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. 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 fundamental 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!).

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 make the effort to convince yourself that recursive programs have the intended effect.
2.3 Recursion

The idea of running a program over and over again is a key concept in computer science. It is used to solve problems by breaking them down into smaller, more manageable parts. Recursion is a technique that can be applied in many different contexts, from mathematics to computer science.

In mathematics, recursion is a method of defining sequences and functions in terms of themselves. For example, the Fibonacci sequence is defined recursively: each number is the sum of the two preceding ones.

In computer science, recursion is used to write programs that perform tasks by repeatedly calling themselves. This technique is especially useful for tasks that involve processing structured data, such as trees or graphs.

Recursion can be implemented using either a recursive function or a recursive process. A recursive function is a function that calls itself repeatedly, while a recursive process is a program that runs in a loop.

Recursion has its advantages and disadvantages. On the one hand, it can make programs easier to understand and write, especially for tasks that involve processing recursive data structures. On the other hand, recursion can also lead to inefficient programs if not used carefully, as it can lead to infinite loops or excessive memory usage.

In this section, we will explore the concept of recursion in depth, looking at its definition, history, and applications in both mathematics and computer science.
Your first recursive program  The HelloWorld for recursion (the first recursive program that most programmers implement) 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 static method calls itself. The implementation clearly produces the desired effect. You can persuade yourself that it does so by noting that \( \text{factorial}(0) \) 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 method 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 \( \text{factorial}(1) \), 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 \( \text{factorial}(n) \) implementation exhibits the two main components that are required for every recursive function. 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 re-

\[
\begin{align*}
\text{factorial}(5) & = 5 \times \text{factorial}(4) \\
\text{factorial}(4) & = 4 \times \text{factorial}(3) \\
\text{factorial}(3) & = 3 \times \text{factorial}(2) \\
\text{factorial}(2) & = 2 \times \text{factorial}(1) \\
\text{factorial}(1) & = 1 \\
\end{align*}
\]
2.3 Recursion

cursion. For \texttt{factorial()}, the base case is \(N = 1\). The \textit{reduction step} is the central part of a recursive function. It relates the function at one (or more) inputs to the function evaluated at one (or more) other inputs. For \texttt{factorial()}, the reduction step is \(N \times \texttt{factorial}(N-1)\). All recursive functions must have these two components. Furthermore, the sequence of parameter values must converge to the base case. For \texttt{factorial()}, the value of \(N\) decreases by one for each call, so the sequence of argument values converges to the base case \(N = 1\).

Tiny programs such as \texttt{factorial()} are perhaps slightly more clear if we put the reduction step in an \texttt{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 braces after the \texttt{else}. Instead, we adopt the convention of always putting the base case as the first statement, ending with a \texttt{return}, and then devoting the rest of the code to the reduction step.

The \texttt{factorial()} implementation itself is not particularly useful in practice because \(N!\) grows so quickly that the multiplication will overflow and produce incorrect answers for \(N > 20\). But the same technique is effective for computing all sorts of functions. For example, the recursive function

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

is an effective method for computing the 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 \textit{any} discrete sum for which you have a compact formula. Recursive programs like these are just loops in disguise, but recursion can help us better understand this sort of computation.
Mathematical induction  Recursive programming is directly related to mathematical induction, a technique for proving facts about discrete functions.

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 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 all \( N \): 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 \( N(N+1)/2 \). That is, we wish to prove that the following equation is valid for all \( N \geq 1 \):

\[
1 + 2 + 3 \ldots + (N-1) + N = N(N+1)/2
\]

The equation is certainly true for \( N = 1 \) (base case). If we assume it to be true for all integers less than \( N \), then, in particular, it is true for \( N-1 \), so

\[
1 + 2 + 3 \ldots + (N-1) = (N-1)N/2
\]

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 do 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 \texttt{factorial}() computes the product of the positive integers less than or equal to \( N \).
Euclid's algorithm 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 $p \geq q$, the gcd of $p$ and $q$ is the same as the gcd of $p$ and $p \% q$.

