i+1][j]) & \text{otherwise}
\end{cases}
\end{align*}
$$

**Dynamic programming solution.** `LongestCommonSubsequence` (Program 2.3.6) begins with a bottom-up dynamic programming approach to solving this recurrence. We maintain a two-dimensional array $opt[i][j]$ that stores the length of the LCS of the suffixes $s[i..m)$ and $t[j..n)$. Initially, the bottom row (the values for $i = m$) and the right column (the values for $j = n$) are 0. These are the initial values. From the recurrence, the order of the rest of the computation is clear: we start with $opt[m][n] = 1$. Then, as long as we decrease either $i$ or $j$ or both, we know that we will have computed what we need to compute $opt[i][j]$, since the two options involve an $opt[][]$ entry with a larger value of $i$ or $j$ or both. The method `lcs()` in Program 2.3.6 computes the elements in $opt[][]$ by filling in values in rows from bottom to top ($i = m-1$ to 0) and from right to left in each row ($j = n-1$ to 0). The alternative choice of filling in values in columns from right to
2.3 Recursion

Program 2.3.6 Longest common subsequence

```java
public class LongestCommonSubsequence {
    public static String lcs(String s, String t) {
        // Compute length of LCS for all subproblems.
        int m = s.length(), n = t.length();
        int[][] opt = new int[m+1][n+1];
        for (int i = m-1; i >= 0; i--)
            for (int j = n-1; j >= 0; j--)
                if (s.charAt(i) == t.charAt(j))
                    opt[i][j] = opt[i+1][j+1] + 1;
                else
                    opt[i][j] = Math.max(opt[i+1][j], opt[i][j+1]);
        // Recover LCS itself.
        String lcs = "";
        int i = 0, j = 0;
        while(i < m && j < n)
            if (s.charAt(i) == t.charAt(j))
                { lcs += s.charAt(i);
                    i++;
                    j++;
                }
            else if (opt[i+1][j] >= opt[i][j+1]) i++;
                else j++;
        return lcs;
    }
    public static void main(String[] args) {
        StdOut.println(lcs(args[0], args[1]));
    }
}
```

The function `lcs()` computes and returns the LCS of two strings `x` and `y` using bottom-up dynamic programming. The method call `s.charAt(i)` returns character `i` of string `s`.

% java LongestCommonSubsequence GGCACCACG ACGGCGGATACG
GGCAACG
left and from bottom to top in each row would work as well. The above diagram has a blue arrow pointing to each entry that indicates which value was used to compute it. (When there is a tie in computing the maximum, both options are shown.) The final challenge is to recover the longest common subsequence itself, not just its length. The key idea is to retrace the steps of the dynamic programming algorithm backward, rediscovering the path of choices (highlighted in gray in the diagram) from opt[0][0] to opt[m][n]. To determine the choice that led to opt[i][j], we consider the three possibilities:

- The character s[i] equals t[j]. In this case, we must have opt[i][j] = opt[i+1][j+1] + 1, and the next character in the LCS is s[i] (or t[j]), so we include the character s[i] (or t[j]) in the LCS and continue tracing back from opt[i+1][j+1].
- The LCS does not contain s[i]. In this case, opt[i][j] = opt[i+1][j] and we continue tracing back from opt[i+1][j].
- The LCS does not contain t[j]. In this case, opt[i][j] = opt[i][j+1] and we continue tracing back from opt[i][j+1].

We begin tracing back at opt[0][0] and continue until we reach opt[m][n]. At each step in the traceback either i increases or j increases (or both), so the process terminates after at most m + n iterations of the while loop.

Longest common subsequence of GGCACCACG and ACGCGGGATACG
Dynamic programming is a fundamental algorithm design paradigm, intimately linked to recursion. If you take later courses in algorithms or operations research, you are sure to learn more about it. The idea of recursion is fundamental in computation, and the idea of avoiding recomputation of values that have been computed before is certainly a natural one. Not all problems immediately lend themselves to a recursive formulation, and not all recursive formulations admit an order of computation that easily avoids recomputation—arranging for both can seem a bit miraculous when one first encounters it, as you have just seen for the LCS problem.

**Perspective**  Programmers who do not use recursion are missing two opportunities. First recursion leads to compact solutions to complex problems. Second, recursive solutions embody an argument that the program operates as anticipated. In the early days of computing, the overhead associated with recursive programs was prohibitive in some systems, and many people avoided recursion. In modern systems like Java, recursion is often the method of choice.

Recursive functions truly illustrate the power of a carefully articulated abstraction. While the concept of a function having the ability to call itself seems absurd to many people at first, the many examples that we have considered are certainly evidence that mastering recursion is essential to understanding and exploiting computation and in understanding the role of computational models in studying natural phenomena.

Recursion has reinforced for us the idea of proving that a program operates as intended. The natural connection between recursion and mathematical induction is essential. For everyday programming, our interest in correctness is to save time and energy tracking down bugs. In modern applications, security and privacy concerns make correctness an essential part of programming. If the programmer cannot be convinced that an application works as intended, how can a user who wants to keep personal data private and secure be so convinced?

Recursion is the last piece in a programming model that served to build much of the computational infrastructure that was developed as computers emerged to take a central role in daily life in the latter part of the 20th century. Programs built from libraries of functions consisting of statements that operate on primitive types of data, conditionals, loops, and function calls (including recursive ones) can solve important problems of all sorts. In the next section, we emphasize this point and review these concepts in the context of a large application. In Chapter 3 and in Chapter 4, we will examine extensions to these basic ideas that embrace the more expansive style of programming that now dominates the computing landscape.
Q&A

Q. Are there situations when iteration is the only option available to address a problem?

A. No, any loop can be replaced by a recursive function, though the recursive version might require excessive memory.

Q. Are there situations when recursion is the only option available to address a problem?

A. No, any recursive function can be replaced by an iterative counterpart. In Section 4.3, we will see how compilers produce code for function calls by using a data structure called a stack.

Q. Which should I prefer, recursion or iteration?

A. Whichever leads to the simpler, more easily understood, or more efficient code.

Q. I get the concern about excessive space and excessive recomputation in recursive code. Anything else to be concerned about?

A. Be extremely wary of creating arrays in recursive code. The amount of space used can pile up very quickly, as can the amount of time required for memory management.
2.3 Recursion

2.3.1 What happens if you call factorial() with a negative value of n? With a large value of, say, 35?

2.3.2 Write a recursive function that takes an integer n as its argument and returns ln (n!).

2.3.3 Give the sequence of integers printed by a call to ex233(6):

```java
public static void ex233(int n)
{
    if (n <= 0) return;
    StdOut.println(n);
    ex233(n-2);
    ex233(n-3);
    StdOut.println(n);
}
```

2.3.4 Give the value of ex234(6):

```java
public static String ex234(int n)
{
    if (n <= 0) return "";
    return ex234(n-3) + n + ex234(n-2) + n;
}
```

2.3.5 Criticize the following recursive function:

```java
public static String ex235(int n)
{
    String s = ex235(n-3) + n + ex235(n-2) + n;
    if (n <= 0) return "";
    return s;
}
```

Answer: The base case will never be reached because the base case appears after the reduction step. A call to ex235(3) will result in calls to ex235(0), ex235(-3), ex235(-6), and so forth until a StackOverflowError.
2.3.6 Given four positive integers $a$, $b$, $c$, and $d$, explain what value is computed by $\text{gcd}(\text{gcd}(a, b), \text{gcd}(c, d))$.

2.3.7 Explain in terms of integers and divisors the effect of the following Euclid-like function:

```java
public static boolean gcdlike(int p, int q)
{
    if (q == 0) return (p == 1);
    return gcdlike(q, p % q);
}
```

2.3.8 Consider the following recursive function:

```java
public static int mystery(int a, int b)
{
    if (b == 0)     return 0;
    if (b % 2 == 0) return mystery(a+a, b/2);
    return mystery(a+a, b/2) + a;
}
```

What are the values of $\text{mystery}(2, 25)$ and $\text{mystery}(3, 11)$? Given positive integers $a$ and $b$, describe what value $\text{mystery}(a, b)$ computes. Then answer the same question, but replace + with * and return 0 with return 1.

2.3.9 Write a recursive program Ruler to plot the subdivisions of a ruler using StdDraw, as in Program 1.2.1.

2.3.10 Solve the following recurrence relations, all with $T(1) = 1$. Assume $n$ is a power of 2.

- $T(n) = T(n/2) + 1$
- $T(n) = 2T(n/2) + 1$
- $T(n) = 2T(n/2) + n$
- $T(n) = 4T(n/2) + 3$

2.3.11 Prove by induction that the minimum possible number of moves needed to solve the towers of Hanoi satisfies the same recurrence as the number of moves used by our recursive solution.
2.3.12 Prove by induction that the recursive program given in the text makes exactly \( F_n \) recursive calls to \( \text{fibonacci}(1) \) when computing \( \text{fibonacci}(n) \).

2.3.13 Prove that the second argument to \( \text{gcd()} \) decreases by at least a factor of 2 for every second recursive call, and then prove that \( \text{gcd}(p, q) \) uses at most \( 2 \log_2 n + 1 \) recursive calls where \( n \) is the larger of \( p \) and \( q \).

2.3.14 Modify \texttt{Htree} (PROGRAM 2.3.4) to animate the drawing of the H-tree. Next, rearrange the order of the recursive calls (and the base case), view the resulting animation, and explain each outcome.
2.3.15 **Binary representation.** Write a program that takes a positive integer \( n \) (in decimal) as a command-line argument and prints its binary representation. Recall, in Program 1.3.7, that we used the method of subtracting out powers of 2. Now, use the following simpler method: repeatedly divide 2 into \( n \) and read the remainders backward. First, write a `while` loop to carry out this computation and print the bits in the wrong order. Then, use recursion to print the bits in the correct order.

2.3.16 **A4 paper.** The width-to-height ratio of paper in the ISO format is the square root of 2 to 1. Format A0 has an area of 1 square meter. Format A1 is A0 cut with a vertical line into two equal halves, A2 is A1 cut with a horizontal line into two halves, and so on. Write a program that takes an integer command-line argument \( n \) and uses `StdDraw` to show how to cut a sheet of A0 paper into \( 2^n \) pieces.

2.3.17 **Permutations.** Write a program `Permutations` that takes an integer command-line argument \( n \) and prints all \( n! \) permutations of the \( n \) letters starting at a (assume that \( n \) is no greater than 26). A permutation of \( n \) elements is one of the \( n! \) possible orderings of the elements. As an example, when \( n = 3 \), you should get the following output (but do not worry about the order in which you enumerate them):

\[
\begin{align*}
&bca \ cba \ cab \ acb \ bac \ abc \\
&bca \ cba \ cab \ acb \ bac \ abc
\end{align*}
\]

2.3.18 **Permutations of size \( k \).** Modify `Permutations` from the previous exercise so that it takes two command-line arguments \( n \) and \( k \), and prints all \( P(n, k) = n! / (n-k)! \) permutations that contain exactly \( k \) of the \( n \) elements. Below is the desired output when \( k = 2 \) and \( n = 4 \) (again, do not worry about the order):

\[
\begin{align*}
&ab \ ac \ ad \ ba \ bc \ bd \ ca \ cb \ cd \ da \ db \ dc
\end{align*}
\]

2.3.19 **Combinations.** Write a program `Combinations` that takes an integer command-line argument \( n \) and prints all \( 2^n \) combinations of any size. A combination is a subset of the \( n \) elements, independent of order. As an example, when \( n = 3 \), you should get the following output:

\[
\begin{align*}
&a \ ab \ abc \ ac \ b \ bc \ c
\end{align*}
\]

Note that your program needs to print the empty string (subset of size 0).
2.3.20 *Combinations of size k.* Modify Combinations from the previous exercise so that it takes two integer command-line arguments \( n \) and \( k \), and prints all \( C(n, k) = \frac{n!}{(k!(n-k)!)} \) combinations of size \( k \). For example, when \( n = 5 \) and \( k = 3 \), you should get the following output:

```
abc abd abe acd ace ade bcd bce bde cde
```

2.3.21 *Hamming distance.* The Hamming distance between two bit strings of length \( n \) is equal to the number of bits in which the two strings differ. Write a program that reads in an integer \( k \) and a bit string \( s \) from the command line, and prints all bit strings that have Hamming distance at most \( k \) from \( s \). For example, if \( k \) is 2 and \( s \) is 0000, then your program should print

```
0011 0101 0110 1001 1010 1100
```

*Hint:* Choose \( k \) of the bits in \( s \) to flip.

2.3.22 *Recursive squares.* Write a program to produce each of the following recursive patterns. The ratio of the sizes of the squares is 2.2:1. To draw a shaded square, draw a filled gray square, then an unfilled black square.

![Recursive squares diagram]

2.3.23 *Pancake flipping.* You have a stack of \( n \) pancakes of varying sizes on a griddle. Your goal is to rearrange the stack in order so that the largest pancake is on the bottom and the smallest one is on top. You are only permitted to flip the top \( k \) pancakes, thereby reversing their order. Devise a recursive scheme to arrange the pancakes in the proper order that uses at most \( 2n - 3 \) flips.
2.3.24 *Gray code.* Modify Beckett ([Program 2.3.3](#)) to print the Gray code (not just the sequence of bit positions that change).

2.3.25 *Towers of Hanoi variant.* Consider the following variant of the towers of Hanoi problem. There are $2n$ discs of increasing size stored on three poles. Initially all of the discs with odd size ($1, 3, ..., 2n-1$) are piled on the left pole from top to bottom in increasing order of size; all of the discs with even size ($2, 4, ..., 2n$) are piled on the right pole. Write a program to provide instructions for moving the odd discs to the right pole and the even discs to the left pole, obeying the same rules as for towers of Hanoi.

2.3.26 *Animated towers of Hanoi.* Use StdDraw to animate a solution to the towers of Hanoi problem, moving the discs at a rate of approximately 1 per second.

2.3.27 *Sierpinski triangles.* Write a recursive program to draw Sierpinski triangles (see [Program 2.2.3](#)). As with Htree, use a command-line argument to control the depth of the recursion.

2.3.28 *Binomial distribution.* Estimate the number of recursive calls that would be used by the code

```java
public static double binomial(int n, int k)
    {
        if ((n == 0) && (k == 0)) return 1.0;
        if ((n < 0) || (k < 0))   return 0.0;
        return (binomial(n-1, k) + binomial(n-1, k-1))/2.0;
    }
```

to compute $\text{binomial}(100, 50)$. Develop a better implementation that is based on dynamic programming. *Hint:* See [Exercise 1.4.41](#).

2.3.29 *Collatz function.* Consider the following recursive function, which is related to a famous unsolved problem in number theory, known as the **Collatz problem**, or the $3n+1$ problem:
2.3 Recursion

```java
public static void collatz(int n)
{
    StdOut.print(n + " ");
    if (n == 1) return;
    if (n % 2 == 0) collatz(n / 2);
    else            collatz(3*n + 1);
}
```

For example, a call to `collatz(7)` prints the sequence

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

as a consequence of 17 recursive calls. Write a program that takes a command-line argument `n` and returns the value of `i < n` for which the number of recursive calls for `collatz(i)` is maximized. The unsolved problem is that no one knows whether the function terminates for all integers (mathematical induction is no help, because one of the recursive calls is for a larger value of the argument).

2.3.30 Brownian island. B. Mandelbrot asked the famous question *How long is the coast of Britain?* Modify Brownian to get a program BrownianIsland that plots Brownian islands, whose coastlines resemble that of Great Britain. The modifications are simple: first, change `curve()` to add a random Gaussian to the `x`-coordinate as well as to the `y`-coordinate; second, change `main()` to draw a curve from the point at the center of the canvas back to itself. Experiment with various values of the parameters to get your program to produce islands with a realistic look.

*Brownian islands with Hurst exponent of 0.76*
2.3.31 Plasma clouds. Write a recursive program to draw plasma clouds, using the method suggested in the text.

2.3.32 A strange function. Consider McCarthy’s 91 function:

```java
public static int mccarthy(int n)
{
    if (n > 100) return n - 10;
    return mccarthy(mccarthy(n+11));
}
```

Determine the value of `mccarthy(50)` without using a computer. Give the number of recursive calls used by `mccarthy()` to compute this result. Prove that the base case is reached for all positive integers `n` or find a value of `n` for which this function goes into an infinite recursive loop.

2.3.33 Recursive tree. Write a program `Tree` that takes a command-line argument `n` and produces the following recursive patterns for `n` equal to 1, 2, 3, 4, and 8.

```
1
\[
\begin{array}{c}
2 \\
\end{array}
\]
\[
\begin{array}{c}
3 \\
\end{array}
\]
\[
\begin{array}{c}
4 \\
\end{array}
\]
\[
\begin{array}{c}
8 \\
\end{array}
\]
```

2.3.34 Longest palindromic subsequence. Write a program `LongestPalindromicSubsequence` that takes a string as a command-line argument and determines the longest subsequence of the string that is a palindrome (the same when read forward or backward). Hint: Compute the longest common subsequence of the string and its reverse.
This page intentionally left blank
2.4 Case Study: Percolation

The programming tools that we have considered to this point allow us to attack all manner of important problems. We conclude our study of functions and modules by considering a case study of developing a program to solve an interesting scientific problem. Our purpose in doing so is to review the basic elements that we have covered, in the context of the various challenges that you might face in solving a specific problem, and to illustrate a programming style that you can apply broadly.

