invokes `m2`, which invokes `m3`, which invokes `m4`, which invokes `m1`. Figure 8.4 depicts a situation that involves indirect recursion. Method invocations are shown with solid lines, and returns are shown with dotted lines. The entire invocation path is followed, and then the recursion unravels following the return path.

Indirect recursion requires paying just as much attention to base cases as direct recursion does. Furthermore, indirect recursion can be more difficult to trace because of the intervening method calls. Therefore, extra care is warranted when designing or evaluating indirectly recursive methods. Ensure that the indirection is truly necessary and that it is clearly explained in documentation.

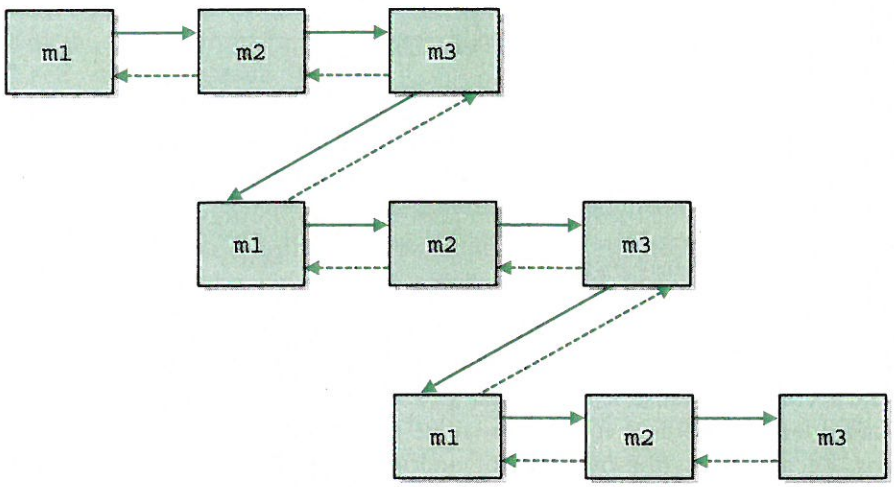

FIGURE 8.4 Indirect recursion

8.3 Using Recursion

The following sections describe problems that we solve using a recursive technique. For each one, we examine exactly how recursion plays a role in the solution and how a base case is used to terminate the recursion. As you explore these examples, consider how complicated a nonrecursive solution for each problem would be.

Traversing a Maze

As we discussed in Chapter 4, solving a maze involves a great deal of trial and error: following a path, backtracking when you cannot go farther, and trying other, untried options. Such activities often are handled nicely using recursion. In Chapter 4,

we solved this problem iteratively, using a stack to keep track of our potential moves. However, we can also solve this problem recursively by using the run-time stack to keep track of our progress. The `MazeTester` program shown in Listing 8.1 creates a `Maze` object and attempts to traverse it.

LISTING 8.1

```
import java.util.*;
import java.io.*;

/**
 * MazeTester uses recursion to determine if a maze can be traversed.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class MazeTester
{
    /**
     * Creates a new maze, prints its original form, attempts to
     * solve it, and prints out its final form.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the name of the file containing the maze: ");
        String filename = scan.nextLine();

        Maze labyrinth = new Maze(filename);

        System.out.println(labyrinth);

        MazeSolver solver = new MazeSolver(labyrinth);
        if (solver.traverse(0, 0))
            System.out.println("The maze was successfully traversed!");
        else
            System.out.println("There is no possible path.");
        System.out.println(labyrinth);
    }
}
```

The Maze class, shown in Listing 8.2, uses a two-dimensional array of integers to represent the maze. The maze is loaded from a file. The goal is to move from the top-left corner (the entry point) to the bottom-right corner (the exit point). Initially, a 1 indicates a clear path, and a 0 indicates a blocked path. As the maze is solved, these array elements are changed to other values to indicate attempted paths and, ultimately, a successful path through the maze if one exists. Figure 8.5 shows the UML illustration of this solution.