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 gcd() in 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 input strictly decreases in each recursive call since $p \% q < q$. If $p < q$, the first recursive call switches the arguments. In fact, the second input decreases by at least a factor of two for every second recursive call, so the sequence of inputs 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 Euclid's algorithm and is one of the oldest known algorithms—it is over 2,000 years old.

Towers of Hanoi No discussion of recursion would be complete without the ancient 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).
2.3 Recursion

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.

TowersOfHanoi (Program 2.3.2) is a direct implementation of this strategy. It reads in a command-line argument \( n \) and prints out the solution to the Towers of Hanoi problem on \( n \) discs. The recursive static method moves() prints the sequence of moves to move the stack of discs to the left (if the argument left is true) or to the right (if left is 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 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 method 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 moves() in 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 of discs in the pile to be moved (and the label of the disc to actu-
Program 2.3.2  Towers of Hanoi

```java
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).

% java TowersOfHanoi 1
1 left

% java TowersOfHanoi 2
1 right
2 left
1 right

% java TowersOfHanoi 3
1 left
2 right
1 left
3 left
1 left
2 right
1 left

% java TowersOfHanoi 4
1 right
2 left
1 right
3 right
1 right
2 left
1 right
4 left
1 right
2 left
1 right
3 right
1 right
2 left
1 right
ally be moved); the second is the direction to move the pile. 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 method 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 method.

Trees are relevant and important in understanding recursion because the tree is the 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, 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.

For the towers of Hanoi, the mathematical model is simple: if we define the function $T(n)$ to be the number of move directives issued by TowersOfHanoi to move $n$ discs from one peg to another, then the recursive code immediately implies that $T(n)$ must satisfy the following equation:

$$T(n) = 2T(n-1) + 1$$

for $n > 1$, 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 $T(n-1) = 2^{n-1} - 1$, then $T(n) = 2 (2^{n-1} - 1) + 1 = 2^n - 1$

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 2.3.9).

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 31 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 1.4 million 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 $2^{1000}$ 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 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 entry in the list differs in precisely one bit from its predecessor. Gray codes directly apply to Beckett's prob-
lem because changing the value of a bit from 0 to 1 corresponds to an character entering the subset onstage; changing a bit from 1 to 0 corresponds to an 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 null, 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 out 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 invoke the ruler function that we considered in one of our first programs (Program 1.2.1). Without the redundant instructions, they both implement a simple recursive function that could allow Ruler to print out 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.

$$\begin{array}{c|c|c}
1\text{-bit code} & 3\text{-bit code} \\
\hline
2\text{-bit} & 4\text{-bit} \\
0 & 0 & 0 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
2\text{-bit code (reversed)} & 3\text{-bit code (reversed)} \\
0 & 0 & 0 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
& & \\
3\text{-bit} \\
0 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 1 \\
0 & 1 & 0 \\
1 & 1 & 1 \\
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & 0 & 0 \\
2\text{-bit code (reversed)} & 3\text{-bit code (reversed)} \\
0 & 0 & 0 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
& & \\
\end{array}$$

2-, 3- and 4-bit Gray codes
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 character alternately enters and exits.

<table>
<thead>
<tr>
<th>% java Beckett 1</th>
<th>% java Beckett 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>enter 2</td>
<td>enter 2</td>
</tr>
<tr>
<td>exit 1</td>
<td>exit 1</td>
</tr>
<tr>
<td>enter 3</td>
<td>enter 1</td>
</tr>
<tr>
<td>exit 1</td>
<td>exit 2</td>
</tr>
<tr>
<td>exit 1</td>
<td>exit 1</td>
</tr>
<tr>
<td>enter 4</td>
<td></td>
</tr>
</tbody>
</table>

In digital-to-binary conversions, the binary-reflected Gray code, were even pro-
Recursive graphics Simple recursive drawing schemes can lead to pictures that are remarkably intricate. Recursive drawings not only relate to numerous applications, but they also provide an appealing platform for developing a better understanding of properties of recursive programs, 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 connected to each tip of the H

with the additional provisos that the H-trees of order n−1 are halved in size and centered in the four quadrants of the square.

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, more than a billion with n = 15, and 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 what 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

lead to pictures that are useful in numerous applications, and help in developing a better understanding of how to watch the process reduce in size.

\[ H \]

\[ (n-1, sz/2, x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x0, y1) \]

\[ (x1, y0) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

The base case is when \( n = 0 \), within the unit

\[ H \]

\[ (n-1, sz/2, x0, y0) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]

\[ (x1, y1) \]

\[ (x, y) \]

\[ (x0, y0) \]
**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 them 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 also 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. We consider one simple example here; you can find much more information about the model on the booksite.

**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 two points, from \((x_0, y_0)\) to \((x_1, y_1)\), 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 interval \([x_0, x_1]\). The base case (when the size 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 \(\delta\), chosen 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 graph strays from the straight line con-
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 < .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 H = Double.parseDouble(args[0]);
        double s = Math.pow(2, 2*H);
        curve(0, .5, 1.0, .5, .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 $H$, known as the Hurst exponent, controls the smoothness of the curves.

![Images of generated curves with different Hurst exponents](Brownian Bridge.png)
necting 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 $1/2$ (divide by 2 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 decrease, resulting in a smoother curve; and when $2 > H > 1/2$, the displacements tend to increase, resulting in a rougher 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 smoothness parameter. 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 produces 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 quadrant 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 computer games.

A plasma cloud
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 H(int N)
{
    return H(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 lines of 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. 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 the original problem. For example, the following method will go into an infinite recursive loop for any value of its argument N except 1:

