Recursion

CHAPTER 7

recursion *n. See* recursion.
—*Eric S. Raymond,* The New Hacker's Dictionary *(1991)*

Chapter Goals

- Understanding recursion
- Recursive methods
- Recursion in two-dimensional grids
- Recursive helper methods
- Analysis of recursive algorithms
- Tracing recursive algorithms

In the multiple-choice section of the AP exam, you will be asked to understand and trace recursive methods. You will not, however, be asked to come up with code for recursive methods in the free-response part of the exam.

RECURSIVE METHODS

A *recursive method* is a method that calls itself. For example, here is a program that calls a recursive method stackWords.

```
public class WordPlay
{
    public static void stackWords()
    {
        String word = IO.readString();     //read user input
        if (word.equals("."))
            System.out.println();
        else
            stackWords();
        System.out.println(word);
    }

    public static void main(String args[])
    {
        System.out.println("Enter list of words, one per line.");
        System.out.println("Final word should be a period (.)");
        stackWords();
    }
}
```

Here is the output if you enter

```
hold
my
hand
.
```

You get

```
.
hand
my
hold
```

The program reads in a list of words terminated with a period, and prints the list in reverse order, starting with the period. How does this happen?

Each time the recursive call to `stackWords()` is made, execution goes back to the start of a new method call. The computer must remember to complete all the pending calls to the method. It does this by stacking the statements that must still be executed as follows: The first time `stackWords()` is called, the word `"hold"` is read and tested for being a period. No it's not, so `stackWords()` is called again. The statement to output `"hold"` (which has not yet been executed) goes on a stack, and execution goes to the start of the method. The word `"my"` is read. No, it's not a period, so the command to output `"my"` goes on the stack. And so on. The stack looks something like this before the recursive call in which the period is read:

`System.out.println("hand");`
`System.out.println("my");`
`System.out.println("hold");`

Imagine that these statements are stacked like plates. In the final `stackWords()` call, `word` has the value `"."`. Yes, it *is* a period, so the `stackWords()` line is skipped, the period is printed on the screen, and the method call terminates. The computer now completes each of the previous method calls in turn by "popping" the statements off the top of the stack. It prints `"hand"`, then `"my"`, then `"hold"`, and execution of method `stackWords()` is complete.[1]

NOTE

1. Each time `stackWords()` is called, a new local variable `word` is created.
2. The first time the method actually terminates, the program returns to complete the most recently invoked previous call. That's why the words get reversed in this example.

GENERAL FORM OF SIMPLE RECURSIVE METHODS

Every recursive method has two distinct parts:

- A base case or termination condition that causes the method to end.
- A nonbase case whose actions move the algorithm toward the base case and termination.

[1]Actually, the computer stacks the pending statements in a recursive method call more efficiently than the way described. But *conceptually* this is how it is done.

Here is the framework for a simple recursive method that has no specific return type.

```
public void recursiveMeth( ... )
{
    if (base case)
        < Perform some action >
    else
    {
        < Perform some other action >
        recursiveMeth( ... );    //recursive method call
    }
}
```

The base case typically occurs for the simplest case of the problem, such as when an integer has a value of 0 or 1. Other examples of base cases are when some key is found, or an end-of-file is reached. A recursive algorithm can have more than one base case.

In the `else` or nonbase case of the framework shown, the code fragment < *Perform some other action* > and the method call `recursiveMeth` can sometimes be interchanged without altering the net effect of the algorithm. Be careful though, because what *does* change is the order of executing statements. This can sometimes be disastrous. (See the `eraseBlob` example on p. 299.)

Example 1

```
public void drawLine(int n)
{
    if (n == 0)
        System.out.println("That's all, folks!");
    else
    {
        for (int i = 1; i <= n; i++)
            System.out.print("*");
        System.out.println();
        drawLine(n - 1);
    }
}
```

The method call `drawLine(3)` produces this output:

```
***
**
*
That's all, folks!
```

NOTE

1. A method that has no pending statements following the recursive call is an example of *tail recursion*. Method `drawLine` is such a case, but `stackWords` is not.
2. The base case in the `drawLine` example is `n == 0`. Notice that each subsequent call, `drawLine(n - 1)`, makes progress toward termination of the method. If your method has no base case, or if you never reach the base case, you will create *infinite recursion*. This is a catastrophic error that will cause your computer eventually to run out of memory and give you heart-stopping messages like `java.lang.StackOverflowError ...`

Example 2

```
//Illustrates infinite recursion.
public void catastrophe(int n)
{
    System.out.println(n);
    catastrophe(n);
}
```

Try running the case `catastrophe(1)` if you have lots of time to waste!