LISTING 8.2

```
import java.util.*;
import java.io.*;

/**
 * Maze represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Maze
{
    private static final int TRIED = 2;
    private static final int PATH = 3;
    private int numberRows, numberColumns;
    private int[][] grid;

    /**
     * Constructor for the Maze class. Loads a maze from the given file.
     * Throws a FileNotFoundException if the given file is not found.
     *
     * @param filename the name of the file to load
     * @throws FileNotFoundException if the given file is not found
     */
    public Maze(String filename) throws FileNotFoundException
    {
        Scanner scan = new Scanner(new File(filename));
        numberRows = scan.nextInt();
        numberColumns = scan.nextInt();

        grid = new int[numberRows][numberColumns];
        for (int i = 0; i < numberRows; i++)
            for (int j = 0; j < numberColumns; j++)
```

LISTING 8.2 *continued*

```
                grid[i][j] = scan.nextInt();
        }

        /**
         * Marks the specified position in the maze as TRIED
         *
         * @param row the index of the row to try
         * @param col the index of the column to try
         */
        public void tryPosition(int row, int col)
        {
            grid[row][col] = TRIED;
        }

        /**
         * Return the number of rows in this maze
         *
         * @return the number of rows in this maze
         */
        public int getRows()
        {
            return grid.length;
        }

        /**
         * Return the number of columns in this maze
         *
         * @return the number of columns in this maze
         */
        public int getColumns()
        {
            return grid[0].length;
        }

        /**
         * Marks a given position in the maze as part of the PATH
         *
         * @param row the index of the row to mark as part of the PATH
         * @param col the index of the column to mark as part of the PATH
         */
        public void markPath(int row, int col)
        {
```

LISTING 8.2 *continued*

```
        grid[row][col] = PATH;
    }

    /**
     * Determines if a specific location is valid. A valid location
     * is one that is on the grid, is not blocked, and has not been TRIED.
     *
     * @param row the row to be checked
     * @param column the column to be checked
     * @return true if the location is valid
     */
    public boolean validPosition(int row, int column)
    {
        boolean result = false;

        // check if cell is in the bounds of the matrix
        if (row >= 0 && row < grid.length &&
            column >= 0 && column < grid[row].length)

            //  check if cell is not blocked and not previously tried
            if (grid[row][column] == 1)
                result = true;
        return result;
    }

    /**
     * Returns the maze as a string.
     *
     * @return a string representation of the maze
     */
    public String toString()
    {
        String result = "\n";
        for (int row=0; row < grid.length; row++)
        {
            for (int column=0; column < grid[row].length; column++)
                result += grid[row][column] + "";
            result += "\n";
        }
        return result;
    }
}
```

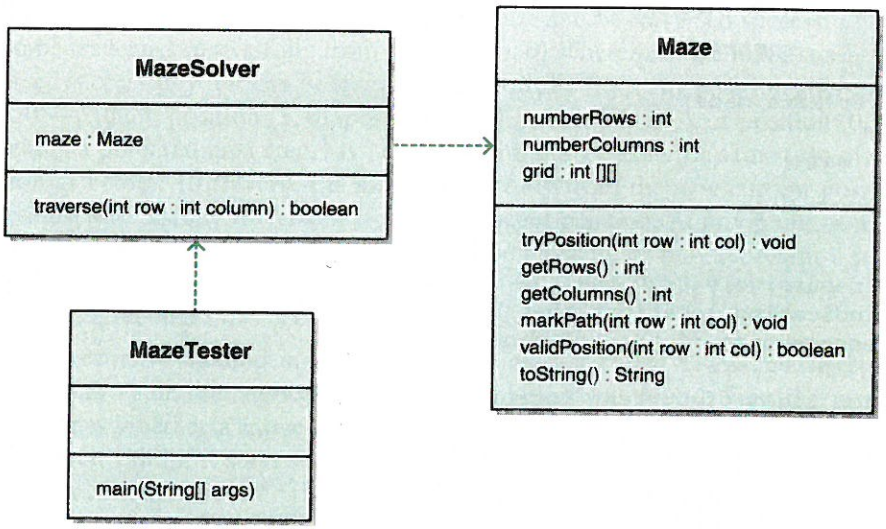

FIGURE 8.5 UML description of the maze-solving program

The only valid moves through the maze are in the four primary directions: down, right, up, and left. No diagonal moves are allowed. Listing 8.3 shows the `MazeSolver` class.