Our example applies a widely applicable computational technique known as *Monte Carlo simulation* to study a natural model known as *percolation*. The term “Monte Carlo simulation” is broadly used to encompass any computational technique that employs randomness to estimate an unknown quantity by performing multiple trials (known as *simulations*). We have used it in several other contexts already—for example, in the gambler’s ruin and coupon collector problems. Rather than develop a complete mathematical model or measure all possible outcomes of an experiment, we rely on the laws of probability.

In this case study we will learn quite a bit about percolation, a model which underlies many natural phenomena. Our focus, however, is on the process of developing modular programs to address computational tasks. We identify subtasks that can be independently addressed, striving to identify the key underlying abstractions and asking ourselves questions such as the following: Is there some specific subtask that would help solve this problem? What are the essential characteristics of this specific subtask? Might a solution that addresses these essential characteristics be useful in solving other problems? Asking such questions pays significant dividend, because they lead us to develop software that is easier to create, debug, and reuse, so that we can more quickly address the main problem of interest.

Programs in this section

| 2.4.1 | Percolation scaffolding | . . . . . . . | 304 |
| 2.4.2 | Vertical percolation detection | . . . . . . . | 306 |
| 2.4.3 | Visualization client | . . . . . . . | 309 |
| 2.4.4 | Percolation probability estimate | . . . . . . . | 311 |
| 2.4.5 | Percolation detection | . . . . . . . | 313 |
| 2.4.6 | Adaptive plot client | . . . . . . . | 316 |
**Percolation**  It is not unusual for local interactions in a system to imply global properties. For example, an electrical engineer might be interested in composite systems consisting of randomly distributed insulating and metallic materials: which fraction of the materials need to be metallic so that the composite system is an electrical conductor? As another example, a geologist might be interested in a porous landscape with water on the surface (or oil below). Under which conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as percolation to model such situations. It has been studied widely, and shown to be an accurate model in a dizzying variety of applications, beyond insulating materials and porous substances to the spread of forest fires and disease epidemics to evolution to the study of the Internet.

For simplicity, we begin by working in two dimensions and model the system as an $n$-by-$n$ grid of sites. Each site is either **blocked** or **open**; open sites are initially **empty**. A **full** site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. If there is a full site in the bottom row, then we say that the system **percolates**. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. For the insulating/metallic materials example, the open sites correspond to metallic materials, so that a system that percolates has a metallic path from top to bottom, with full sites conducting. For the porous substance example, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.

In a famous scientific problem that has been heavily studied for decades, scientists are interested in the following question: if sites are independently set to be open with **site vacancy probability** $p$ (and therefore blocked with probability $1 - p$), what is the probability that the system percolates? No mathematical solution to this problem has yet been derived. Our task is to write computer programs to help study the problem.
Basic scaffolding  To address percolation with a Java program, we face numerous decisions and challenges, and we certainly will end up with much more code than in the short programs that we have considered so far in this book. Our goal is to illustrate an incremental style of programming where we independently develop modules that address parts of the problem, building confidence with a small computational infrastructure of our own design and construction as we proceed.

The first step is to pick a representation of the data. This decision can have substantial impact on the kind of code that we write later, so it is not to be taken lightly. Indeed, it is often the case that we learn something while working with a chosen representation that causes us to scrap it and start all over using a new one.

For percolation, the path to an effective representation is clear: use an $n$-by-$n$ array. Which type of data should we use for each element? One possibility is to use integers, with the convention that 0 indicates an empty site, 1 indicates a blocked site, and 2 indicates a full site. Alternatively, note that we typically describe sites in terms of questions: Is the site open or blocked? Is the site full or empty? This characteristic of the elements suggests that we might use $n$-by-$n$ arrays in which element is either \texttt{true} or \texttt{false}. We refer to such two-dimensional arrays as \textit{boolean} matrices. Using boolean matrices leads to code that is easier to understand than the alternative.

Boolean matrices are fundamental mathematical objects with many applications. Java does not provide direct support for operations on boolean matrices, but we can use the methods in \texttt{StdArrayIO} (see \texttt{PROGRAM 2.2.2}) to read and write them. This choice illustrates a basic principle that often comes up in programming: \textit{the effort required to build a more general tool usually pays dividends}.

Eventually, we will want to work with random data, but we also want to be able to read and write to files because debugging programs with random inputs can be counterproductive. With random data, you get different input each time that you run the program; after fixing a bug, what you want to see is the \textit{same} input that you just used, to check that the fix was effective. Accordingly, it is best to start with some specific cases that we understand, kept in files formatted compatible with \texttt{StdArrayIO} (dimensions followed by 0 and 1 values in row-major order).
2.4 Case Study: Percolation

When you start working on a new problem that involves several files, it is usually worthwhile to create a new folder (directory) to isolate those files from others that you may be working on. For example, we might create a folder named percolation to store all of the files for this case study. To get started, we can implement and debug the basic code for reading and writing percolation systems, create test files, check that the files are compatible with the code, and so forth, before worrying about percolation at all. This type of code, sometimes called scaffolding, is straightforward to implement, but making sure that it is solid at the outset will save us from distraction when approaching the main problem.

Now we can turn to the code for testing whether a boolean matrix represents a system that percolates. Referring to the helpful interpretation in which we can think of the task as simulating what would happen if the top were flooded with water (does it flow to the bottom or not?), our first design decision is that we will want to have a flow() method that takes as an argument a boolean matrix isOpen[][] that specifies which sites are open and returns another boolean matrix isFull[][] that specifies which sites are full. For the moment, we will not worry at all about how to implement this method; we are just deciding how to organize the computation. It is also clear that we will want client code to be able to use a percolates() method that checks whether the array returned by flow() has any full sites on the bottom.

Percolation (Program 2.4.1) summarizes these decisions. It does not perform any interesting computation, but after running and debugging this code we can start thinking about actually solving the problem. A method that performs no computation, such as flow(), is sometimes called a stub. Having this stub allows us to test and debug percolates() and main() in the context in which we will need them. We refer to code like Program 2.4.1 as scaffolding. As with scaffolding that construction workers use when erecting a building, this kind of code provides the support that we need to develop a program. By fully implementing and debugging this code (much, if not all, of which we need, anyway) at the outset, we provide a sound basis for building code to solve the problem at hand. Often, we carry the analogy one step further and remove the scaffolding (or replace it with something better) after the implementation is complete.
Program 2.4.1  Percolation scaffolding

```java
public class Percolation {
    public static boolean[][] flow(boolean[][] isOpen) {
        int n =.isOpen.length;
        boolean[][] isFull = new boolean[n][n];
        // The isFull[][] matrix computation goes here.
        return isFull;
    }

    public static boolean percolates(boolean[][] isOpen) {
        boolean[][] isFull = flow(isOpen);
        int n = isOpen.length;
        for (int j = 0; j < n; j++)
            if (isFull[n-1][j]) return true;
        return false;
    }

    public static void main(String[] args) {
        boolean[][] isOpen = StdArrayIO.readBoolean2D();
        StdArrayIO.print(flow(isOpen));
        StdOut.println(percolates(isOpen));
    }
}
```

To get started with percolation, we implement and debug this code, which handles all the straightforward tasks surrounding the computation. The primary function `flow()` returns a boolean matrix giving the full sites (none, in the placeholder code here). The helper function `percolates()` checks the bottom row of the returned matrix to decide whether the system percolates. The test client `main()` reads a boolean matrix from standard input and prints the result of calling `flow()` and `percolates()` for that matrix.
Vertical percolation Given a boolean matrix that represents the open sites, how do we figure out whether it represents a system that percolates? As we will see later in this section, this computation turns out to be directly related to a fundamental question in computer science. For the moment, we will consider a much simpler version of the problem that we call \textit{vertical percolation}.

The simplification is to restrict attention to vertical connection paths. If such a path connects top to bottom in a system, we say that the system \textit{vertically percolates} along the path (and that the system itself vertically percolates). This restriction is perhaps intuitive if we are talking about sand traveling through cement, but not if we are talking about water traveling through cement or about electrical conductivity. Simple as it is, vertical percolation is a problem that is interesting in its own right because it suggests various mathematical questions. Does the restriction make a significant difference? How many vertical percolation paths do we expect?

Determining the sites that are filled by some path that is connected vertically to the top is a simple calculation. We initialize the top row of our result array from the top row of the percolation system, with full sites corresponding to open ones. Then, moving from top to bottom, we fill in each row of the array by checking the corresponding row of the percolation system. Proceeding from top to bottom, we fill in the rows of \texttt{isFull[][]} to mark as \texttt{true} all elements that correspond to sites in \texttt{isOpen[][]} that are vertically connected to a full site on the previous row. 

\textbf{Program 2.4.2} is an implementation of \texttt{flow()} for \texttt{Percolation} that returns a boolean matrix of full sites (\texttt{true} if connected to the top via a vertical path, \texttt{false} otherwise).

\textbf{Testing} After we become convinced that our code is behaving as planned, we want to run it on a broader variety of test cases and address some of our scientific questions. At this point, our initial scaffolding becomes less useful, as representing large boolean matrices with 0s and 1s on standard input and standard output and maintaining large numbers of test cases quickly becomes unwieldy. Instead,
Program 2.4.2  Vertical percolation detection

```java
public static boolean[][] flow(boolean[][] isOpen)
{
    // Compute full sites for vertical percolation.
    int n = isOpen.length;
    boolean[][] isFull = new boolean[n][n];
    for (int j = 0; j < n; j++)
        isFull[0][j] = isOpen[0][j];
    for (int i = 1; i < n; i++)
        for (int j = 0; j < n; j++)
            isFull[i][j] = isOpen[i][j] && isFull[i-1][j];
    return isFull;
}
```

Substituting this method for the stub in Program 2.4.1 gives a solution to the vertical-only percolation problem that solves our test case as expected (see text).

we want to automatically generate test cases and observe the operation of our code on them, to be sure that it is operating as we expect. Specifically, to gain confidence in our code and to develop a better understanding of percolation, our next goals are to:

- Test our code for large random boolean matrices.
- Estimate the probability that a system percolates for a given $p$.

To accomplish these goals, we need new clients that are slightly more sophisticated than the scaffolding we used to get the program up and running. Our modular programming style is to develop such clients in independent classes without modifying our percolation code at all.
Data visualization. We can work with much bigger problem instances if we use StdDraw for output. The following static method for Percolation allows us to visualize the contents of boolean matrices as a subdivision of the StdDraw canvas into squares, one for each site:

```java
public static void show(boolean[][] a, boolean which) {
    int n = a.length;
    StdDraw.setXscale(-1, n);
    StdDraw.setYscale(-1, n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (a[i][j] == which)
                StdDraw.filledSquare(j, n-i-1, 0.5);
}
```

The second argument which specifies which squares we want to fill—those corresponding to true elements or those corresponding to false elements. This method is a bit of a diversion from the calculation, but pays dividends in its ability to help us visualize large problem instances. Using show() to draw our boolean matrices representing blocked and full sites in different colors gives a compelling visual representation of percolation.

Monte Carlo simulation. We want our code to work properly for any boolean matrix. Moreover, the scientific question of interest involves random boolean matrices. To this end, we add another static method to Percolation:

```java
public static boolean[][] random(int n, double p) {
    boolean[][] a = new boolean[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            a[i][j] = StdRandom.bernoulli(p);
    return a;
}
```

This method generates a random n-by-n boolean matrix of any given size n, each element true with probability p.

Having debugged our code on a few specific test cases, we are ready to test it on random systems. It is possible that such cases may uncover a few more bugs, so some care is in order to check results. However, having debugged our code for a small system, we can proceed with some confidence. It is easier to focus on new bugs after eliminating the obvious bugs.
With these tools, a client for testing our percolation code on a much larger set of trials is straightforward. `PercolationVisualizer` (Program 2.4.3) consists of just a `main()` method that takes `n` and `p` from the command line and displays the result of the percolation flow calculation.

This kind of client is typical. Our eventual goal is to compute an accurate estimate of percolation probabilities, perhaps by running a large number of trials, but this simple tool gives us the opportunity to gain more familiarity with the problem by studying some large cases (while at the same time gaining confidence that our code is working properly). Before reading further, you are encouraged to download and run this code from the booksite to study the percolation process. When you run `PercolationVisualizer` for moderate-size `n` (50 to 100, say) and various `p`, you will immediately be drawn into using this program to try to answer some questions about percolation. Clearly, the system never percolates when `p` is low and always percolates when `p` is very high. How does it behave for intermediate values of `p`? How does the behavior change as `n` increases?

### Estimating probabilities

The next step in our program development process is to write code to estimate the probability that a random system (of size `n` with site vacancy probability `p`) percolates. We refer to this quantity as the *percolation probability*. To estimate its value, we simply run a number of trials. The situation is no different from our study of coin flipping (see Program 2.2.6), but instead of flipping a coin, we generate a random system and check whether it percolates.

`PercolationProbability` (Program 2.4.4) encapsulates this computation in a method `estimate()`, which takes three arguments `n`, `p`, and `trials` and returns an estimate of the probability that an `n`-by-`n` system with site vacancy probability `p` percolates, obtained by generating `trials` random systems and calculating the fraction of them that percolate.

How many trials do we need to obtain an accurate estimate? This question is addressed by basic methods in probability and statistics, which are beyond the scope of this book, but we can get a feeling for the problem with computational experience. With just a few runs of `PercolationProbability`, you can learn that if the site vacancy probability is close to either 0 or 1, then we do not need many trials, but that there are values for which we need as many as 10,000 trials to be able to estimate it within two decimal places. To study the situation in more detail, we might modify `PercolationProbability` to produce output like `Bernoulli` (Program 2.2.6), plotting a histogram of the data points so that we can see the distribution of values (see Exercise 2.4.9).
Program 2.4.3  Visualization client

```java
public class PercolationVisualizer {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        double p = Double.parseDouble(args[1]);
        StdDraw.enableDoubleBuffering();

        // Draw blocked sites in black.
        boolean[][] isOpen = Percolation.random(n, p);
        StdDraw.setPenColor(StdDraw.BLACK);
        Percolation.show(isOpen, false);

        // Draw full sites in blue.
        StdDraw.setPenColor(StdDraw.BOOK_BLUE);
        boolean[][] isFull = Percolation.flow(isOpen);
        Percolation.show(isFull, true);

        StdDraw.show();
    }
}
```

This client takes two command-line argument `n` and `p`, generates an `n`-by-`n` random system with site vacancy probability `p`, determines which sites are full, and draws the result on standard drawing. The diagrams below show the results for vertical percolation.
Using `PercolationProbability.estimate()` represents a giant leap in the amount of computation that we are doing. All of a sudden, it makes sense to run thousands of trials. It would be unwise to try to do so without first having thoroughly debugged our percolation methods. Also, we need to begin to take the time required to complete the computation into account. The basic methodology for doing so is the topic of Section 4.1, but the structure of these programs is sufficiently simple that we can do a quick calculation, which we can verify by running the program. If we perform $T$ trials, each of which involves $n^2$ sites, then the total running time of `PercolationProbability.estimate()` is proportional to $n^2T$. If we increase $T$ by a factor of 10 (to gain more precision), the running time increases by about a factor of 10. If we increase $n$ by a factor of 10 (to study percolation for larger systems), the running time increases by about a factor of 100.

Can we run this program to determine percolation probabilities for a system with billions of sites with several digits of precision? No computer is fast enough to use `PercolationProbability.estimate()` for this purpose. Moreover, in a scientific experiment on percolation, the value of $n$ is likely to be much higher. We can hope to formulate a hypothesis from our simulation that can be tested experimentally on a much larger system, but not to precisely simulate a system that corresponds atom-for-atom with the real world. Simplification of this sort is essential in science.

You are encouraged to download `PercolationProbability` from the book-site to get a feel for both the percolation probabilities and the amount of time required to compute them. When you do so, you are not just learning more about percolation, but are also testing the hypothesis that the models we have just described apply to the running times of our simulations of the percolation process.