A recursive method must have a base case.

WRITING RECURSIVE METHODS

Optional topic

To come up with a recursive algorithm, you have to be able to frame a process *recursively* (i.e., in terms of a simpler case of itself). This is different from framing it *iteratively*, which repeats a process until a final condition is met. A good strategy for writing recursive methods is to first state the algorithm recursively in words.

Example 1

Write a method that returns $n!$ (n factorial).

***n*! defined iteratively**	***n*! defined recursively**
$0! = 1$	$0! = 1$
$1! = 1$	$1! = (1)(0!)$
$2! = (2)(1)$	$2! = (2)(1!)$
$3! = (3)(2)(1)$	$3! = (3)(2!)$
...	...

The general recursive definition for $n!$ is

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

The definition seems to be circular until you realize that if $0!$ is defined, all higher factorials are defined. Code for the recursive method follows directly from the recursive definition:

```
/** Compute n! recursively.
 *  @param n a nonnegative integer
 *  @return n!
 */
public static int factorial(int n)
{
    if (n == 0)     //base case
        return 1;
    else
        return n * factorial(n - 1);
}
```

Example 2

Write a recursive method `revDigs` that outputs its integer parameter with the digits reversed. For example,

(continued)

```
revDigs(147)   outputs   741
revDigs(4)     outputs   4
```

First, describe the process recursively: Output the rightmost digit. Then, if there are still digits left in the remaining number `n/10`, reverse its digits. Repeat this until `n/10` is `0`. Here is the method:

```
/** @param n a nonnegative integer
 *  @return n with its digits reversed
 */
public static void revDigs(int n)
{
    System.out.print(n % 10);  //rightmost digit
    if (n / 10 != 0)              //base case
        revDigs(n / 10);
}
```

NOTE

On the AP exam, you are expected to "understand and evaluate" recursive methods. This means that you would not be asked to come up with the code for methods such as `factorial` and `revDigs` (as shown above). You could, however, be asked to identify output for any given call to `factorial` or `revDigs`.

ANALYSIS OF RECURSIVE METHODS

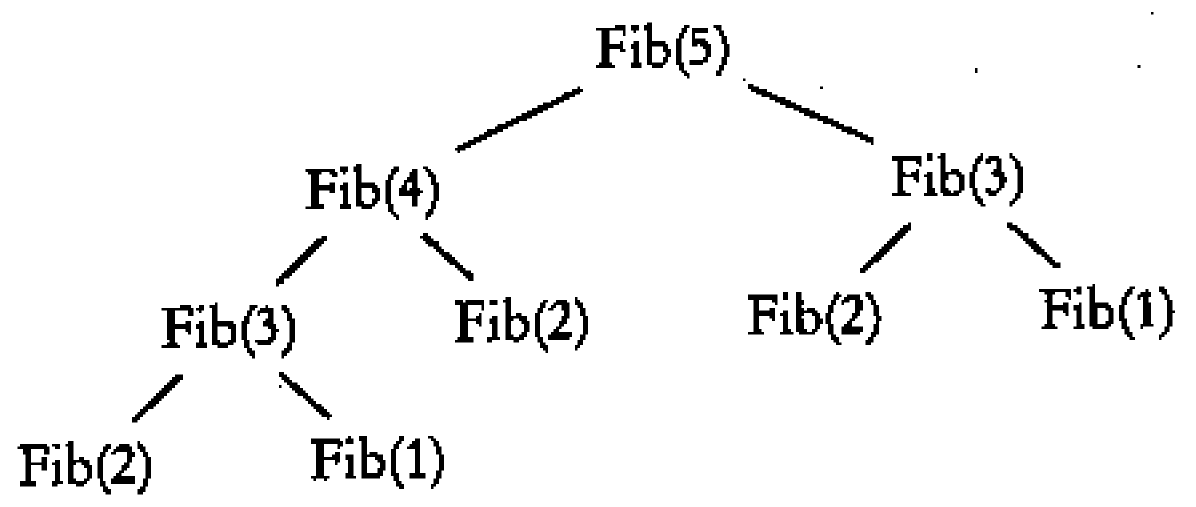
Recall the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ... The nth Fibonacci number equals the sum of the previous two numbers if $n \geq 3$. Recursively,

$$\mathrm{Fib}(n) = \begin{cases} 1, & n = 1,2 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2), & n \geq 3 \end{cases}$$