LISTING 8.3

```
/**
 * MazeSolver attempts to recursively traverse a Maze. The goal is to get from the
 * given starting position to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class MazeSolver
{
    private Maze maze;

    /**
     * Constructor for the MazeSolver class.
     */
```

LISTING 8.3 *continued*

```
    public MazeSolver(Maze maze)
    {
        this.maze = maze;
    }

    /**
     * Attempts to recursively traverse the maze. Inserts special
     * characters indicating locations that have been TRIED and that
     * eventually become part of the solution PATH.
     *
     * @param row row index of current location
     * @param column column index of current location
     * @return true if the maze has been solved
     */
    public boolean traverse(int row, int column)
    {
        boolean done = false;

        if (maze.validPosition(row, column))
        {
            maze.tryPosition(row, column);   // mark this cell as tried
            if (row == maze.getRows()-1 && column == maze.getColumns()-1)
                done = true;  // the maze is solved
            else
            {
                done = traverse(row+1, column);       // down
                if (!done)
                    done = traverse(row, column+1);   // right
                if (!done)
                    done = traverse(row-1, column);   // up
                if (!done)
                    done = traverse(row, column-1);   // left
            }
            if (done)  // this location is part of the final path
                maze.markPath(row, column);
        }

        return done;
    }
}
```

Let's think this through recursively. The maze can be traversed successfully if it can be traversed successfully from position (0, 0). Therefore, the maze can be traversed successfully if it can be traversed successfully from any position adjacent to (0, 0)—namely, position (1, 0), position (0, 1), position (–1, 0), or position (0, –1). Picking a potential next step, say (1, 0), we find ourselves in the same type of situation as before. To traverse the maze successfully from the new current position, we must successfully traverse it from an adjacent position. At any point, some of the adjacent positions may be invalid, may be blocked, or may represent a possible successful path. We continue this process recursively. If the base case position is reached, the maze has been traversed successfully.

TRAVERSE = recursive

The recursive method in the `MazeSolver` class is called `traverse`. It returns a `boolean` value that indicates whether a solution was found. First the method determines whether a move to the specified row and column is valid. A move is considered valid if it stays within the grid boundaries and if the grid contains a 1 in that location, indicating that a move in that direction is not blocked. The initial call to `traverse` passes in the upper-left location (0, 0).

If the move is valid, the grid entry is changed from a 1 to a 2, marking this location as visited so that we don't retrace our steps later. Then the `traverse` method determines whether the maze has been completed by having reached the bottom-right location. Therefore, there are actually three possibilities of the base case for this problem that will terminate any particular recursive path:

- An invalid move because the move is out of bounds or blocked
- An invalid move because the move has been tried before
- A move that arrives at the final location

If the current location is not the bottom-right corner, we search for a solution in each of the primary directions. First, we look down by recursively calling the `traverse` method and passing in the new location. The logic of the `traverse` method starts all over again using this new position. Either a solution is ultimately found by first attempting to move down from the current location, or it is not found. If it's not found, we try moving right. If that fails, we try moving up. Finally, if no other direction has yielded a correct path, we try moving left. If no direction from the current location yields a correct solution, then there is no path from this location, and `traverse` returns false. If the very first invocation of the `traverse` method returns false, then there is no possible path through this maze.

If a solution is found from the current location, then the grid entry is changed to a 3. The first 3 is placed in the bottom-right corner. The next 3 is placed in the location that led to the bottom-right corner, and so on until the final 3 is placed in the upper-left corner. Therefore, when the final maze is printed, 0 still indicates a blocked path, 1 indicates an open path that was never tried, 2 indicates a path

that was tried but failed to yield a correct solution, and 3 indicates a part of the final solution of the maze.

Here are a sample maze input file and its corresponding output:

```
5 5
1 0 0 0 0
1 1 1 1 0
0 1 0 0 0
1 1 1 1 0
0 1 0 1 1

3 0 0 0 0
3 3 1 1 0
0 3 0 0 0
1 3 3 3 0
0 2 0 3 3
```

Note that there are several opportunities for recursion in each call to the `traverse` method. Any or all of them might be followed, depending on the maze configuration. Although there may be many paths through the maze, the recursion terminates when a path is found. Carefully trace the execution of this code, while following the maze array to see how the recursion solves the problem. Then consider the difficulty of producing a nonrecursive solution.