What is the probability that a system with site vacancy probability $p$ vertically percolates? Vertical percolation is sufficiently simple that elementary probabilistic models can yield an exact formula for this quantity, which we can validate experimentally with `PercolationProbability`. Since our only reason for studying vertical percolation was an easy starting point around which we could develop supporting software for studying percolation methods, we leave further study of vertical percolation for an exercise (see Exercise 2.4.11) and turn to the main problem.
2.4 Case Study: Percolation

Program 2.4.4 Percolation probability estimate

```java
public class PercolationProbability {
    public static double estimate(int n, double p, int trials) {
        // Generate trials random n-by-n systems; return empirical
        // percolation probability estimate.
        int count = 0;
        for (int t = 0; t < trials; t++) {
            // Generate one random n-by-n boolean matrix.
            boolean[][] isOpen = Percolation.random(n, p);
            if (Percolation.percolates(isOpen)) count++;
        }
        return (double) count / trials;
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        double p = Double.parseDouble(args[1]);
        int trials = Integer.parseInt(args[2]);
        double q = estimate(n, p, trials);
        StdOut.println(q);
    }
}
```

The method `estimate()` generates `trials` random `n`-by-`n` systems with site vacancy probability `p` and computes the fraction of them that percolate. This is a Bernoulli process, like coin flipping (see Program 2.2.6). Increasing the number of `trials` increases the accuracy of the estimate. If `p` is close to 0 or to 1, not many trials are needed to achieve an accurate estimate. The results below are for vertical percolation.

```
% java PercolationProbability 20 0.05 10
0.0
% java PercolationProbability 20 0.95 10
1.0
% java PercolationProbability 20 0.85 10
0.7
% java PercolationProbability 20 0.85 1000
0.564
% java PercolationProbability 40 0.85 100
0.1
```
Recursive solution for percolation  How do we test whether a system percolates in the general case when any path starting at the top and ending at the bottom (not just a vertical one) will do the job?

Remarkably, we can solve this problem with a compact program, based on a classic recursive scheme known as depth-first search. PROGRAM 2.4.5 is an implementation of \( \text{flow}() \) that computes the matrix \( \text{isFull}[i][j] \), based on a recursive four-argument version of \( \text{flow}() \) that takes as arguments the site vacancy matrix \( \text{isOpen}[i][j] \), the current matrix \( \text{isFull}[i][j] \), and a site position specified by a row index \( i \) and a column index \( j \). The base case is a recursive call that just returns (we refer to such a call as a null call), for one of the following reasons:

- Either \( i \) or \( j \) is outside the array bounds.
- The site is blocked \( (\text{isOpen}[i][j] \text{ is false}) \).
- We have already marked the site as full \( (\text{isFull}[i][j] \text{ is true}) \).

The reduction step is to mark the site as filled and issue recursive calls for the site’s four neighbors: \( \text{isOpen}[i+1][j] \), \( \text{isOpen}[i][j+1] \), \( \text{isOpen}[i][j-1] \), and \( \text{isOpen}[i-1][j] \). The one-argument \( \text{flow}() \) calls the recursive method for every site on the top row. The recursion always terminates because each recursive call either is null or marks a new site as full. We can show by an induction-based argument (as usual for recursive programs) that a site is marked as full if and only if it is connected to one of the sites on the top row.

Tracing the operation of \( \text{flow}() \) on a tiny test case is an instructive exercise. You will see that it calls \( \text{flow}() \) for every site that can be reached via a path of open sites from the top row. This example illustrates that simple recursive programs can mask computations that otherwise are quite sophisticated. This method is a special case of the depth-first search algorithm, which has many important applications.
2.4 Case Study: Percolation

Program 2.4.5 Percolation detection

```
public static boolean[][] flow(boolean[][] isOpen)
{  // Fill every site reachable from the top row.
    int n = isOpen.length;
    boolean[][] isFull = new boolean[n][n];
    for (int j = 0; j < n; j++)
        flow(isOpen, isFull, 0, j);
    return isFull;
}
public static void flow(boolean[][] isOpen,
                boolean[][] isFull, int i, int j)
{  // Fill every site reachable from (i, j).
    int n = isFull.length;
    if (i < 0 || i >= n) return;
    if (j < 0 || j >= n) return;
    if (!isOpen[i][j]) return;
    if (isFull[i][j]) return;
    isFull[i][j] = true;
    flow(isOpen, isFull, i+1, j);  // Down.
    flow(isOpen, isFull, i, j+1);  // Right.
    flow(isOpen, isFull, i, j-1);  // Left.
    flow(isOpen, isFull, i-1, j);  // Up.
}
```

Substituting these methods for the stub in Program 2.4.1 gives a depth-first-search-based solution to the percolation problem. The recursive flow() sets to true the element in isFull[][], corresponding to any site that can be reached from isOpen[i][j] via a chain of neighboring open sites. The one-argument flow() calls the recursive method for every site on the top row.

```
% more test8.txt
8 8
0 1 1 1 0 0 0
1 0 0 1 1 1 1
1 1 0 0 1 1 0
0 1 1 0 1 1 1
0 1 1 1 0 1 1
0 1 0 0 0 1 1
1 0 1 0 1 1 1
1 1 1 0 1 0 0

% java Percolation < test8.txt
8 8
0 1 1 1 0 0 0
0 0 1 1 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 1
0 0 0 0 0 1 1 1
0 0 0 0 0 1 0 0
true
```
To avoid conflict with our solution for vertical percolation (PROGRAM 2.4.2), we might rename that class PercolationVertical, making another copy of Percolation (PROGRAM 2.4.1) and substituting the two flow() methods in PROGRAM 2.4.5 for the placeholder flow(). Then, we can visualize and perform experiments with this algorithm with the PercolationVisualizer and PercolationProbability tools that we have developed. If you do so, and try various values for \( n \) and \( p \), you will quickly get a feeling for the situation: the systems always percolate when the site vacancy probability \( p \) is high and never percolate when \( p \) is low, and (particularly as \( n \) increases) there is a value of \( p \) above which the systems (almost) always percolate and below which they (almost) never percolate.

```
P = 0.65
```

```
P = 0.60
```

```
P = 0.55
```

*Percolation is less probable as the site vacancy probability \( p \) decreases*

Having debugged PercolationVisualizer and PercolationProbability on the simple vertical percolation process, we can use them with more confidence to study percolation, and turn quickly to study the scientific problem of interest. Note that if we want to experiment with vertical percolation again, we would need to edit PercolationVisualizer and PercolationProbability to refer to PercolationVertical instead of Percolation, or write other clients of both PercolationVertical and Percolation that run methods in both classes to compare them.

**Adaptive plot** To gain more insight into percolation, the next step in program development is to write a program that plots the percolation probability as a function of the site vacancy probability \( p \) for a given value of \( n \). Perhaps the best way to produce such a plot is to first derive a mathematical equation for the function, and then use that equation to make the plot. For percolation, however, no one has been able to derive such an equation, so the next option is to use the Monte Carlo method: run simulations and plot the results.
Immediately, we are faced with numerous decisions. For how many values of \( p \) should we compute an estimate of the percolation probability? Which values of \( p \) should we choose? How much precision should we aim for in these calculations? These decisions constitute an experimental design problem. Much as we might like to instantly produce an accurate rendition of the curve for any given \( n \), the computation cost can be prohibitive. For example, the first thing that comes to mind is to plot, say, 100 to 1,000 equally spaced points, using \texttt{StdStats (PROGRAM 2.2.5)}. But, as you learned from using \texttt{PercolationProbability}, computing a sufficiently precise value of the percolation probability for each point might take several seconds or longer, so the whole plot might take minutes or hours or even longer. Moreover, it is clear that a lot of this computation time is completely wasted, because we know that values for small \( p \) are 0 and values for large \( p \) are 1. We might prefer to spend that time on more precise computations for intermediate \( p \). How should we proceed?

\texttt{PercolationPlot (PROGRAM 2.4.6)} implements a recursive approach with the same structure as \texttt{Brownian (PROGRAM 2.3.5)} that is widely applicable to similar problems. The basic idea is simple: we choose the maximum distance that we wish to allow between values of the \( x \)-coordinate (which we refer to as the \textit{gap tolerance}), the maximum known error that we wish to tolerate in the \( y \)-coordinate (which we refer to as the \textit{error tolerance}), and the number of trials \( T \) per point that we wish to perform. The recursive method draws the plot within a given interval \( [x_0, x_1] \), from \((x_0, y_0)\) to \((x_1, y_1)\). For our problem, the plot is from \((0, 0)\) to \((1, 1)\). The base case (if the distance between \( x_0 \) and \( x_1 \) is less than the gap tolerance, or the distance between the line connecting the two endpoints and the value of the function at the midpoint is less than the error tolerance) is to simply draw a line from \((x_0, y_0)\) to \((x_1, y_1)\). The reduction step is to (recursively) plot the two halves of the curve, from \((x_0, y_0)\) to \((x_m, f(x_m))\) and from \((x_m, f(x_m))\) to \((x_1, y_1)\).

The code in \texttt{PercolationPlot} is relatively simple and produces a good-looking curve at relatively low cost. We can use it to study the shape of the curve for various values of \( n \) or choose smaller tolerances to be more confident that the curve is close to the actual values. Precise mathematical statements about quality of approximation can, in principle, be derived, but it is perhaps not appropriate to go into too much detail while exploring and experimenting, since our goal is simply to develop a hypothesis about percolation that can be tested by scientific experimentation.
Program 2.4.6  Adaptive plot client

```java
public class PercolationPlot {
    public static void curve(int n, double x0, double y0, double x1, double y1) {
        // Perform experiments and plot results.
        double gap = 0.01;
        double err = 0.0025;
        int trials = 10000;
        double xm = (x0 + x1)/2;
        double ym = (y0 + y1)/2;
        double fxm = PercolationProbability.estimate(n, xm, trials);
        if ((x1 - x0 < gap || Math.abs(ym - fxm) < err)) {
            StdDraw.line(x0, y0, x1, y1);
            return;
        }
        curve(n, x0, y0, xm, fxm);
        StdDraw.filledCircle(xm, fxm, 0.005);
        curve(n, xm, fxm, x1, y1);
    }

    public static void main(String[] args) {
        // Plot experimental curve for n-by-n percolation system.
        int n = Integer.parseInt(args[0]);
        curve(n, 0.0, 0.0, 1.0, 1.0);
    }
}
```

This recursive program draws a plot of the percolation probability (experimental observations) against the site vacancy probability \( p \) (control variable) for random \( n \)-by-\( n \) systems.

% java PercolationPlot 20

% java PercolationPlot 100
Indeed, the curves produced by `PercolationPlot` immediately confirm the hypothesis that there is a threshold value (about 0.593): if \( p \) is greater than the threshold, then the system almost certainly percolates; if \( p \) is less than the threshold, then the system almost certainly does not percolate. As \( n \) increases, the curve approaches a step function that changes value from 0 to 1 at the threshold. This phenomenon, known as a phase transition, is found in many physical systems.

The simple form of the output of `PROGRAM 2.4.6` masks the huge amount of computation behind it. For example, the curve drawn for \( n = 100 \) has 18 points, each the result of 10,000 trials, with each trial involving \( n^2 \) sites. Generating and testing each site involves a few lines of code, so this plot comes at the cost of executing billions of statements. There are two lessons to be learned from this observation. First, we need to have confidence in any line of code that might be executed billions of times, so our care in developing and debugging code incrementally is justified. Second, although we might be interested in systems that are much larger, we need further study in computer science to be able to handle larger cases—that is, to develop faster algorithms and a framework for knowing their performance characteristics.

With this reuse of all of our software, we can study all sorts of variants on the percolation problem, just by implementing different `flow()` methods. For example, if you leave out the last recursive call in the recursive `flow()` method in `PROGRAM 2.4.5`, it tests for a type of percolation known as directed percolation, where paths that go up are not considered. This model might be important for a situation like a liquid percolating through porous rock, where gravity might play a role, but not for a situation like electrical connectivity. If you run `PercolationPlot` for both methods, will you be able to discern the difference (see Exercise 2.4.10)?

```
PercolationPlot.curve()
PercolationProbability.estimate()
  Percolation.random()
    StdRandom.bernoulli()
      \( n^2 \) times
    StdRandom.bernoulli()
  return
Percolation.percolates()
  flow()
  return
return
T times
Percolation.random()
  StdRandom.bernoulli()
    \( n^2 \) times
  StdRandom.bernoulli()
return
Percolation.percolates()
  flow()
  return
return
  once for each point
PercolationProbability.estimate()
  Percolation.random()
    StdRandom.bernoulli()
      \( n^2 \) times
    StdRandom.bernoulli()
  return
Percolation.percolates()
  flow()
  return
return
T times
Percolation.random()
  StdRandom.bernoulli()
    \( n^2 \) times
  StdRandom.bernoulli()
return
Percolation.percolates()
  flow()
  return
return
return
```

Function-call trace for `PercolationPlot`
To model physical situations such as water flowing through porous substances, we need to use three-dimensional arrays. Is there a similar threshold in the three-dimensional problem? If so, what is its value? Depth-first search is effective for studying this question, though the addition of another dimension requires that we pay even more attention to the computational cost of determining whether a system percolates (see Exercise 2.4.18). Scientists also study more complex lattice structures that are not well modeled by multidimensional arrays—we will see how to model such structures in Section 4.5.

Percolation is interesting to study via *in silico* experimentation because no one has been able to derive the threshold value mathematically for several natural models. The only way that scientists know the value is by using simulations like Percolation. A scientist needs to do experiments to see whether the percolation model reflects what is observed in nature, perhaps through refining the model (for example, using a different lattice structure). Percolation is an example of an increasing number of problems where computer science of the kind described here is an essential part of the scientific process.

**Lessons**

We might have approached the problem of studying percolation by sitting down to design and implement a single program, which probably would run to hundreds of lines, to produce the kind of plots that are drawn by Program 2.4.6. In the early days of computing, programmers had little choice but to work with such programs, and would spend enormous amounts of time isolating bugs and correcting design decisions. With modern programming tools like Java, we can do better, using the incremental modular style of programming presented in this chapter and keeping in mind some of the lessons that we have learned.

**Expect bugs.** Every interesting piece of code that you write is going to have at least one or two bugs, if not many more. By running small pieces of code on small test cases that you understand, you can more easily isolate any bugs and then more easily fix them when you find them. Once debugged, you can depend on using a library as a building block for any client.
2.4 Case Study: Percolation

**Keep modules small.** You can focus attention on at most a few dozen lines of code at a time, so you may as well break your code into small modules as you write it. Some classes that contain libraries of related methods may eventually grow to contain hundreds of lines of code; otherwise, we work with small files.

**Limit interactions.** In a well-designed modular program, most modules should depend on just a few others. In particular, a module that calls a large number of other modules needs to be divided into smaller pieces. Modules that are called by a large number of other modules (you should have only a few) need special attention, because if you do need to make changes in a module’s API, you have to reflect those changes in all its clients.

**Develop code incrementally.** You should run and debug each small module as you implement it. That way, you are never working with more than a few dozen lines of unreliable code at any given time. If you put all your code in one big module, it is difficult to be confident that any of it is free from bugs. Running code early also forces you to think sooner rather than later about I/O formats, the nature of problem instances, and other issues. Experience gained when thinking about such issues and debugging related code makes the code that you develop later in the process more effective.

**Solve an easier problem.** Some working solution is better than no solution, so it is typical to begin by putting together the simplest code that you can craft that solves a given problem, as we did with vertical percolation. This implementation is the first step in a process of continual refinements and improvements as we develop a more complete understanding of the problem by examining a broader variety of test cases and developing support software such as our `PercolationVisualizer` and `PercolationProbability` classes.
Consider a recursive solution. Recursion is an indispensable tool in modern programming that you should learn to trust. If you are not already convinced of this fact by the simplicity and elegance of *Percolation* and *PercolationPlot*, you might wish to try to develop a nonrecursive program for testing whether a system percolates and then reconsider the issue.

Build tools when appropriate. Our visualization method `show()` and random boolean matrix generation method `random()` are certainly useful for many other applications, as is the adaptive plotting method of *PercolationPlot*. Incorporating these methods into appropriate libraries would be simple. It is no more difficult (indeed, perhaps easier) to implement general-purpose methods like these than it would be to implement special-purpose methods for percolation.

Reuse software when possible. Our StdIn, StdRandom, and StdDraw libraries all simplified the process of developing the code in this section, and we were also immediately able to reuse programs such as *PercolationVisualizer*, *PercolationProbability*, and *PercolationPlot* for percolation after developing them for vertical percolation. After you have written a few programs of this kind, you might find yourself developing versions of these programs that you can reuse for other Monte Carlo simulations or other experimental data analysis problems.

The primary purpose of this case study is to convince you that modular programming will take you much further than you could get without it. Although no approach to programming is a panacea, the tools and approach that we have discussed in this section will allow you to attack complex programming tasks that might otherwise be far beyond your reach.

The success of modular programming is only a start. Modern programming systems have a vastly more flexible programming model than the class-as-a-library-of-static-methods model that we have been considering. In the next two chapters, we develop this model, along with many examples that illustrate its utility.
Q. Editing `PercolationVisualizer` and `PercolationProbability` to rename `Percolation` to `PercolationVertical` or whatever method we want to study seems to be a bother. Is there a way to avoid doing so?