```java
public static double H(int N)
{
    if (N == 1) return 1.0;
    return H(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 program for computing the Harmonic numbers for increasing values of N:

```java
public static double H(int N)
{
    if (N == 1) return 1.0;
    return H(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 space.

Excessive recomputation. The temptation to write a simple recursive program to solve a problem must always be tempered by the understanding that a simple program might take exponential time (unnecessarily) due to excessive recomputation. This effect is possible even in the simplest recursive programs, 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 ...
```

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
public static long F(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
}
```
However, this program 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 program to compute $F_{50}$. To see why it is futile to do so, consider what the function does to compute $F(8) = 21$. It first computes $F(7) = 13$ and $F(6) = 8$. To compute $F(7)$, it recursively computes $F(6) = 8$ again and $F(5) = 5$. Things rapidly get worse because both times it computes $F(6)$, it ignores the fact that it already computed $F(5)$, and so forth. In fact, you can prove by induction that the number of times this program computes $F(1)$ when computing $F(n)$ is precisely $F_n$ (see Exercise 2.3.12). The mistake of recomputation is compounded exponentially. As an example, to compute $F(200)$, the number of times this method needs to compute $F(1)$ is $F_{200} > 10^{43}$! 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 programs fall into this category. Do not fall into the trap of implementing and trying to run them.

The following is one caveat: a systematic technique known as memoization allows us to avoid this pitfall while still taking advantage of the compact recursive description of a computation. In memoization, we maintain an array that keeps track of the values we have computed so that we can return those values and make recursive calls only for new values. This technique is a form of dynamic programming, a well-studied technique for organizing computations that you will learn if you take courses in algorithms or operations research.
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.

If you are intrigued by the mystery of how the Java system manages to implement the illusion of independently operating copies of the same piece of code, be assured that we will consider this issue in Chapter 4. You may be surprised at the simplicity of the solution. It is so easy to implement that programmers were using recursion well before the advent of high-level programming languages like Java. Indeed, you might be surprised to learn that you could write programs equivalent to the ones considered in this chapter with just the basic loops, conditionals, and arrays programming model discussed in Chapter 1.

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?

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 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 applications 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 a more expansive style of programming that now dominates the computing landscape.