The Towers of Hanoi

The *Towers of Hanoi* puzzle was invented in the 1880s by Edouard Lucas, a French mathematician. It has become a favorite among computer scientists because its solution is an excellent demonstration of recursive elegance.

The puzzle consists of three upright pegs (towers) and a set of disks with holes in the middle so that they slide onto the pegs. Each disk has a different diameter. Initially, all of the disks are stacked on one peg in order of size such that the largest disk is on the bottom, as shown in Figure 8.6.

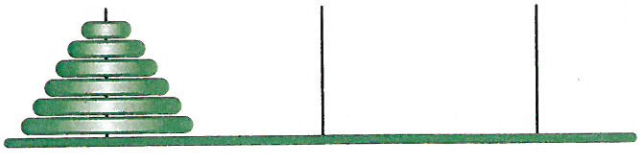

FIGURE 8.6 The Towers of Hanoi puzzle

The goal of the puzzle is to move all of the disks from their original (first) peg to the destination (third) peg. We can use the "extra" peg as a temporary place to put disks, but we must obey the following three rules:

- We can move only one disk at a time.
- We cannot place a larger disk on top of a smaller disk.
- All disks must be on some peg except for the disk that is in transit between pegs.

These rules imply that we must move smaller disks "out of the way" in order to move a larger disk from one peg to another. Figure 8.7 shows the step-by-step solution for the Towers of Hanoi puzzle using three disks. To move all three disks from the first peg to the third peg, we first have to get to the point where the smaller two disks are out of the way on the second peg so that the largest disk can be moved from the first peg to the third peg.

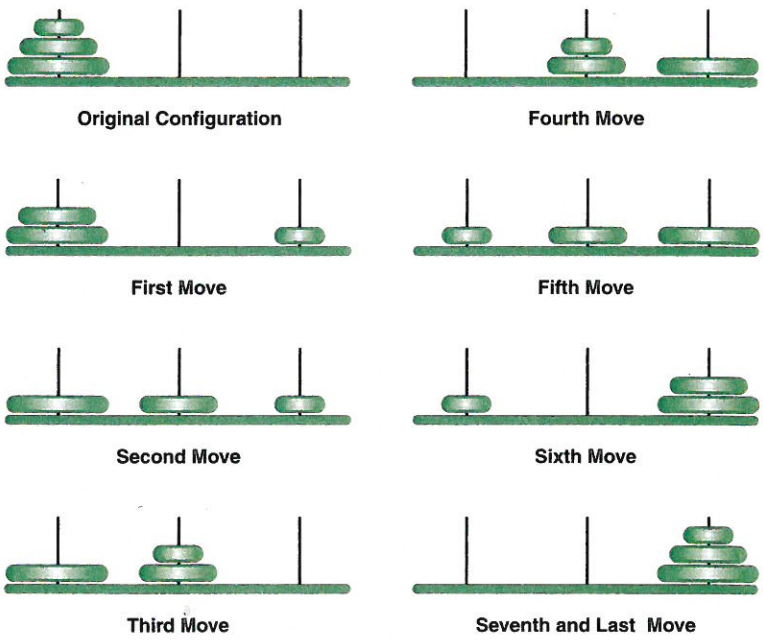

FIGURE 8.7 A solution to the three-disk Towers of Hanoi puzzle

The first three moves shown in Figure 8.7 can be thought of as "moving the smaller disks out of the way." The fourth move puts the largest disk in its final place. The last three moves put the smaller disks in their final place on top of the largest one.

Let's use this idea to form a general strategy. To move a stack of N disks from the original peg to the destination peg:

- Move the topmost N–1 disks from the original peg to the extra peg.
- Move the largest disk from the original peg to the destination peg.
- Move the N–1 disks from the extra peg to the destination peg.

This strategy lends itself nicely to a recursive solution. The step to move the N–1 disks out of the way is the same problem all over again: moving a stack of disks. For this subtask, though, there is one less disk, and our destination peg is what we were originally calling the extra peg. An analogous situation occurs after we have moved the largest disk, and we have to move the original N–1 disks again.

The base case for this problem occurs when we want to move a "stack" that consists of only one disk. That step can be accomplished directly and without recursion.

The program in Listing 8.4 creates a `TowersOfHanoi` object and invokes its `solve` method. The output is a step-by-step list of instructions that describes how the disks should be moved to solve the puzzle. This example uses four disks, which is specified by a parameter to the `TowersOfHanoi` constructor.