A. Yes, this is a key issue to be revisited in Chapter 3. In the meantime, you can keep the implementations in separate subdirectories, but that can get confusing. Advanced Java mechanisms (such as the `classpath`) are also helpful, but they also have their own problems.

Q. That recursive `flow()` method makes me nervous. How can I better understand what it’s doing?

A. Run it for small examples of your own making, instrumented with instructions to print a function-call trace. After a few runs, you will gain confidence that it always marks as full the sites connected to the start site via a chain of neighboring open sites.

Q. Is there a simple nonrecursive approach to identifying the full sites?

A. There are several methods that perform the same basic computation. We will revisit the problem in Section 4.5, where we consider breadth-first search. In the meantime, working on developing a nonrecursive implementation of `flow()` is certain to be an instructive exercise, if you are interested.

Q. `PercolationPlot` (Program 2.4.6) seems to involve a huge amount of computation to produce a simple function graph. Is there some better way?

A. Well, the best would be a simple mathematical formula describing the function, but that has eluded scientists for decades. Until scientists discover such a formula, they must resort to computational experiments like the ones in this section.
2.4.1 Write a program that takes a command-line argument \( n \) and creates an \( n \)-by-\( n \) boolean matrix with the element in row \( i \) and column \( j \) set to true if \( i \) and \( j \) are relatively prime, then shows the matrix on the standard drawing (see Exercise 1.4.16). Then, write a similar program to draw the Hadamard matrix of order \( n \) (see Exercise 1.4.29). Finally, write a program to draw the boolean matrix such that the element in row \( n \) and column \( j \) is set to true if the coefficient of \( x^j \) in \((1 + x)^i\) (binomial coefficient) is odd (see Exercise 1.4.41). You may be surprised at the pattern formed by the third example.

2.4.2 Implement a `print()` method for Percolation that prints 1 for blocked sites, 0 for open sites, and * for full sites.

2.4.3 Give the recursive calls for `flow()` in Program 2.4.5 given the following input:

```
3 3
1 0 1
0 0 0
1 1 0
```

2.4.4 Write a client of Percolation like PercolationVisualizer that does a series of experiments for a value of \( n \) taken from the command line where the site vacancy probability \( p \) increases from 0 to 1 by a given increment (also taken from the command line).

2.4.5 Describe the order in which the sites are marked when Percolation is used on a system with no blocked sites. Which is the last site marked? What is the depth of the recursion?

2.4.6 Experiment with using PercolationPlot to plot various mathematical functions (by replacing the call `PercolationProbability.estimate()` with a different expression that evaluates a mathematical function). Try the function \( f(x) = \sin x + \cos 10x \) to see how the plot adapts to an oscillating curve, and come up with interesting plots for three or four functions of your own choosing.
2.4.7 Modify Percolation to animate the flow computation, showing the sites filling one by one. Check your answer to the previous exercise.

2.4.8 Modify Percolation to compute that maximum depth of the recursion used in the flow calculation. Plot the expected value of that quantity as a function of the site vacancy probability $p$. How does your answer change if the order of the recursive calls is reversed?

2.4.9 Modify PercolationProbability to produce output like that produced by Bernoulli (Program 2.2.6). Extra credit: Use your program to validate the hypothesis that the data obeys a Gaussian distribution.

2.4.10 Create a program PercolationDirected that tests for directed percolation (by leaving off the last recursive call in the recursive `flow()` method in Program 2.4.5, as described in the text), then use PercolationPlot to draw a plot of the directed percolation probability as a function of the site vacancy probability $p$.

2.4.11 Write a client of Percolation and PercolationDirected that takes a site vacancy probability $p$ from the command line and prints an estimate of the probability that a system percolates but does not percolate down. Use enough experiments to get an estimate that is accurate to three decimal places.
**Creative Exercises**

2.4.12 *Vertical percolation.* Show that a system with site vacancy probability $p$ vertically percolates with probability $1 - (1 - p^n)^n$, and use `PercolationProbability` to validate your analysis for various values of $n$.

2.4.13 *Rectangular percolation systems.* Modify the code in this section to allow you to study percolation in rectangular systems. Compare the percolation probability plots of systems whose ratio of width to height is 2 to 1 with those whose ratio is 1 to 2.

2.4.14 *Adaptive plotting.* Modify `PercolationPlot` to take its control parameters (gap tolerance, error tolerance, and number of trials) as command-line arguments. Experiment with various values of the parameters to learn their effect on the quality of the curve and the cost of computing it. Briefly describe your findings.

2.4.15 *Nonrecursive directed percolation.* Write a nonrecursive program that tests for directed percolation by moving from top to bottom as in our vertical percolation code. Base your solution on the following computation: if any site in a contiguous subrow of open sites in the current row is connected to some full site on the previous row, then all of the sites in the subrow become full.

2.4.16 *Fast percolation test.* Modify the recursive `flow()` method in **Program 2.4.5** so that it returns as soon as it finds a site on the bottom row (and fills no more sites). **Hint:** Use an argument `done` that is `true` if the bottom has been hit, `false` otherwise. Give a rough estimate of the performance improvement factor for this change when running `PercolationPlot`. Use values of $n$ for which the programs run at least a few seconds but not more than a few minutes. Note that the improvement is ineffective unless the first recursive call in `flow()` is for the site below the current site.
2.4.17 **Bond percolation.** Write a modular program for studying percolation under the assumption that the edges of the grid provide connectivity. That is, an edge can be either empty or full, and a system percolates if there is a path consisting of full edges that goes from top to bottom. *Note:* This problem has been solved analytically, so your simulations should validate the hypothesis that the bond percolation threshold approaches 1/2 as \( n \) gets large.

2.4.18 **Percolation in three dimensions.** Implement a class `Percolation3D` and a class `BooleanMatrix3D` (for I/O and random generation) to study percolation in three-dimensional cubes, generalizing the two-dimensional case studied in this section. A percolation system is an \( n \)-by-\( n \)-by-\( n \) cube of sites that are unit cubes, each open with probability \( p \) and blocked with probability \( 1 - p \). Paths can connect an open cube with any open cube that shares a common face (one of six neighbors, except on the boundary). The system percolates if there exists a path connecting any open site on the bottom plane to any open site on the top plane. Use a recursive version of `flow()` like Program 2.4.5, but with eight recursive calls instead of four. Plot the percolation probability versus site vacancy probability \( p \) for as large a value of \( n \) as you can. Be sure to develop your solution incrementally, as emphasized throughout this section.

2.4.19 **Bond percolation on a triangular grid.** Write a modular program for studying bond percolation on a triangular grid, where the system is composed of \( 2n^2 \) equilateral triangles packed together in an \( n \)-by-\( n \) grid of rhombus shapes. Each interior point has six bonds; each point on the edge has four; and each corner point has two.
2.4.20 Game of Life. Implement a class `GameOfLife` that simulates Conway’s Game of Life. Consider a boolean matrix corresponding to a system of cells that we refer to as being either live or dead. The game consists of checking and perhaps updating the value of each cell, depending on the values of its neighbors (the adjacent cells in every direction, including diagonals). Live cells remain live and dead cells remain dead, with the following exceptions:

- A dead cell with exactly three live neighbors becomes live.
- A live cell with exactly one live neighbor becomes dead.
- A live cell with more than three live neighbors becomes dead.

Initialize with a random boolean matrix, or use one of the starting patterns on the booksite. This game has been heavily studied, and relates to foundations of computer science (see the booksite for more information).

---

**Five generations of a glider**
This page intentionally left blank
Charge, 383
Color, 343
Comparable, 545
Complex, 403
Counter, 436–437
data types, 388
designing, 233, 429–431
draw, 361
draw, 675–679
Histogram, 392
implementing, 231
In, 354
libraries, 29, 230–232
modular programming, 432
Out, 355
PathFinder, 683
Picture, 347
Queue, 592
SET, 652
Sketch, 459
spatial vectors, 442–443
ST, 625
StackOfStrings, 568
StdArray, 237
StdAudio, 159
StdDraw, 149, 154
StdIn, 132–133
StdOut, 130
StdRandom, 233
StdStats, 244
StockAccount, 410
Stopwatch, 390
String, 322–333
symbol tables, 625–627
Turtle, 394
Universe, 483
Vector, 443
Approximation algorithms, 852
Arbitrary-size input streams, 137–138
args argument, 7, 208
Arguments
arrays, 207–210
command-line, 7–8, 11, 127
constructors, 333, 385
methods, 30
passing, 207–210, 364–365
printf(), 130–132
static methods, 197
Ariane 5 rocket, 35
Arithmetic
CPU instructions, 1079
floating point numbers, 890
integers, 884–885
operators, 22
TOY machine instructions, 912
Arithmetic logic units (ALUs), 1031
bitwise operations, 1031
inputs, 1031
outputs, 1032
summary, 1032–1033
TOY machine, 910
Arithmetic expression evaluation, 586–589
Arithmetic shifts
bits, 891–892
purpose, 898–899
ArrayIndexOutOfBoundsException
Arrays
aliasing, 516
as arguments, 207–210
assigning, 117
associative, 630
binary searches, 538–539
bitonic, 563
bounds checking, 95
comparing, 117
coupon collector problem, 101–103
decks of cards, 97–100
declarling, 91, 116
default initialization, 93
exchanging values, 96
FIFO queues, 596
hash tables, 636
I/O libraries, 237–238
images, 346–347
immutable types, 439–440
iterable classes, 603
linked structures, 942–944
machine-language, 938–941
memory, 91, 94, 515–517
multidimensional, 111
overview, 90–92
parallel, 411
plotting, 246–248
precomputed values, 99–100
references, 365
resizing, 578–581, 635
as return values, 210
setting values, 95–96
shuffling, 97
side effects, 208–210
Sieve of Eratosthenes, 103–105
stacks, 568–570, 578–581
summary, 115
transposition, 120
two-dimensional.
See Two-dimensional arrays
Arrays.binarySearch(), 559
Arrays.sort(), 559
ArrayStackOfStrings program, 568–570, 603
Arrival rate in M/M/1 queues, 597–598
The Art of Computer Programming
book, 947
ASCII standard, 874, 894–895
Assemblers for TOY machine, 964
Assembly language
description, 930
symbolic names, 981
Index