Here is the method:

```
/** @param n a positive integer
 *  @return the nth Fibonacci number
 */
public static int fib(int n)
{
    if (n == 1 || n == 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Notice that there are two recursive calls in the last line of the method. So to find Fib(5), for example, takes eight recursive calls to `fib`!



In general, each call to `fib` makes two more calls, which is the tipoff for an exponential algorithm (i.e., one that is *very* inefficient). This is *much* slower than the run time of the corresponding iterative algorithm (see Chapter 5, Question 13).

You may ask: Since every recursive algorithm can be written iteratively, when should programmers use recursion? Bear in mind that recursive algorithms can incur extra run time and memory. Their major plus is elegance and simplicity of code.

General Rules for Recursion

1. Avoid recursion for algorithms that involve large local arrays—too many recursive calls can cause memory overflow.
2. Use recursion when it significantly simplifies code.
3. Avoid recursion for simple iterative methods like factorial, Fibonacci, and the linear search on the next page.
4. Recursion is especially useful for
   - Branching processes like traversing trees or directories.
   - Divide-and-conquer algorithms like mergesort and binary search.

SORTING ALGORITHMS THAT USE RECURSION

Mergesort and quicksort are discussed in Chapter 8.

RECURSIVE HELPER METHODS

Optional topic

A common technique in designing recursive algorithms is to have a public nonrecursive driver method that calls a private *recursive helper method* to carry out the task. The main reasons for doing this are

- To hide the implementation details of the recursion from the user.
- To enhance the efficiency of the program.

Example 1

Consider the simple example of recursively finding the sum of the first n positive integers.

```
/** @param n a positive integer
 *  @return 1 + 2 + 3 + ... + n
 */
public static int sum(int n)
{
    if (n == 1)
        return 1;
    else
        return n + sum(n - 1);
}
```

(continued)

Notice that you get infinite recursion if $n \leq 0$. Suppose you want to include a test for $n > 0$ before you execute the algorithm. Placing this test in the recursive method is inefficient because if n is initially positive, it will remain positive in subsequent recursive calls. You can avoid this problem by using a driver method called getSum, which does the test on *n just once*. The recursive method sum becomes a private helper method.

```
public class FindSum
{
    /** Private recursive helper method.
     *  @param n a positive integer
     *  @return 1 + 2 + 3 + ... + n
     */
    private static int sum(int n)
    {
        if (n == 1)
            return 1;
        else
            return n + sum(n - 1);
    }

    /* Driver method */
    public static int getSum(int n)
    {
        if (n > 0)
            return sum(n);
        else
        {
            throw new IllegalArgumentException
                    ("Error: n must be positive");
        }
    }
}
```

NOTE

This is a trivial method used to illustrate a private recursive helper method. In practice, you would never use recursion to find a simple sum!

Example 2

Consider a recursive solution to the problem of doing a sequential search for a key in an array of strings. If the key is found, the method returns `true`, otherwise it returns `false`.

The solution can be stated recursively as follows:

- If the key is in `a[0]`, then the key is found.
- If not, recursively search the array starting at `a[1]`.
- If you are past the end of the array, then the key wasn't found.

Here is a straightforward (but inefficient) implementation:

(continued)