LISTING 8.4

```
/**
 * SolveTowers uses recursion to solve the Towers of Hanoi puzzle.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class SolveTowers
{
    /**
     * Creates a TowersOfHanoi puzzle and solves it.
     */
    public static void main(String[] args)
    {
        TowersOfHanoi towers = new TowersOfHanoi(4);
        towers.solve();
    }
}
```

The `TowersOfHanoi` class, shown in Listing 8.5, uses the `solve` method to make an initial call to `moveTower`, the recursive method. The initial call indicates that all of the disks should be moved from peg 1 to peg 3, using peg 2 as the extra position.

The `moveTower` method first considers the base case (a "stack" of one disk). When that occurs, it calls the `moveOneDisk` method, which prints a single line describing that particular move. If the stack contains more than one disk, we call `moveTower` again to get the N–1 disks out of the way, then move the largest disk, then move the N–1 disks to their final destination with yet another call to `moveTower`.

LISTING 8.5

```
/**
 * TowersOfHanoi represents the classic Towers of Hanoi puzzle.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class TowersOfHanoi
{
    private int totalDisks;

    /**
     * Sets up the puzzle with the specified number of disks.
     *
     * @param disks the number of disks
     */
    public TowersOfHanoi(int disks)
    {
        totalDisks = disks;
    }

    /**
     * Performs the initial call to moveTower to solve the puzzle.
     * Moves the disks from tower 1 to tower 3 using tower 2.
     */
    public void solve()
    {
        moveTower(totalDisks, 1, 3, 2);
    }
```

LISTING 8.5 *continued*

```
    /**
     * Moves the specified number of disks from one tower to another
     * by moving a subtower of n-1 disks out of the way, moving one
     * disk, then moving the subtower back. Base case of 1 disk.
     *
     * @param numDisks  the number of disks to move
     * @param start     the starting tower
     * @param end       the ending tower
     * @param temp      the temporary tower
     */
    private void moveTower(int numDisks, int start, int end, int temp)
    {
        if (numDisks == 1)
            moveOneDisk(start, end);
        else
        {
            moveTower(numDisks-1, start, temp, end);
            moveOneDisk(start, end);
            moveTower(numDisks-1, temp, end, start);
        }
    }

    /**
     * Prints instructions to move one disk from the specified start
     * tower to the specified end tower.
     *
     * @param start  the starting tower
     * @param end    the ending tower
     */
    private void moveOneDisk(int start, int end)
    {
        System.out.println("Move one disk from " + start + " to " + end);
    }
}
```

Note that the parameters to `moveTower` describing the pegs are switched around as needed to move the partial stacks. This code follows our general strategy and uses the `moveTower` method to move all partial stacks. Trace the code carefully for a stack of three disks to understand the processing. Figure 8.8 shows the UML diagram for this problem.

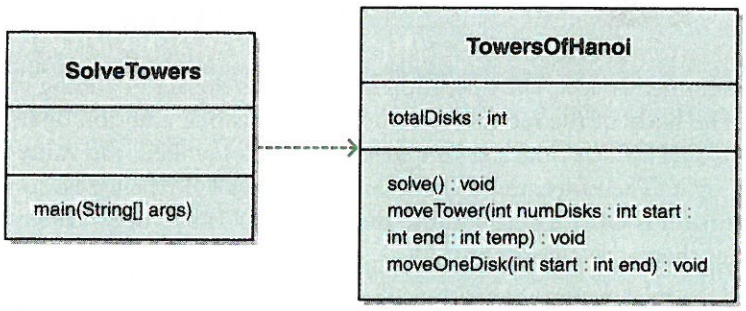

FIGURE 8.8 UML description of Towers of Hanoi puzzle solution

8.4 Analyzing Recursive Algorithms

In Chapter 2, we explored the concept of analyzing an algorithm to determine its complexity (usually its time complexity) and expressed it in terms of a growth function. The growth function gave us the order of the algorithm, which can be used to compare it to other algorithms that accomplish the same task.

When analyzing a loop, we determined the order of the body of the loop and multiplied it by the number of times the loop was executed. Analyzing a recursive algorithm uses similar thinking. Determining the order of a recursive algorithm is a matter of determining the order of the recursion (the number of times the recursive definition is followed) and multiplying that by the order of the body of the recursive method.