Assertions, 466–467
Assignments
arrays, 117
chained, 43
compound, 60
description, 17
references, 363
Associative arrays, 630
Associative axiom, 990, 993
Associativity, 17
Asterisks (*)
comments, 9
floating-point numbers, 24–26
integers, 22–23
regular expressions, 724
Audio
plotting sound waves, 249
standard, 155–159
superposition, 211–215
Autoboxing, 457, 585–586
Automatic promotion, 33
Average-case performance, 648
Average magnitude, 164
Average path lengths, 693
Average power, 164
Average program, 137–138
Axioms in Boolean algebra, 990
B
Backslashes (\)
escape sequences, 19
regular expressions, 731
Backward compatibility, 976
Bacon, Kevin, 684
Balanced binary trees, 661
Ball animation, 152–153
Barnsley ferns, 240–243
Base cases
binary search trees, 640
recursion, 264–265, 281
Base classes, 452–453
Base64 encoding, 904
Bases in positional notation, 875
Basic scaffolding, 302–304
Basic statistics, 244–246
Beck exploit, 529
Beckett, Samuel, 273
Beckett program, 274–275
Behavior of objects, 340
Benford’s law, 224
Bernoulli, Jacob, 398
Bernoulli program, 249–250
Best-case performance
binary search trees, 647
insertion sort, 544
Big-O notation, 520–521
BigInteger class, 827, 897–898
Binary adders, 771
Binary digits, 22
Binary frequency count equality,
772–773
Binary incrementers, 769–771
Binary operators, 17
Binary program, 67–69
Binary reflected Gray code, 274
Binary representation
decimal conversions, 877
description, 875
examples, 878–879
hex conversions, 876–877
literals, 891
Binary search trees (BSTs)
implementation, 645–646
insert process, 644–645
machine-language, 942–944
ordered operations, 651
overview, 640–643
performance, 647–648
search process, 643–644
symbol tables, 624–625
traversing, 649–650
Binary searches
binary representation, 536
correctness proof, 535
exception filters, 540
inverting functions, 536–538
overview, 533–534
random web surfer, 176
running time, 535
sorted arrays, 538–539
symbol tables, 635
weighing objects, 540–541
Binary strings, 718–719
Binary trees
balanced, 661
heap-ordered, 661
isomorphic, 661
Binary16 format, 888
BinarySearch program, 538–539
Binomial coefficients, 125
Binomial distributions, 125, 249
Biology
computational, 732–734
DNA computers, 795
genomics application, 336–340
graphs, 672
Bipartite graphs, 682
Bisection searches, 537
Bit-slice memory design, 1054–1056
Bitmapped images, 346
Bitonic arrays, 563
Bits
binary number system, 38, 875
bitwise operations, 891–892
computer dependence, 874
description, 22
logical instructions, 912–913
manipulating, 891–893
memory, 1056
memory size, 513
register, 1051
shifting, 891–892
Bitwise operations
and, 913
arithmetic logic units, 1031
exclusive or, 39, 913
shift, 913
Black–Scholes formula, 222, 565
Blobs, 709
Blocks
statements, 50
variable scope, 200
Bodies
loops, 53
static methods, 196
Body program
memory, 514
N-body simulation, 479–482
Bollobás–Chung graph model, 713
Book indexes, 632–633
Booksite, 2–3
Boole, George, 986
boolean data type
conversion codes, 131–132
description, 14–15
input, 133
memory size, 513
overview, 26–27
Boolean logic
cryptography application, 992–994
description, 27
expressions, 995–996
functions, 987–991, 994–997
overview, 986
Boolean matrices, 302
Boolean satisfiability, 832, 836
boolean equation satisfiability problem, 838
NP-completeness, 844–846, 853–856
Booting, 959–960, 968–969
Bootstrapping, 971
BouncingBall program, 152–153
Bounding boxes for drawings, 146
Bounds
arrays, 95
exponential time, 826
polynomial time, 825
Boxing, 457, 585–586
Box–Muller formula, 47
Breadth-first searches, 683,
687–688, 690, 692
break statement, 74
Bridges, Brownian, 278–280
Brin, Sergey, 184
Brown, Robert, 400
Brownian bridges, 278–280
Brownian motion, 400–401
Brownian program, 278–280
Brute-force algorithm, 535–536
BST program, 645–646
BSTs. See Binary search trees (BSTs)
Buffer overflow
arrays, 95
attacks, 963
Buffering drawings, 151
Bugs
aliasing, 363, 439, 441
overview, 6
testing for, 318
Built-in data types
boolean, 26–27
characters and strings, 19–21
comparisons, 27–29
conversions, 32–35
floating-point numbers, 24–26
integers, 22–24
library methods, 29–32
overview, 14–15
summary, 35–36
terminology, 15–18
Built-in interfaces, 451
Buses, 1034–1036
CPU connections, 1077
program counter connections, 1073–1074
Buzzers, 1048
byte data type, 24
Bytecode
compiling, 589, 788
Java virtual machine, 965
Bytes memory size, 513
C
c conversion specification, 131
Caches
and instruction time, 509
in top-down dynamic programming, 284
Calculators, 908
Callbacks in event-based programming, 451
Calls, 193
chaining, 404
in machine language, 932–933
methods, 30, 197, 340
reverse Polish notation, 591
Canvas, 151
Card decks, arrays for, 97–100
Carets (^)
bitwise operations, 891–892
regular expressions, 731
Carroll, Lewis, 710
Carry bits in adders, 1028
Cartesian representation, 433
Casts, 33–34
Cat program, 356
Cellular automata, 794
Central processing units (CPUs)
bus connections, 1077
control lines, 1077–1078
execute phase, 1079
fetch phase, 1078
instructions, 1079–1080
interfaces, 1076
load address, 1080
modules, 1076
overview, 985
TOY-8 machine, 1076–1080
Centroids, 164
Chained assignments, 43
Chained comparisons, 43
Chaining method calls, 404
Characters and char data type
ASCII, 894
conversion to numbers, 880–881
description, 15
memory size, 513
representing, 894–895
Unicode, 894–895
working with, 19–21
Charge program, 383–389, 515
Checksums
description, 86
formula, 220
Chords, 211
Chromatic scale, 156
Church, Alonso, 790
Church–Turing thesis
extended, 823
overview, 790–791
Turing machine simulation, 798
virtual machines, 958
Ciphers, Kamasutra, 377
Ciphertext, 993
Circuit models
building circuits, 1006–1008
connections, 1002–1004
controlled switches, 1005–1006
conventions, 1004
inputs, 1002–1004
logical design, 1008–1009
outputs, 1002–1004
overview, 1002–1003
wires, 1002–1004
Circuits
combinational.
See Combinational circuits
description, 1010
from gates, 1019–1021
memory, 1054–1057
Circular linked lists, 622
Circular queues, 620
Circular shifts, 375
class extension, 3, 8, 228
ClassDefFoundError, 160
Classes, 4–5
accessing, 227–229
description, 226
implementing, 383–389
inner, 609
modules as, 228
variables, 284
Classifying NP-complete problems, 851
Client code
data types, 430
library methods, 230
Clocks
CPU, 1077–1079
fetch and execute, 1059, 1061
overview, 1058–1059
run and halt inputs, 1060
write control, 1059–1060
Closure operation in REs, 724
Clocks, plasma, 280
Clouds
CMYK color format, 48–49, 371
Code and coding
description, 2
encapsulating, 438
incremental development, 319, 701
reuse, 226, 253, 701
static methods, 205–206
Codebooks, 992
Codons, genes, 336
Coefficients for floating-point
numbers, 889
Coercion, 33
Coin flip, 52–53
Collatz function, 784
Collatz problem, 296–297, 818
Collatz sequence, 948
Collections
description, 566
iterable, 601–605
objects, 582–583
queues. See Queues
stacks. See Stacks
symbol tables. See Symbol Tables
Colons (:)
in Turing machine tapes, 767
foreach statements, 601–602
Color and Color data type
blobs, 709
compatibility, 344
conversion, 48–49
drawings, 150
greyscale, 344
luminance, 343
memory, 514
overview, 341–343
Columns in 2D arrays, 106, 108
Combinational circuits
adders, 1028–1030
ALUs, 1031–1033
buses, 1034–1036
decoders, 1021–1022
demultiplexers, 1022
description, 1007–1008
gates, 1013–1021
layers of abstraction, 1037–1039
modules, 1034
multiplexers, 1023
overview, 1012
sum-of-products, 1024–1027
Comma-separated-value (.csv) files, 358, 360
Command-line arguments, 7–8, 11, 127
Commas (,), arguments, 30
constructors, 333
lambda expressions, 450
methods, 30, 196
two-dimensional arrays, 108
Comments, 5, 9
Commercial data processing, 410–413
Common sequences, longest, 285–288
Commutative axiom, 990
Compact trace format, 770
Comparable interface, 451, 545
Comparable keys
sorting, 546
symbol tables, 626–627
CompareDocuments program, 462–463
compareTo() method
description, 451
String, 332
user defined, 545–546
Comparisons
arrays, 117
chained, 43
objects, 364, 545–546
operators, 27–29
performance, 508–509
sketches, 462–463
Compatibility
backward, 976
Color, 344
Compile-time errors, 6
Compilers
description, 3, 589
optimizing, 814
programs as data, 922–924
purpose, 788
TOY machine, 964–965
Compiling
array values set at, 95–96, 108
classes in, 229
description, 2
programs, 3
Complement operation
bitwise, 891
Boolean algebra, 990
Complete small-world graphs, 694
Complex program
chaining method calls, 404
encapsulation, 433–434
instance variables, 403–404
objects, 404
overview, 402–403
program, 405
Complex numbers, 406–409
Compound assignments, 60
Compression, optimal, 814
Computability
algorithms, 787
halting problem, 808–810
Hilbert’s program, 806–807
liar’s paradox, 807–808
overview, 806
reduction, 811–813
unsolvability proof, 810
unsolvable problems.
See Unsolvability proof
Computation: Finite and Infinite Machines, 780
Computational biology, 732–734
Computational models, 716
Computer animations, 151
Computer speed in performance, 507–508
Computer systems, 1094–1095
Computers Ltd.: What They Really Can’t Do book, 780
Computing devices
boolean logic. See Boolean logic circuit models.
See Circuit models
combational circuits. See Combinational circuits
digital. See Digital devices overview, 985
sequential circuits.
See Sequential circuits
Computing machines
machine-language programming. See Machine-language programming
overview, 873
representing information. See Representing information
TOY. See TOY machine
Computing sketches, 459–460
Concatenation
files, 356
strings, 19–20, 723–724
Concert A, 155
Concordances, 659
Conditionals and loops, 50
applications, 64–73
break statement, 74
continue statement, 74
do-while loops, 75
examples, 61
for loops, 59–61
if statement, 50–53
infinite loops, 76
miscellaneous, 74–75
in modular programming, 227–228
nesting, 62–64
performance analysis, 500, 510
static methods, 193–195
summary, 77
switch statement, 74–75
TOY machine, 913, 918–921
while loops, 53–59
Connected components, 709
Connecting programs, 141
Connections
buses, 1034
circuit models, 1002–1004
CPU, 1077
power source, 1003–1004
program counters, 1073–1075
Constant order of growth, 503
Constants, 16
Constructors
data types, 384–385
String, 333
Containing symbol table keys, 624
Context-free languages, 755
Continue statements, 74
Contracts
APIs, 230–231
design by contract, 465–467
interface, 446–447
machine-language, 932
Control characters, 894
Control circuit
CPU, 1078
execute signals, 1082–1083
fetch signals, 1080, 1082–1083
overview, 1080
Control flow
conditionals and loops.
See Conditionals and loops
static method calls, 193–195
Control lines
CPU, 1077–1080
memory bits, 1056
multiplexers, 1019–1020
program counters, 1074–1075
register bits, 1051
Controlled switches, 1002–1003, 1005–1006
Conversion codes, 131–132
Conversion specifications, 130–131
Conversions
casts, 33–34
color, 48–49
data types, 339
decimal to binary, 877
explicit, 34–35
hex and binary, 876–877
implicit, 33
numbers, 21, 67–69
overview, 32
strings, 21, 453, 880–881
Convert program, 880–882
Conway, John, 326, 794
Cook, Stephen, 840, 845
Cook–Levin theorem, 844–845, 847
Cook reduction, 841
Coordinates
drawing, 144–146
images, 347
polar, 47
Corner cases, 236
Cosine similarity measure, 462
Cost of immutable types, 440
Coulomb’s law, 383
Counter circuits, 1008
Counter machines, 794
Counter program, 436–437
Coupon collector problem, 101–103
Coupon program, 206
CouponCollector program, 101–103, 205
CPUs. See Central processing units (CPUs)
Craps game, 259
Cray, Seymour, 971
Crichton, Michael, 424
Cross-coupled NOR gates, 1050
Cross-coupled switches, 1049
Cross products of vectors, 472
Cryptographic keys, 992
Cryptography application, 992–994
Cryptosystems, 992–993
<Ctrl-C> keys, 76
<Ctrl-D> keys, 137
<Ctrl-Z> keys, 137
Cubic order of growth, 505–508
Cumulative distribution function, 202–203
Curly braces ({}), regular expressions, 724, 731
statements, 5, 78–79
static methods, 196
two-dimensional arrays, 108
Curves
Brownian bridges, 278–280
Dragon, 49, 424
Koch, 397
space-filling, 425
spirals, 398–399
Cycles per second, 155
D
dantzig, George, 831
Data abstraction, 329, 382
Data as instructions, 922–924
Data compression, 814
Data-driven code, 141, 171, 184
Data mining example, 458–459
Data paths for buses, 1034
Data structures, 493
arrays. See Arrays
binary search trees. See Binary search trees (BSTs)
linked lists, 571–578
queues. See Queues
resource allocation, 606–607
stacks. See Stacks
stock example, 411
summary, 608
symbol tables. See Symbol tables

Data-type design
APIs, 429–431
data mining example, 458–464
design by contract, 465–467
encapsulation, 432–438
immutability, 439–446
subclasiing, 452–457
subtyping, 446–451
overview, 428

Data types
access modifiers, 384
APIs, 383
boolean, 991
built-in. See Built-in data types
classes, 383
Color, 341–345
Complex, 402–405
constructors, 384–385
conversions, 34–35, 339
creating, 382
definitions, 331–335
DrunkenTurtle, 400–401
elements summary, 383
generic, 583–585
Histogram, 392–393
image processing, 346–352
immutable, 364, 439
input and output, 353–362
insertion sorts, 545–548
instance methods, 385–386
instance variables, 384
Koch, 397
Mandelbrot, 406–409
output, 355
overview, 330
reference, 362–369
Spiral, 398–399
StockAccount, 410–413
Stopwatch, 390–391
String. See Strings and String data type
summary, 368
TOY machine, 907
Turtle, 394–396
type safety, 18
variables within methods, 386–388
Data visualization, 307–309
Davis, Martin, 816
Dead Sea Scrolls, 659

Debugging
abstraction layers, 1037
assertions, 466–467
capsulation for, 432
immutable types, 440
incremental, 317, 319
linked lists, 596
modular programming, 251–254
test client main() for, 235
unit testing, 246

Decidability, 786–787

Decimal number system
conversion to binary, 877
description, 38, 875
examples, 878–879
Decision problems, NP, 835

Decks of cards, 97–100

Declaration statements, 15–16
Declaring
arrays, 91, 116
String variables, 333

Decoders, 1021–1022
Decoding numbers, 889
Decrementers, binary, 770–771
Decryption devices, 992

DeDup operation
punched paper tape, 942–944
strings, 652–653

DeDup program, 652–653
Default values
arrays, 93, 106–107
canvas size, 145
ink color, 150
instance variables, 415
Node objects, 572
pen radius, 146

Defensive copies, 441

Defining
functions, 192
interfaces, 446
static methods, 193, 196

Definite integration, 816

Degrees of separation
description, 670
shortest paths, 684–686

DeMorgan’s laws, 989–991, 1014–1015

Demultiplexers, 1022

Denial-of-service attacks, 512

Dependencies in subclasses, 453

Dependency graphs, 252

Deprecated methods, 469

Depth-first searches
vs. breadth-first searches, 690
percolation case study, 312

Deques, 618

Derived classes, 452

Descartes, René, 398

Design
APIs, 233
by contract, 465–467
data types. See Data-type design
### Index

**Deterministic finite-state automata (DFAs)**
- examples, 740–741
- implementation, 741–743
- Kleene’s theorem.
  
  See Kleene’s theorem
- language recognized, 739–740
- NFA equivalence, 749–750
- nondeterminism, 744–748
- operations, 739
- overview, 738
- power limitations, 753–755
- summary, 756
- universal virtual, 788–789

**DFA program**, 742–743

**Diameters of graphs**, 711

**Diamond operators (<>)**, 585

**Dice**
- Sicherman, 259
- simulation, 121

**Dictionary lookup**, 624, 628–632

**Difficult problems**
- intractability, 828–829
- search problems, 837–838

**Digital circuits**, 1013

**Digital devices**, 1070
- control, 1080–1082
- CPU, 1076–1080
- program counters, 1073–1075

**Digital image processing**
- digital images, 346–347
- fade effect, 351–352
- grayscale, 347–349
- overview, 346
- scaling, 349–350

**Digital signal processing**, 155, 158

**Dijkstra, Edsger**
- Dutch-national-flag problem, 564
- goto statements, 926
- two-stack algorithm, 587

**Dijkstra’s algorithm**, 692

**Diophantine**, 816

**Directed graphs**, 711

**Directed percolation**, 317

**Discrete distributions**, 172

**Disjunctive normal forms**, 996–997

**Distances of graph paths**, 683, 687–688

**Distributive axiom**, 990

**Divide-and-conquer approach**
- linearithmic order of growth, 504
  
  mergesort, 550–551, 554

**Division**
- floating-point numbers, 24–26
- integers, 22–23
- polar representation, 433

**DivisorPattern** program, 62–64

**DNA computers**, 795

**DNS (domain name system)**, 629

**do-while loops**, 75

**Documents, searching for**, 464

**Dollar signs ($) in REs**, 731

**Domain name system (DNS)**, 629

**Domains, function**, 192

**Dot products**
- function implementation, 209
- vectors, 92, 442–443

**Double.parseDouble() method**
- calls to, 30–31
- type conversion, 21, 34

**Double buffering drawings**, 151

**double data type**
- conversion codes, 132
- description, 14–15
- input, 133
- memory size, 513
- overview, 24–26

**Double negation identity**, 990

**Double negatives in gates**, 1015–1016

**Double quotes (""")**
- escape sequences, 19
- text, 5, 10

**Doublet game**, 710

**Doubling hypotheses**, 496, 498–499

**DoublingTest program**, 496, 498–499

**Downscaling in image processing**, 349

**Dragon curves**, 49, 424

**Dragon program**, 163

**Draw library**, 361

**Drawings**
- recursive graphics, 276–277
- standard. See Standard drawing

**DrunkenTurtle program**, 400

**DrunkenTurtles program**, 401

**Dumping virtual machines**, 960–961

**Dutch-national-flag problem**, 564

**Dynamic dictionaries**, 628

**Dynamic dispatch**, 448

**Dynamic programming**
- bottom-up, 285
- longest common subsequence, 285–288
- overview, 284
- summary, 289
- top-down, 284

**E**

**Easy problems**
- intractability, 829
- search, 837

**Eavesdroppers**, 992–993

**Eccentricity in vertices**, 711

**Eckert, J. Presper**, 924–925

**Edges**
- graphs, 671, 674
- self-loops and parallel, 676

**EDVAC computer**, 924–925
### Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fair coin flip</td>
<td>52–53</td>
</tr>
<tr>
<td>Falsifiable hypotheses</td>
<td>495</td>
</tr>
<tr>
<td>Fecundity parameter</td>
<td>89</td>
</tr>
<tr>
<td>Feedback circuits</td>
<td>1048–1049</td>
</tr>
<tr>
<td>Fermat's Last Theorem</td>
<td>89, 722</td>
</tr>
<tr>
<td>Ferns, Barnsley</td>
<td>240–243</td>
</tr>
<tr>
<td>Fetch–increment–execute cycle</td>
<td>910–911</td>
</tr>
<tr>
<td>Fibonacci numbers</td>
<td></td>
</tr>
<tr>
<td>formulas</td>
<td>82</td>
</tr>
<tr>
<td>machine language</td>
<td>935–936</td>
</tr>
<tr>
<td>recursion</td>
<td>282–283</td>
</tr>
<tr>
<td>FIFO queues. See First-in first-out (FIFO) queues</td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td></td>
</tr>
<tr>
<td>concatenating and filtering</td>
<td>356</td>
</tr>
<tr>
<td>in I/O</td>
<td>126</td>
</tr>
<tr>
<td>n-body simulation</td>
<td>483</td>
</tr>
<tr>
<td>redirection</td>
<td>139–141</td>
</tr>
<tr>
<td>splitting</td>
<td>360</td>
</tr>
<tr>
<td>stock example</td>
<td>411</td>
</tr>
<tr>
<td>symbol tables</td>
<td>629</td>
</tr>
<tr>
<td>Filled shapes</td>
<td>149</td>
</tr>
<tr>
<td>Filters</td>
<td></td>
</tr>
<tr>
<td>exception</td>
<td>540</td>
</tr>
<tr>
<td>files</td>
<td>356</td>
</tr>
<tr>
<td>image processing</td>
<td>379</td>
</tr>
<tr>
<td>piping</td>
<td>142–143</td>
</tr>
<tr>
<td>standard drawing data</td>
<td>146–147</td>
</tr>
<tr>
<td>standard input</td>
<td>140</td>
</tr>
<tr>
<td>final keyword</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>384</td>
</tr>
<tr>
<td>immutable types</td>
<td>440</td>
</tr>
<tr>
<td>instance variables</td>
<td>404</td>
</tr>
<tr>
<td>Financial systems, graphs for</td>
<td>673</td>
</tr>
<tr>
<td>Finite-state transducers</td>
<td>762</td>
</tr>
<tr>
<td>Finite sums</td>
<td>64–65</td>
</tr>
<tr>
<td>First-in first-out (FIFO) queues</td>
<td></td>
</tr>
<tr>
<td>applications overview</td>
<td>597</td>
</tr>
<tr>
<td>array implementation</td>
<td>596</td>
</tr>
<tr>
<td>linked-list implementation</td>
<td>593</td>
</tr>
<tr>
<td>$M/M/1$, 597–600</td>
<td></td>
</tr>
<tr>
<td>overview</td>
<td>566, 592–593</td>
</tr>
<tr>
<td>Flexibility</td>
<td>702</td>
</tr>
<tr>
<td>Flip program</td>
<td>52–53</td>
</tr>
<tr>
<td>Flip-flops</td>
<td>1049–1050</td>
</tr>
<tr>
<td>float data type</td>
<td>26, 513</td>
</tr>
<tr>
<td>Floating-point numbers</td>
<td></td>
</tr>
<tr>
<td>conversion codes</td>
<td>131–132</td>
</tr>
<tr>
<td>exponents</td>
<td>889</td>
</tr>
<tr>
<td>overview</td>
<td>24–26</td>
</tr>
<tr>
<td>precision</td>
<td>40</td>
</tr>
<tr>
<td>representing</td>
<td>888–890</td>
</tr>
<tr>
<td>storing</td>
<td>40</td>
</tr>
<tr>
<td>Flow of control</td>
<td></td>
</tr>
<tr>
<td>conditionals and loops</td>
<td></td>
</tr>
<tr>
<td>static method calls</td>
<td>193–195</td>
</tr>
<tr>
<td>Flowcharts</td>
<td>51–52</td>
</tr>
<tr>
<td>for loops</td>
<td></td>
</tr>
<tr>
<td>continue statement</td>
<td>74</td>
</tr>
<tr>
<td>examples</td>
<td>61</td>
</tr>
<tr>
<td>nesting</td>
<td>62–64</td>
</tr>
<tr>
<td>working with</td>
<td>59–61</td>
</tr>
<tr>
<td>Foreach statements</td>
<td>601–602</td>
</tr>
<tr>
<td>Formal languages</td>
<td></td>
</tr>
<tr>
<td>abstract machines</td>
<td>737–738</td>
</tr>
<tr>
<td>alphabets</td>
<td>720–721</td>
</tr>
<tr>
<td>binary strings</td>
<td>718–719</td>
</tr>
<tr>
<td>definitions</td>
<td>718–723</td>
</tr>
<tr>
<td>DFAs. See Deterministic finite-state automata (DFAs)</td>
<td></td>
</tr>
<tr>
<td>recognition problem</td>
<td>722</td>
</tr>
<tr>
<td>regular</td>
<td>723–729</td>
</tr>
<tr>
<td>regular expressions. See Regular expressions (REs)</td>
<td></td>
</tr>
<tr>
<td>specification problem</td>
<td>722</td>
</tr>
<tr>
<td>Format, files</td>
<td>237</td>
</tr>
<tr>
<td>Format strings</td>
<td>130–131</td>
</tr>
<tr>
<td>Formatted input</td>
<td>135</td>
</tr>
<tr>
<td>Formatted printing</td>
<td>130–132</td>
</tr>
<tr>
<td>Forth language</td>
<td>590</td>
</tr>
<tr>
<td>Fortran language</td>
<td>1094</td>
</tr>
<tr>
<td>Fourier series</td>
<td>211</td>
</tr>
<tr>
<td>Fractal dimensions</td>
<td>280</td>
</tr>
<tr>
<td>Fractals</td>
<td>278–280</td>
</tr>
<tr>
<td>Fractional Brownian motion</td>
<td>278</td>
</tr>
<tr>
<td>Fractions</td>
<td>889–890</td>
</tr>
<tr>
<td>Fragile base class problem</td>
<td>453</td>
</tr>
<tr>
<td>Freeing memory</td>
<td>367</td>
</tr>
<tr>
<td>Frequencies</td>
<td></td>
</tr>
<tr>
<td>counting</td>
<td>555</td>
</tr>
<tr>
<td>sorting</td>
<td>556</td>
</tr>
<tr>
<td>Zipf's law</td>
<td>556</td>
</tr>
<tr>
<td>FrequencyCount program</td>
<td>555–557</td>
</tr>
<tr>
<td>Fully parenthesized arithmetic expressions</td>
<td>587</td>
</tr>
<tr>
<td>Function calls</td>
<td></td>
</tr>
<tr>
<td>abstraction</td>
<td>590–591</td>
</tr>
<tr>
<td>static methods</td>
<td>197</td>
</tr>
<tr>
<td>traces</td>
<td>195</td>
</tr>
<tr>
<td>trees</td>
<td>269, 271</td>
</tr>
<tr>
<td>Function graphs</td>
<td>148, 248</td>
</tr>
<tr>
<td>Functional interfaces</td>
<td>449</td>
</tr>
<tr>
<td>Functional programming</td>
<td>449</td>
</tr>
<tr>
<td>Functional property of programs</td>
<td>812–813</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>987–991, 994–997</td>
</tr>
<tr>
<td>computing with</td>
<td>449</td>
</tr>
<tr>
<td>defining</td>
<td>192</td>
</tr>
<tr>
<td>inverting</td>
<td>536–538</td>
</tr>
<tr>
<td>iterated function systems</td>
<td>239–243</td>
</tr>
<tr>
<td>libraries. See Libraries</td>
<td></td>
</tr>
<tr>
<td>machine language</td>
<td>931–933</td>
</tr>
<tr>
<td>mathematical</td>
<td>202–204</td>
</tr>
<tr>
<td>modules. See Modules</td>
<td>193–201</td>
</tr>
<tr>
<td>overview</td>
<td>191</td>
</tr>
<tr>
<td>recursive. See Recursion</td>
<td></td>
</tr>
<tr>
<td>static methods</td>
<td>193–201</td>
</tr>
<tr>
<td>tables of</td>
<td>907–908</td>
</tr>
</tbody>
</table>
G
Gambler program, 70–71
Gambler’s ruin simulation, 69–71
Game of Life, 326, 794
Garbage collection, 367, 516
Gardner, Martin, 424
Garey, Michael R., 859
Gates
abstraction layers, 1037
AND, 1014
circuits from, 1019–1021
multiway, 1015–1017
NOR, 1014
NOT, 1013–1014
OR, 1014
overview, 1013
sum-of-products, 1026–1027
summary, 1018–1019
universal sets of, 1045
Gaussian distribution functions
API, 231
cumulative, 202–203
probability density, 202–203
Gaussian elimination, 830
Gaussian program, 203
Gaussian random numbers, 47
General purpose computers, 790
Generalized multiway gates, 1016–1017
Generalized regular expressions, 730–732
Generic types, 583–585
Genomics
application, 336–340
indexing, 634
regular expressions, 727, 732–734
symbol tables, 629
Geometric mean, 162
Geometry
abstraction layers, 1037–1039
gates, 1015–1016
German Enigma code, 717
Get operations
hash tables, 639
symbol tables, 624
Gilbert–Shannon–Reeds model, 125
Glass filters, 379
Global clustering coefficients, 713
Global variables, 284
Glossary of terms, 1097–1101
Gödel, Kurt, 807, 840
Golden ratio, 83
Goldstine, Herman, 925
Gore, Al, 436
Gosper, R., 805
Goto statements, 926
Graph data type, 675–679
Graph program, 676–679
Graphics
recursive, 276–277, 397
turtle, 394–396
Graphs
bipartite, 682
client example, 679–682
connected components, 709
dependency, 252
description, 671
DFAs, 738
diameters, 711
directed, 711
text examples, 695
function, 148, 248
generators, 700
Graph data type, 675–679
grid, 708
isomorphism problem, 859
lessons, 700–702
matching, 713
overview, 670–671
random web surfer, 170
small-world, 693–699
systems examples, 671–674
vertex cover, 828, 834, 842
Gravity, 481
Gray codes, 273–275
Grayscale
Color, 344
image processing, 347–349
Grayscale program, 347–349
Greater than signs (>)
bitwise operations, 891–892
comparisons, 27–29
lambda expressions, 450
redirection, 139–140
Greatest common divisor (gcd)
machine language, 931
recursive algorithm, 267–268
TOY machine, 918–921
grep program, 736
grep tool
filters, 142–143
regular expressions, 734–736
Grid graphs, 708
Guarantees
NP-complete problems, 852
performance, 512, 627
worst-case analysis, 825
H
H-trees of order n, 276–277
Hadamard matrices, 122
Halt instructions
CPU, 1079
TOY machine, 912
Halting problem, 808–810
Hamilton, William, 424
Hamming distances, 295
Handles for pointers, 371
Inner loops
  description, 62
  performance, 500, 510
Inorder tree traversal, 649
Input
  arithmetic logic units, 1031
  array libraries, 237–238
  circuit models, 1002–1004
  clocks, 1060
  command-line arguments, 7
  data types, 353
  demultiplexers, 1022
  file concatenation, 356
  gates, 1013
  insertion sorts, 548–549
  machine-language, 936–938
  multiplexers, 1019–1020
  overview, 126–129
  in performance, 510
  program counters, 1073–1075
  random web surfer, 171
  screen scraping, 357–359
  standard, 132–138
  stream data type, 354–355
  virtual machines, 969–970
Input/off switches, 1005
InputMismatchException, 135
Inserting
  BST nodes, 644–645
  linked list nodes, 573–574
Insertion program, 546–547
Insertion sorts
  data types, 545–548
  input sensitivity, 548–549
  overview, 543–544
  performance, 544–545
InsertionDoublingTest
  program, 548–549
Instance methods
  data types, 385–386
  invoking, 334
  vs. static, 340
Instance variables
  Complex program, 403–404
  data types, 384
  initial values, 415
  Instances of objects, 333
Instruction register (IR), 910
Instructions
  components, 911
  CPU, 1079–1080
  as data, 922–924
  execution time, 509
  instruction sets, 911–913
  parsing, 966–967
  TOY machine, 909
  TOY-8 machine, 1070–1071
Integer linear inequality
  satisfiability, 831, 838, 845
Integer linear programming, 831
  NP-completeness, 846
  vertex cover problem, 842
Integer.parseInt() method
  calls to, 30–31
  type conversion, 21, 23, 34
  strings, 880–882
Integers and int data type
  arithmetic, 884–885
  bitwise operations, 891–892
  conversion codes, 131–132
  description, 14–15
  input, 133–134
  overview, 22–24
Integrals, approximating, 449
Integrated development environments (IDEs), 3
Integration, definite, 816
Interactions between modules, 319
Interactive user input, 135–136
Interface construct, 446
Interfaces
  APIs, 430
  built-in, 451
  circuit models, 1003
  CPU, 1076
  defining, 446
  functional, 450
  gates, 1016–1017
  implementing, 447
  memory, 1054
  multiplexers, 1020
  program counters, 1073
  using, 447–448
Internet DNS, 629–630
Internet Protocol (IP), 435
Interpolation in fade effect, 351
Interpreters
  Evaluate program, 589
  TOY machine, 964
IntOps program, 23
Intractability
  difficult problems, 828–829
  easy problems, 829
  exponential-time algorithms, 826
  main question, 840–841
  NP-completeness. See NP-completeness
  numbers, 827
  overview, 822–824
  path problems, 829
  polynomial-time algorithms, 825–826
  polynomial-time reductions, 841–843
  problem size, 824
  satisfiability, 830–832
  search problems, 833–840
  subset sum problem, 827–828
  vertex cover, 828
  worst case, 825
Index

Introduction to the Theory of Computation book, 780
Invariants in assertions, 467
Inverse permutations, 122
Inverters, 1013–1014
Inverting functions, 536–538
Invoking instance methods, 334
IP (Internet Protocol), 435
IPv4
  vs. IPv6, 435
  number of addresses, 900, 904
IPv6
  vs. IPv4, 435
  number of addresses, 901
IR (instruction register), 910
IR write control line, 1082
ISBN (International Standard Book Number), 86
Isolated vertices in graphs, 703
Isomorphic binary trees, 661
Isomorphism in graphs, 859
Items in collections, 566
Iterable interface, 451, 602
Iterable collections, 601–605
  arrays, 603
  linked lists, 604–605
Queue, 604–605
SET, 652
Stack, 603
Iterated function systems, 239–243
Iterations in BSTs, 650
Iterator interface, 451, 602–605
Java command, 3, 134
  .java extension, 3, 6, 8, 197, 383
Java language
  benefits, 9
  libraries, 1094
  overview, 1–8
Java platform, 2
Java Virtual Machine (JVM)
  description, 3
  overview, 965–966
  as program, 788
Java virtual machines, 429
Johnson, David S., 859
Josephus problem, 619
Julia sets, 427
Jump and link instruction, 931
Jump register instruction, 931
K
  K-ring graphs, 694–695
  K-way multiplexers, 1019–1020
Kamasutra ciphers, 377
Karp, Richard, 845–848
Karp’s reductions
  NP-completeness, 845–848
  polynomial-time, 841
Kevin Bacon game, 684–686
Key-sorted tree traversal, 649
Keys
  BSTs, 640–642, 650
  cryptographic, 992
  immutable, 625
  Kamasutra ciphers, 377
  symbol tables, 624–626, 655
Key-value pairs, 624–626
Kleene, Stephen, 748
Kleene’s theorem
  applications, 753–756
  DFA, NFA, and RE equivalence, 749–752
  overview, 748
  power limitations, 753–756
  proof strategy, 748
  RE recognition, 753
Knuth, Donald
  MIX machine, 947
  optimization, 518
  running time, 496, 501
SAT solvers, 832
Koch program, 397
L
  Ladders, word, 710
  Ladner, R., 859
  Lambda calculus, 790, 794
  Lambda expressions, 450
Languages. See Formal languages;
  Programming languages
  Last-in first-out (LIFO), 566–567
  Lattices in random walks, 112–115
  Layers of abstraction, 1037–1039
  LCS (longest common subsequence), 285–288
  Leading zeros, 883
  Leaf nodes in BSTs, 640
  Leaks, memory, 367, 581
  LeapYear program, 28–29
  Left associativity, 17
  Left shift operations
    bitwise, 891–892
    TOY machine, 913
  Left subtrees, 640
Length
  arrays, 91–92
  graphs paths, 674, 683
  strings, 332
Less than signs (<)
  bitwise operations, 891–892
  comparisons, 27–29
  redirection, 140–141
Let’s Make a Deal simulation, 88
Levin, Leonid, 845
Liar’s paradox, 807–808
Libraries
  APIs, 230–232
  array I/O, 237–238
  clients, 230
  extensible, 452
  Java, 1094
  methods, 29–32
modifying, 255
in modular programming, 227–228, 251–254
modules, 191
overview, 226, 230
random numbers, 232–236
statistics, 244–250
stress testing, 236
unit testing, 235
LIFO (last-in first-out), 566–567
Lights for TOY machine, 916
Lindenmayer systems, 803
Linear algebra for vectors, 442–443
Linear equation satisfiability problem, 830, 839
Linear feedback shift registers (LFSRs), 1000–1001
Linear inequality satisfiability problem, 831, 839
Linear interpolation, 351
Linear order of growth, 504–505, 507–508
Linear programming problem, 831
Linear order of growth, 504–505, 507–508
Linked lists
circular, 622
FIFO queues, 593, 596
hash tables, 636
iterable classes, 604–605
overview, 571–574
stacks, 574–576
summary, 578
symbol tables, 635
traversal, 574, 577
Linked structures. See Binary search trees (BSTs)
LinkedStackOfStrings program, 574–576
Links in BSTs, 640–642
Lipton, R. J., 856
Lissajous, Jules A., 168
Lissajous patterns, 168
Lists, linked. See Linked lists
Literals
array elements, 116
binary and hex, 891
booleans, 26
characters, 18–19
description, 15
floating-point numbers, 24
integers, 22
strings, 19, 334
Little's law, 598
Load address instruction, 1080
Load instructions, 938, 1080
LoadBalance program, 606–607
Local clustering, 693–694
Local variables
vs. instance variables, 384
static methods, 196
Logarithmic order of growth, 503
Logarithmic spirals, 398–399
Logical design, 1008–1009
Logical instructions, 912–913
Logical shifts, 891–892
Logical switches
bus muxes, 1036
demultiplexers, 1022
multiplexers, 1020
Logo language, 400
Loitering condition, 581
Long data type, 24, 513
Long path problems, 829
Longest common subsequence (LCS), 285–288
Longest path problem, 838
LongestCommonSubsequence program, 286–288
Lookup program, 630–632
Loops. See Conditionals and loops
Lost letter, 840
Lower bounds, 826
Luminance, 343–345
Luminance program, 344–345
M
M/M/1 queues, 597–600
MAC addresses, 877
Machine-language programming
arrays, 938–941
benefits, 945
description, 907
functions, 931–933
overview, 930
standard input, 936–938
standard output, 934–936
summary, 945–946
TOY machine, 914
Magnitude
complex numbers, 402–403
spatial vectors, 442–443
Magritte, René, 363
main() methods, 4–5
multiple, 229
transfer of control, 193–194
Majority function
adder circuits, 1028–1030
sum-of-products circuits, 1027
truth tables for, 1025
Mandelbrot, Benoît, 297, 406
Mandelbrot program, 406–409
Maps, Mercator projections, 48
Markov, Andrey, 176
Markov chains
impact, 184
mixing, 179–184
overview, 176
power method, 180–181
squaring, 179–180
Markov model paradigm, 460
Markov program, 180–182
Markov systems, 802–803
Markovian queues, 597
Marsaglia’s method, 85, 259
Masking bitwise operations, 892–893
Matcher class for REs, 763
Matching graphs, 713
Math library, 192
accessing, 228
methods, 30–32, 193, 198
Mathematical analysis, 498–502
Mathematical functions, 202–204
Mathematical induction, 262, 266
Mathematical models, 716
Matiyasevich, Yuri, 816
Matlab language, 1094
Matrices
boolean, 302
Hadamard, 122
images, 346–347
matrix multiplication, 109
sparse, 666
transition, 172–173
two-dimensional arrays, 106, 109–110
vector multiplication, 110, 180
Mauchly, John, 924–925
Maximum values in arrays, 209
Maximum keys in BSTs, 651
Maxwell–Boltzmann distributions, 257
McCarthy’s 91 function, 298
Mechanical systems, graphs for, 673
 Memoization, 284
Memory
arrays, 91, 94, 515–517
ArrayStackOfStrings, 569–570
available, 520
bit-slice design, 1054–1056
circuits, 1054–1057
 feedback loops as, 1048
flip-flops, 1049–1050
interfaces, 1054
leaks, 367, 581
linked lists, 571
memory bits, 1056
objects, 338, 514
 performance, 513–517
recursion, 282
references, 367
strings, 515
TOY machine, 908–909
two-dimensional arrays, 107
virtual, 972, 975–976
Memory dumps, 909
Memory instructions
address instructions, 912
TOY machine, 913
Memory writes for CPU, 1079
Memoryless queues, 597
Mercator projections, 48
Merge program, 550–552
Mergesort
 divide-and-conquer, 554
overview, 550–552
 performance, 553
Metacharacters, 724, 730–731
Method references, 470
Methods
 abstract, 446
call chaining, 404
deprecated, 469
instance, 334, 385–386
instance vs. static, 340
library, 29–32
main(), 4–5
overriding, 452
static. See Functions; Static methods
stub, 303
variables within, 386–388
MIDI Tuning Standard, 161
Midpoint displacement method, 278, 280
Milgram, Stanley, 670
Minimum keys in BSTs, 651
Minsky, Marvin, 780, 794
Minus signs (–)
 compound assignments, 60
floating-point numbers, 24–26
 integers, 22
lambda expressions, 450
MIX machine, 947
Mixed-type operators, 27–29
Mixing Markov chains, 176, 179–184
MM1Queue program, 598–600
Models
circuit. See Circuit models
 computational, 716
mathematical, 716
universal, 794–797
Modular programming, 191
classes in, 227–229
code reuse, 226, 253
debugging, 253
designing, 29–32
encapsulation, 367
flow of control in, 227–228
tables in, 251–254
machine language, 932
maintenance, 253
program size, 252–253
Modules
 abstraction layers, 1037
as classes, 228
CPU, 1076
description, 1034
tables, 319
overview, 191
program counters, 1073
summary, 254
Monochrome luminance, 343–344
Monte Carlo simulation, 300, 307–308
Moore’s Law
coping with, 971
description, 507–508
Move-to-front strategy, 620
Movie–performer graph, 680
Multidimensional arrays, 111
Multiple arguments, 197
Multiple inheritance, 470
Multiple main() methods, 229
Multiple return statements, 198
Multiple I/O streams, 143
Multiplexers
bus switching, 1035
description, 1023
selection, 1019–1020
Multiplication
complex numbers, 402–403
floating-point numbers, 24–26
integers, 22–23, 885
matrices, 109–110
P search problems, 839
polar representation, 433
Multiway gates, 1015–1017, 1023
Music, 155–159
Mutable types, 364, 439
N
N-body simulation
Body data type, 479–480
file format, 483
force, 480–482
overview, 478–479
summary, 488
Universe data type, 483–487
Names
arrays, 91
methods, 5, 30, 196
objects, 362
variables, 16
vertices, 675
NaN value, 26, 40
NAND function, 989–991
Nash, John, 840
Natural numbers, 875
Natural recursion, 262
Negation axiom, 990
Negative numbers
array indexes, 116
representing, 38, 886–888
Neighbor vertices, 671
Nested classes
iterators, 574
linked lists, 603–605
Nesting conditionals and loops, 62–64
new keyword
constructors, 385
Node objects, 609
String objects, 333
Newcomb, Simon, 224
Newline characters (\n)
compiler considerations, 10
escape sequences, 19
Newton, Isaac
dice question, 88
motion simulation, 478–479
square root method, 65
Newton’s law of gravitation, 481
Newton’s method, 65–67
Newton’s second law of motion, 480–481
NFAs. See Nondeterministic finite-state automata (NFAs)
90–10 rule, 170, 176
Nodes
BSTs, 640–642, 942
linked lists, 571–573
new keyword, 609
Nondeterministic finite-state
automata (NFAs)
DFA equivalence, 749–750
Kleene’s theorem.
See Kleene’s theorem
overview, 744
RE equivalence, 750–751
recognition problem, 744–745
trace example, 747
Nondominant inner loops, 510
NOR function, 989–991
NOR gates
cross-coupled, 1050
description, 1014
Normal distribution functions
cumulative, 202–203
probability density, 202–203
NOT gates, 1013–1014
Not operation, 26–27, 987–989
NP-completeness
addressing problems, 852
boolean satisfiability, 853–856
classifying problems, 851
Cook–Levin theorem, 844–847
coping, 850–857
Karp’s reductions, 845–848
overview, 843–844
proving, 844–849
NP-hard problems, 858
NP search problems
difficult, 837
easy, 837
main question, 840–841
nondeterminism, 835
overview, 833
solutions, 835
subset sum, 834
TSP problem, 862
vertex cover problem, 834, 842
0/1 ILP problem, 835
Null calls, 312
Index