KEY CONCEPT
The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.

Consider the recursive method presented in Section 8.2 that computes the sum of the integers from 1 to some positive value. We reprint it here for convenience:

```
// This method returns the sum of 1 to num

public int sum (int num)
{
   int result;
   if (num == 1)
      result = 1;
   else
      result = num + sum (num-1);
   return result;
}
```

The size of this problem is naturally expressed as the number of values to be summed. Because we are summing the integers from 1 to num, the number of values to be summed is num. The operation of interest is the act of adding two values together. The body of the recursive method performs one addition operation and therefore is O(1). Each time the recursive method is invoked, the value of num is decreased by 1. Therefore, the recursive method is called num times, so the order of the recursion is O(n). Thus, because the body is O(1) and the recursion is O(n), the order of the entire algorithm is O(n).

We will see that in some algorithms the recursive step operates on half as much data as the previous call, thus creating an order of recursion of O(log n). If the body of the method is O(1), then the whole algorithm is O(log n). If the body of the method is O(n), then the whole algorithm is O(n log n).

Now consider the Towers of Hanoi puzzle. The size of the puzzle is naturally the number of disks, and the processing operation of interest is the step of moving one disk from one peg to another. Each call to the recursive method `moveTower` results in one disk being moved. Unfortunately, except for the base case, each recursive call results in calling itself *twice more*, and each call operates on a stack of disks that is only one less than the stack that is passed in as the parameter. Thus, calling `moveTower` with 1 disk results in 1 disk being moved, calling `moveTower` with 2 disks results in 3 disks being moved, calling `moveTower` with 3 disks results in 7 disks being moved, calling `moveTower` with 4 disks results in 15 disks being moved, and so on. Looking at it another way, if `f(n)` is the growth function for this problem, then

KEY CONCEPT

The Towers of Hanoi solution has exponential complexity, which is very inefficient, yet the code is remarkably short and elegant.

$$f(n) = 1 \text{ when n is equal to 1}$$

$$\text{for } n > 1,$$

$$f(n) = 2*(f(n-1)+1)$$

$$= 2^n - 1$$

VideoNote

Analyzing recursive algorithms

Contrary to its short and elegant implementation, the solution to the Towers of Hanoi puzzle is terribly inefficient. To solve the puzzle with a stack of n disks, we have to make $2^n - 1$ individual disk moves. Therefore, the Towers of Hanoi algorithm is $O(2^n)$. This order is an example of exponential complexity. As the number of disks increases, the number of required moves increases exponentially.

Legend has it that priests of Brahma are working on this puzzle in a temple at the center of the world. They are using 64 gold disks, moving them between pegs of pure diamond. The downside is that when the priests finish the puzzle, the world will end. The upside is that even if they move one disk every second of every day, it will take them over 584 billion years to complete it. That's with a puzzle of only 64 disks! It is certainly an indication of just how intractable exponential algorithm complexity is.

Summary of Key Concepts

- Recursion is a programming technique in which a method calls itself. A key to being able to program recursively is to be able to think recursively.
- Any recursive definition must have a nonrecursive part, called the base case, that permits the recursion to eventually end.
- Mathematical problems and formulas are often expressed recursively.
- Each recursive call to a method creates new local variables and parameters.
- A careful trace of recursive processing can provide insight into the way it is used to solve a problem.
- Recursion is the most elegant and appropriate way to solve some problems, but for others it is less intuitive than an iterative solution.
- The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.
- The Towers of Hanoi solution has exponential complexity, which is very inefficient, yet the code is incredibly short and elegant.

Summary of Terms

base case The part of an operation's definition that is not recursive.

direct recursion The type of recursion in which a method invokes itself directly (as opposed to indirect recursion).

indirect recursion The type of recursion in which a method calls another method, which may call yet another, and so on until the original method is called (as opposed to direct recursion).

infinite recursion The problem that occurs when a base case is never reached or not defined for an operation.

recursion A programming technique in which a method calls itself to fulfill its overall purpose.

Towers of Hanoi A classic computing puzzle in which the goal is to move disks from one tower to another under specific rules.

Self-Review Questions

SR 8.1 What is recursion?

SR 8.2 What is infinite recursion?