Null keys in symbol tables, 626
Null links in BSTs, 640
Null nodes in linked lists, 571–572
null keyword, 415
Null transitions in NFAs, 744–746
Null values in symbol tables, 626
NullPointerException, 370
Numbers
  conversions, 21, 67–69, 880–881
  intractability, 827
  negative, 886–888
  real, 888–890
Numerical integration, 449
Nyquist frequency, 161

Object class, 453–455
Object-oriented programming
  data types. See Data types
  description, 254
  overview, 329
Objects
  arrays, 365
  collections, 582–583
  comparing, 364, 545–546
  Complex, 404
  equality, 454–456
  memory, 514
  names, 362
  orphaned, 366
  references, 338–339
  String, 333–334
  type conversions, 339
  uninitialized variables, 339
  working with, 338–339
Observations, 495–496
Occam’s Razor, 814
Octal representation, 898
Odd parity function
  adder circuits, 1028–1030
  sum-of-products circuits, 1026
  truth tables for, 1026
Off-by-one errors, 92
Offscreen canvas, 151
On computable numbers, with an application to the Entscheidungsproblem article, 717
On/off switches, 1005
One-dimensional arrays, 90
One-hot OR gates, 1023
Onscreen canvas, 151
Opcodes, 911
Operands, 17
Operators and operations
  boolean, 26–27, 989–991
  comparisons, 27–29, 364
  compound assignments, 60
  data types, 14, 331
  description, 15
  expressions, 17, 587
  floating-point numbers, 24
  integers, 22, 891
  lambda, 450
  overloading, 416
  precedence, 17
  reverse Polish notation, 590
  stacks, 590
  strings, 19, 21, 334, 453
  TOY machine, 906
Optimal data compression, 814
Optimization
  NP problems, 835
  premature, 518
Optimizing compilers, 814
OR function, 987–989
OR gates, 1014, 1023
Or operation
  bitwise, 891–892
  boolean type, 26–27
  TOY machine, 913
Order in BSTs, 640, 642–643
Order statistics, 651
Order-of-growth classifications
  constant, 503
  cubic, 505–508
  exponential, 505–508
  linear, 504–505, 507–508
  linearithmic, 504–505, 507–508
  logarithmic, 503
  overview, 503
  performance analysis, 500–501
  quadratic, 504–505, 507–508
Ordered operations
  binary search trees, 651
  symbol tables, 624
Orphaned objects, 366
Orphaned stack items, 581
Out library, 355–356
Outer loops, 62
Outline shapes, 149
Output
  arithmetic logic units, 1032
  array libraries, 237–238
  circuit models, 1002–1004
  clocks, 1059–1060
  data types, 353
  file concatenation, 356
  gates, 1013
  machine language, 934–936
  print statements, 8
  printf() method, 126–129
  standard, 127, 129–132
  standard audio, 155–159
  standard drawing.
    See Standard drawing
  stream data types, 355
  two-dimensional arrays, 107
  virtual machines, 969–970
Overflow
  arithmetic, 885
  arrays, 95
  attacks, 963
  guarding against, 898
  integers, 23
  negative numbers, 38
Overhead for objects, 514
Overloading
operators, 416
static methods, 198
Overriding methods, 452

P
The \( P = NP \) Question and Gödel’s Lost Letter book, 856
\( P \) search problems, 837
examples, 839
main question, 840–841
Padding object memory, 514
Page, Lawrence, 184
Page ranks, 176–177
Palindromes
description, 719
Watson–Crick, 374
Paper size, 294
Paper tape, 934–938
Papert, Seymour, 400
Parallel arrays, 411
Parallel edges, 676
Parameter variables
lambda expressions, 450
static methods, 196–197, 207
Parameterized data types, 582–586
Parameters
in performance, 511
TOY-8 machine, 1070
Parentheses ()
casts, 33
constructors, 333, 385
expressions, 17, 27
functions, 24, 197
lambda expressions, 450
methods, 30, 196
operator precedence, 17
regular expressions, 724
stacks, 587, 590
static methods, 196
vectors, 442
Parity in ripple–carry adders, 1028
Parsing
instructions, 966–967
strings, 880–882
Pascal’s triangle, 125
Passing arguments
references by value, 364–365
static methods, 207–210
PathFinder program, 683–686, 690–692
Paths
graphs, 674, 683–692
intractability problems, 829
shortest. See Shortest paths
simple, 710
Pattern class for REs, 763
PCs. See Program counters (PCs)
PDA (pushdown automata), 755–756
PDP-8 computers, 906
Peaks in terrain analysis, 167
Pell’s equation, 869
Pens
color, 150
drawings, 146
Pepys, Samuel, 88
Pepys problem, 88
Percent signs (%)
conversion codes, 131–132
remainder operation, 22–23
Percolation case study
adaptive plots, 314–318
lessons, 318–320
overview, 300–301
Percolation, 303–304
PercolationPlot, 315–317
PercolationProbability, 310–311
PercolationVisualizer, 308–309
probability estimates, 310–311
recursive solution, 312–314
scaffolding, 302–304
testing, 305–308
vertical percolation, 305–306
Performance
binary search trees, 647–648
binary searches, 535
caveats, 509–511
comparing, 508–509
guarantees, 512, 627
hypotheses, 496–502
importance, 702
insertion sorts, 544–545
memory use, 513–517
mergesort, 553
multiple parameters, 511
order of growth, 503–506
overview, 494–495
perspective, 518
prediction, 507–509
scientific method, 495–502
shortest paths, 690
wrapper types, 369
Performer program, 697–699
Periods (. )
classes, 227
regular expressions, 724
Permutations
inverse, 122
sampling, 97–99
Phase transitions, 317
Phone books, 628
Photographs, 346
Physical systems, graphs for, 672
Pi constant, 31–32
Picture library, 346–347
Piecewise approximation, 148
Pigeonhole principle, 754–755
Piping
  connecting programs, 141
  filters, 142–143
Pixels in image processing, 346
Plasma clouds, 280
Playing card possibilities, 823
PlayThatTune program, 157–158
PlayThatTuneDeluxe program, 213–215
PlotFilter program, 146–147
Plotting
  array values, 246–248
  experimental results, 249–250
  function graphs, 148, 248
  percolation case study, 314–318
  sound waves, 249
Plus signs (+)
  compound assignments, 60
  floating-point numbers, 24–26
  integers, 22
  regular expressions, 731
  string concatenation, 19–20
Pointers
  array elements, 94
  handles, 371
  object references, 338
  safe, 366
Poisson processes, 597
Polar coordinates, 47
Polar representation, 433–434
Polling, statistical, 167
Polymorphism, 448
Polynomial time, 823
Polynomial-time algorithms
  intractability, 825–826
  \( P \) search problems, 837, 839
  usefulness, 858
Polynomial-time reductions, 841–843
Pop operation
  reverse Polish notation, 590–591
  in stacks, 567–568
Positional notation, 875
Post, Emil, 813–814
Post correspondence problem, 813–814
Postconditions in assertions, 467
Postfix notation, 590
Postorder tree traversal, 649
PostScript language, 400, 590
PotentialGene program, 336–337
Pound signs (#), 769
Power method, 180–181
Power source, 1003–1004
PowersOfTwo program, 56–58
Precedence
  arithmetic operators, 17
  regular expressions, 724
Precision
  floating-point numbers, 25, 40
  \printf(), 130–131
  standard output, 129–130
Precomputed array values, 99–100
Preconditions in assertions, 467
Prediction, performance, 507–509
Preferred attachment process, 713
Prefix-free strings, 564
Premature optimization, 518
Preorder tree traversal, 649
Primality-testing function, 198–199
Prime numbers
  in factoring, 72–73
  Sieve of Eratosthenes, 103–105
PrimeSieve program, 103–105
Primitive data types, 14
  memory size, 513
  overflow checking, 39
  performance, 369
  wrappers, 457
Principle of superposition, 483
print() method, 31
  arrays, 237–238
  impurity, 32
Out, 355
  vs. println(), 8
  standard output, 129–130
Print statements, 5
printf() method, 129–132, 355
Printing, formatted, 130–132
println() method, 31
  description, 5
  impurity, 32
  Out, 355
  vs. print(), 8
  standard output, 129–130
  string concatenation, 20
private keyword
  access modifier, 384
  encapsulation, 433
Probabilities, 308, 310–311
Probability density function, 202–203
Problem reduction
  overview, 811
  program equivalence, 812
  Rice's theorem, 812–813
  totality problem, 811–812
Problem size in intractability, 824
Procedural programming style, 329
Program counters (PCs)
  bus connections, 1073–1074
  connections and timing, 1075
  control lines, 1074–1075
  interfaces, 1073
  modules, 1073
  overview, 1073
  TOY machine, 910
Program equivalence problem, 812
Program size, 252–253
Programming environments, 1094
Programming languages
  indexing, 634
  stack-based, 590
  symbol tables, 629
Programming overview, 1
  HelloWorld example, 4–6
  input and output, 7–8
  process, 2–3
Programs
  connecting, 141
  processing programs, 788–790, 964–966
Proof by contradiction, 754
Pseudo-code, 911
public keyword
  access modifiers, 384
  description, 228
  static methods, 196
Pulses, clock, 1058
Punched cards, 940
Punched paper tape, 934–938
Pure functions, 201
Pure methods, 32
Push operation
  reverse Polish notation, 590–591
  stacks, 567–568
Pushbuttons for TOY machine, 916
Pushdown automata, 755–756
Pushdown stacks, 567–568
Put operations
  hash tables, 639
  symbol tables, 624
Putnam, Hilary, 816
R
  Race conditions in flip-flops, 1050
  Ragged arrays, 111
  Ramanujan, Srinivasa, 86
  Ramanujan's taxi, 86
  Random graphs, 695
  Random numbers
    fair coin flips, 52–53
    function implementation, 199
    Gaussian, 47
    impurity, 32
    libraries, 232–236
    random sequences, 127–128
    Sierpinski triangles, 239–240
    simulations, 72–73
    Math.random(), 30–31
  Random queues, 596
  Random shortcuts, 699
  Random walks
    Brownian bridges, 278
    self-avoiding, 112–115
    two-dimensional, 86
    undirected graphs, 712
  Random web surfer case study
    histograms, 177
    input format, 171
  lessons, 184–185
  Markov chains, 176, 179–184
  overview, 170–171
  page ranks, 176–177
  simulation, 174–178
  transition matrices, 172–173
  RandomInt program, 33–34
  RandomSeq program, 127–128
  RandomSurfer program, 175–177
  RangeFilter program, 140–143
  Ranges
    binary search trees, 651
    functions, 192
  Ranks
    binary search trees, 651
  Raphson, Joseph, 65
  Raster images, 346
  Real numbers, 888–890
  Receivers in cryptography, 992
  Recognition problem
    formal languages, 722
    NFAs, 744–745
    REs, 728–729, 735, 753
  Recomputation, 282–283
  Rectangle rule, 449
  Recursion, 191
    base cases, 281
    BSTs, 640–641, 644, 649
    binary searches, 533
    Brownian bridges, 278–280
    considering, 320
    convergence issues, 281–282
    dynamic programming, 284–289
    Euclid's algorithm, 267–268
    exponential time, 272–273
    factorial example, 264–265
    function-call trees, 269, 271
    graphics, 276–277, 397
  Quadratic Koch island fractal, 803
  Quadratic order of growth, 504–505, 507–508
  Quadratic program, 25–26
  Quadrature integration, 449
  Quaternions, 424
  Question marks (?) in REs, 731
  Questions program, 533–535
  Queue program, 592–596, 604–605
  Queues
    circular, 620
    deques, 618
    FIFO. See First-in first-out (FIFO) queues
    overview, 566
    random, 596
    summary, 608
  Queuing theory, 597–600
  Quotes (" in text, 5
  Quadratic program, 25–26
  Quadrature integration, 449
  Quaternions, 424
  Question marks (?) in REs, 731
  Questions program, 533–535
  Queue program, 592–596, 604–605
  Queues
    circular, 620
    deques, 618
    FIFO. See First-in first-out (FIFO) queues
    overview, 566
    random, 596
    summary, 608
  Queuing theory, 597–600
  Quotes (" in text, 5
  Quadratic program, 25–26
  Quadrature integration, 449
  Quaternions, 424
Gray codes, 273–275
linked lists, 571
mathematical induction, 266
memory requirements, 282
mergesort, 550
overview, 262–263
percolation case study, 312–314
perspective, 289
pitfalls, 281–283
recomputation issues, 282–283
towers of Hanoi, 268–272
Red–black trees, 648
Redirection, 139
  piping, 142–143
  standard input, 140–141
  standard output, 139–140
Reduced instruction set computing (RISC), 974
Reductio ab absurdum, 808
Reduction
  binary search trees, 640
  mergesort, 554
  polynomial-time, 841–843
  problem, 811–813
  recursion, 264–265
References
  accessing, 339
  aliasing, 363
  arrays, 365
  equality, 454–455
  garbage collection, 367
  immutable types, 364, 441
  linked lists, 572
  memory, 367
  method, 470
  object-oriented programming, 330
  objects, 338–339
  orphaned objects, 366
  passing, 207, 210, 364–365
  performance, 369
  properties, 362–363
  safe pointers, 366
  Reflexive property, 454
  Registers
    implementing, 1052
    machine language, 931
    overview, 1051–1052
    TOY machine, 909, 911
    writing to, 1052–1053
  Regular expressions (REs)
    applications, 732–736
    computational biology, 732–734
    generalized, 730–732
    NFA equivalence, 750–752
    overview, 724–725
    recognition problem, 728–729, 735, 753
    regular languages, 725–727
    searches, 734–736
    shorthand notations, 730–731
    validity checking, 732
  Regular languages, 723
  basic operations, 723–724
  regular expressions. See Regular expressions (REs)
  Reject states
    DFAs, 738–739
    Turing machines, 766–767
  Relative entropy, 667–668
  Relays in circuit models, 1006
  Remainder operation, 22–23
  Removing
    array items, 569
    collection items, 566, 602–603
    linked list items, 573–574
    NFA nodes, 751
    queue items, 592, 596
    set keys, 652
    stack items, 567–569
    symbol table keys, 624–627
  Rendell, Paul, 805
  Repetitive code, simplifying, 100
  Representation in APIs, 431
  Representing information
    binary and hex, 875–880
    bit manipulation, 891–893
    characters, 894–895
    integer arithmetic, 884–885
    negative numbers, 886–888
    overview, 874
    real numbers, 888–890
    strings, 880–883
    summary, 896
  Reproducible experiments, 495
  Reserved words, 16
  Resetting flip-flops, 1050
  Resizing arrays, 578–581, 635
  ResizingArrayStackOfStrings program, 578–581
  Resource allocation
    graphs for, 673
    overview, 606–607
  Resource-sharing systems, 606–607
  Return addresses, 931
  return statements, 194, 196, 198
  Return values
    arrays as, 210
    methods, 30, 196, 200, 207–210
    reverse Polish notation, 591
  Reuse, code, 226, 253, 701
  Reverse Polish notation, 590
  RGB color format, 48–49, 341, 371
  Rice, Henry, 812
  Rice’s theorem, 812
  Riemann integral, 449
  Riffle shuffles, 125
  Right shift operations
    bitwise, 891–892
    TOY machine, 913
  Right subtrees, 640
  Right triangles, 199
  Ring buffers, 620
Index

Ring graphs, 694–695, 699
Ripple–carry adders, 1028–1030
RISC (reduced instruction set computing), 974
Robinson, Julia, 816
Roots in binary search trees, 640
Rotation filters, 379
Roulette-wheel selection, 174
Round-robin policies, 606
Rows in 2D arrays, 106, 108
RR-format instructions, 911
Rule program, 19–20
Run-time errors, 6
Running time. See Performance
Running virtual machines, 969
RuntimeException, 466

S
Safe pointers, 366
Sample program, 98–99
Sample standard deviation, 246
Sample variance, 244
Sampling
  audio, 156–157
  function graphs, 148
  scaling, 349–350
  without replacement, 97–99
SAT problem, 832
  nondeterministic TM, 836
NP-completeness, 844–846, 853–856
SAT program, 855–856
Satisfiability, 830
  boolean, 832, 836
  integer linear inequality, 831
  linear equation, 830
  linear inequality, 831
NP-completeness, 844–846, 853–856
Saving audio files, 157
Scaffolding, 302–304
Scale program, 349–350
Scaling
  drawings, 146
  image processing, 349–350
  spatial vectors, 442–443
Scientific computing, 1094
Scientific method, 494–495
  hypotheses, 496–502
  observations, 495–496
Scientific notation
  conversion codes, 131–132
  real numbers, 888–889
Scope of variables, 60, 200
Screen scraping, 357–359
Search problems
  difficult, 837–838
  easy, 837
  nondeterminism, 836
  overview, 833
  solutions, 835
  subset sum, 834
  TSP problem, 862
  vertex cover problem, 834, 842
0/1 ILP problem, 835, 842
Searches
  binary. See Binary searches
  binary search trees. See Binary search trees (BSTs)
  bisection, 537
  breadth-first, 683, 687–688, 690, 692
  data mining example, 458–464
  depth-first, 312, 690
  indexing, 634
  overview, 532
  regular expressions, 734–736
  for similar documents, 464
Secret messages, 992
Seeds for random numbers, 475
Select control lines, 1056
Self-avoiding walks, 112–115, 710
Self-loops for edges, 676
Self-modifying code, 922–924
SelfAvoidingWalk program, 112–115
Semantics, 52
Semicolons (;)
  for loops, 59
  statements, 5
Sequential circuits
  clocks, 1058–1061
  description, 1008
  feedback circuits, 1048–1049
  flip-flops, 1049–1050
  memory, 1054–1057
  overview, 1048
  registers, 1051–1053
  summary, 1062–1063
Sequential searches, 535–536
Server farms, 976
Servers, 606
Service rate, 597–598
SET library, 652–653
Sets
  elementary functions, 1001
  gates, 1045
  graphs, 676
  Julia, 427
  Mandelbrot, 406–409
  overview, 652–653
  of values, 14
Setting flip-flops, 1050
Shadow variables, 419
Shannon, Claude, 1013, 1041
Shannon entropy, 378
Shapes, outline and filled, 149
Shifts
  bits, 891–892
  circular, 375
  linear feedback shift registers (LFSRs), 1000–1001
  purpose, 898–899
  TOY machine, 913
Index

Stacks
- arithmetic expression evaluation, 586–589
- arrays, 568–570, 578–581
- function calls, 590–591
- linked lists, 574–576
- overview, 566
- parameterized types, 582–586
- pushdown, 567–568
- stack-based languages, 590
- summary, 608

Standard audio
- concert A, 155
- description, 126, 128–129
- music example, 157–158
- notes, 156
- overview, 155
- sampling, 156–157
- saving files, 157
- summary, 159

Standard deviation, 246

Standard drawing
- control commands, 145–146
- description, 126, 128–129
- double buffering, 151
- filtering data to, 146–147
- function graphs, 148
- outline and filled shapes, 149
- overview, 144–145
- summary, 159
- text and color, 150

Standard input
- arbitrary size, 137–138
- description, 126, 128–129
- formatted, 135
- interactive, 135–136
- machine language, 936–938
- multiple streams, 143
- overview, 132–133
- redirecting, 140–141
- summary, 159

Stacks
- typing, 134
- virtual machines, 969–970

Standard output
- description, 127
- formatted, 130–132
- machine language, 934–936
- multiple streams, 143
- overview, 129–130
- piping, 141–143
- redirecting, 139–140
- summary, 159
- virtual machines, 969–970

Standard statistics, 244–250

Standards
- API, 429

Start codons, 336

Statements
- assignment, 17
- blocks, 50
- declaration, 15–16
- methods, 5

States
- DFAs, 738–739
- NFAs, 744–746
- objects, 340
- Turing machines, 766–772
- virtual machines, 968

Static methods, 191–192
- accessing, 227–229
- arguments, 197
- for code organization, 205–206
- control flow, 193–195
- defining, 193, 196
- function-call traces, 195
- function calls, 197
- implementation examples, 199
- vs. instance, 340
- libraries. See Libraries
- overloading, 198
- passing arguments, 207–210
- returning values, 207–210

Side effects, 201
- summary, 215
- superposition example, 211–215
- terminology, 195–196
- variable scope, 200

Static variables, 284

Statistical polling, 167

Statistics, 244–250

StdArrayIO library, 237–238

StdAudio library, 128–129, 155

StdDraw library, 128–129, 144–145, 150, 154

StdIn library, 128–129, 132–133

StdOut library, 129–131

StdRandom program, 232–236

StdStats program, 244–247

StockAccount program, 410–413

StockQuote program, 358–359

Stop codons, 336

Stopwatch program, 390–391

Store instruction, 938, 1080

Stored-program computers, 922–924

Streams
- input, 354–355
- output, 355
- screen scraping, 357–359

Stress testing, 236

Strings and String data type
- alphabet symbols, 718
- API, 332–333
- binary, 718–719
- circular shifts, 375
- concatenation, 19–20, 723–724
- conversion codes, 131–132
- conversions, 21, 453
- description, 14–15
- genomics application, 336–340
- immutable types, 439–440
- input, 133
- internal storage, 37
invoking instance methods, 334
memory, 515
objects, 333–334
overview, 331
parsing, 880–882
prefix-free, 564
representation, 882–883
as sequence of characters, 19
shortcuts, 334–335
string replacement systems, 795
unions, 723
variables, 333
vertices, 675
working with, 19–21
Strogatz, Stephen, 670, 693, 713
Structured programming, 926
Stub methods, 303
Subclassing inheritance, 452–457
Subgraphs, induced, 705
Subset sum problem
intractability, 827–828
\textbf{NP}, 834, 838
Subtraction
floating-point numbers, 24–26
integers, 22
negative numbers, 887
Subtrees, 640, 651
Subtyping inheritance, 446–451
Sum-of-powers conjecture, 89
Sum-of-products
adders, 1028
boolean representation, 996–997
circuits, 1024–1027
Sums, finite, 64–65
Superclasses, 452
Superposition
force vectors, 483
sound waves, 211–215
Swirl filters, 379
Switch control lines, 1005
Switch statements, 74–75
Switches
bus muxes, 1036
circuit models, 1002, 1005–1006
demultiplexers, 1022
gates, 1013
multiplexers, 1020
TOY machine, 916–917
Switching circuit analysis, 1007
Switching time of gates, 1013
Symbol tables
APIs, 625–627
BSTs. See Binary search trees
dictionary lookup, 628–632
graphs, 676
hash tables, 636–639
implementations, 635–636
indexing, 632–634
machine language, 944
overview, 624–625
perspective, 654
sets, 652–653
Symbolic names in assembly, 981
Symbols
definition, 757
description, 718–719
DFA, 738
NFA, 744
regular expressions, 724
Turing machines, 766–767
Symmetric order in BSTs, 640
Symmetric property, 454
Syntax errors, 10–11
\textbf{T}
Tables
of functions, 907–908
hash, 636–639
symbol. See Symbol tables
Tabs
compiler considerations, 10
escape sequences, 19
Tape and tape readers
DFAs, 738–739
Turing machines, 766–769, 774–776
Tape program, 776
Taylor series approximations, 204
Templates, 50
TenHe11os program, 54–55, 60
Terminal windows, 127
Terms, glossary for, 1097–1101
Terrain analysis, 167
Testing
for bugs, 318
importance, 701
percolation case study, 305–308
Text. See also Strings and String
data type
drawings, 150
printing, 5, 10
Text editors, 3
Theory of computing, 715–717
\textbf{this} keyword, 445
Thompson, Ken, 735
3n+1 problem, 296–297
ThreeSum program, 497–502
Throwing exceptions, 465–466
Thue word problem, 819
Ticks, clock, 1058
Tilde notation, 500
Tildes (\texttt{~})
bitwise operations, 891
boolean type, 991
frequency analysis, 500
Time
exponential, 272–273, 823
performance. See Performance
polynomial, 823
Stopwatch timers, 390–391
TimePrimitives program, 519
Timesharing, 965
Tools, building, 320
Index

self-avoiding walks, 112–115
setting values, 108
spreadsheets, 108
Two's complement, 38, 886–888
Type arguments, 585, 611
Type conversions, 34–35
Type parameters, 585
Type safety, 18
Types. See Data types

U
Unboxing, 457, 585–586
Undirected graphs, 675
Unicode characters
description, 19
overview, 894–895
strings, 37
Uniform random numbers, 199
Uninitialized variables, 94, 339
Union operation in REs, 723
Unit testing, 235
Universal models, 794–797
Universal sets
elementary functions, 1001
gates, 1045
Universal Turing machines
(UTMs), 789–790
Universal virtual DFAs, 741–743
Universal virtual TMs, 774–779
Universality
algorithms, 786–787
Church–Turing thesis, 790–791
overview, 786
programs processing programs,
788–790
Turing machine variations,
791–794
universal models, 794–797
virtual DFA/NFA, 788–789
Universe program, 483–487
Unreachable code error, 216
Unsigned integers, 884
Unsolvability proof, 810
Unsolvable problems, 430
blank tape halting problem, 820
definite integration, 816
description, 806
examples, 815
halting problem, 808–810
Hilbert's 10th problem, 816
implications, 816–817
liar's paradox, 807–808
optimal data compression, 814
optimizing compilers, 814
Post correspondence, 813–814
program equivalence, 812
totality, 811–812
Upper bounds, 825
Upscaling in image processing, 349
UseArgument program, 7–8
User-defined libraries, 230
UTF-8 encoding, 895
UTMs, 789–790

V
Validate program, 729
Validity checking, 732
Values
array, 95–96
data types, 14, 331
passing arguments by, 207, 210,
364–365
precomputed, 99–100
symbol tables, 624–626
Variables
assignment statements, 17
boolean, 987, 994–997
compound assignments, 60
constants, 16
description, 15–16
initial values, 415
inline initialization, 18
instance, 384
within methods, 196, 386–388
names, 16
scope, 60, 200
shadow, 419
static, 284
string, 333
tracing values, 18
uninitialized, 339
Vector images, 346
Vector program, 443–445, 515
Vectors
arrays, 92
cross products, 472
dot products, 92, 442–443
matrix–vector multiplication,
110
n-body simulation, 479–480
sparse, 666
spatial, 442–445
vector–matrix multiplication,
110, 180
Vertex cover problem
intractability, 828
NP-completeness, 846–847
NP search problems, 834, 842
Vertical bars (|)
bitwise operations, 891–892
boolean type, 26–27, 991
piping, 141
regular expressions, 724
Vertical OR gates, 1023
Vertical percolation, 305–306
Vertices
bipartite graphs, 682
creating, 676
eccentricity, 711
graphs, 671, 674
isolated, 703
names, 675
PathFinder, 683
String, 675
Virtual machines
booting, 959–960, 968–969
cautions, 961–963
and cloud computing, 924
description, 965
dumping, 960–961
instructions, 966–967
Moore’s law, 971
overview, 958–959
program development, 970–971
programs that process
programs, 964–966
running, 969
standard input, 969–970
standard output, 969–970
states, 968
TOY machine family, 972–977
universal virtual DFAs, 742
universal virtual TM, 774–779
Viruses, 963
Viterbi algorithm, 286
void keyword, 201, 216
Volatility
Black–Scholes formula, 565
Brownian bridges, 278, 280
W
Walks
random. See Random walks
self-avoiding, 112–115, 710
Watson–Crick palindrome, 374
Watts, Duncan, 670, 693, 713
Watts–Strogatz graph model, 713
.Wav format, 157
Wave filters, 379
Web graphs, 695
Web pages, 170
indexes searches, 634
preferential attachment, 713
Weighing objects, 540–541
Weighted averages, 120
Weighted superposition, 212
while loops, 53–59
examples, 61
nesting, 62
Whitelists, binary searches for, 540
Whitespace characters
compiler considerations, 10
input, 135
Wide interfaces
APIs, 430
examples, 610–611
Wildcard operation in REs, 724
Wiles, Andrew, 722
Wind chill, 47
Wires
circuit models, 1002–1004
gates, 1013
Word ladders, 710
Words
binary representation, 875
computer, 874
memory size, 513
size, 897
Worst-case performance
big-O notation, 520–521
binary search trees, 648
description, 512
insertion sort, 544
intractability, 825
NP-completeness, 852
Wrapper types
autoboxing, 585–586
references, 369, 457
Write control lines
CPU, 1079–1080
memory bits, 1056
register bits, 1051
X
XOR circuits
in arithmetic logic units, 1031
sum-of-products, 1024–1025
xor (exclusive or) operation, 891–892, 913
Y
Y2K problem, 435, 976
Young tableaux, 530
Z
Zero-based indexing, 92
Zero crossings, 164
Zero extension convention, 899
0/1 ILP problem, 831, 835
NP-completeness, 845–846
vertex cover problem, 842
Zeros, leading, 883
ZIP codes, 435
Zipf’s law, 556