```
public class Searcher
{
    /** Recursively search array a for key.
     * @param a the array of String objects
     * @param key a String object
     * @return true if a[k] equals key for 0 <= k < a.length;
     * false otherwise
     */
    public boolean search(String[] a, String key)
    {
        if (a.length == 0)  //base case. key not found
            return false;
        else if (a[0].compareTo(key) == 0) //base case
            return true;                   //key found
        else
        {
            String[] shorter = new String[a.length-1];
            for (int i = 0; i < shorter.length; i++)
                shorter[i] = a[i+1];
            return search(shorter, key);
        }
    }

    public static void main(String[] args)
    {
        String[] list = {"Mary", "Joe", "Lee", "Jake"};
        Searcher s = new Searcher();
        System.out.println("Enter key: Mary, Joe, Lee or Jake.");
        String key = IO.readString();  //read user input
        boolean result = s.search(list, key);
        if (!result)
            System.out.println(key + " was not found.");
        else
            System.out.println(key + " was found.");
    }
}
```

Notice how horribly inefficient the search method is: For each recursive call, a new array shorter has to be created! It is much better to use a parameter, startIndex, to keep track of where you are in the array. Replace the search method above with the following one, which calls the private helper method recurSearch:

```
/** Driver method. Searches array a for key.
 * Precondition: a contains at least one element.
 * @param a the array of String objects
 * @param key a String object
 * @return true if a[k] equals key for 0 <= k < a.length;
 * false otherwise
 */
public boolean search(String[] a, String key)
{
    return recurSearch(a, 0, key);
}
```

(continued)

```
/** Recursively search array a for key, starting at startIndex.
 *  Precondition:
 *   - a contains at least one element.
 *   - 0 <= startIndex <= a.length.
 *  @return true if a[k] equals key for 0 <= k < a.length;
 *  false otherwise
 */
private boolean recurSearch(String[] a, int startIndex,
    String key)
{
    if(startIndex == a.length)   //base case. key not found
        return false;
    else if(a[startIndex].compareTo(key) == 0) //base case
        return true;                           //key found
    else
        return recurSearch(a, startIndex+1, key);
}
```

NOTE

Use a recursive helper method to hide private coding details from a client.

1. Using the parameter `startIndex` avoids having to create a new array object for each recursive call. Making `startIndex` a parameter of a helper method hides implementation details from the user.
2. The helper method is private because it is called only by `search` within the `Searcher` class.
3. It's easy to modify the `search` method to return the index in the array where the key is found: Make the return type `int` and return `startIndex` if the key is found, `-1` (say) if it isn't.

RECURSION IN TWO-DIMENSIONAL GRIDS

Here is a commonly used technique: using recursion to traverse a two-dimensional array. The problem comes in several different guises, for example,

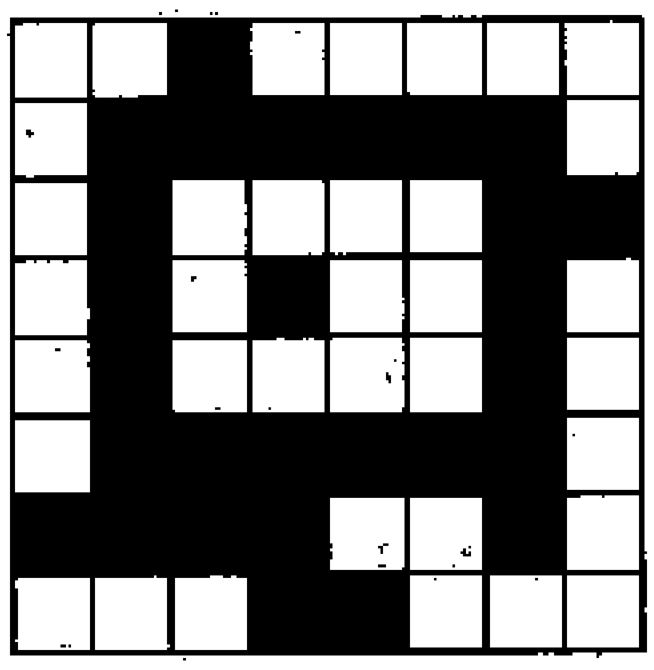
1. A game board from which you must remove pieces.
2. A maze with walls and paths from which you must try to escape.
3. White "containers" enclosed by black "walls" into which you must "pour paint."

In each case, you will be given a starting position (row, col) and instructions on what to do. The recursive solution typically involves these steps:

Check that the starting position is not out of range:
  If (starting position satisfies some requirement)
    Perform some action to solve problem
    RecursiveCall(row + 1, col)
    RecursiveCall(row − 1, col)
    RecursiveCall(row, col + 1)
    RecursiveCall(row, col − 1)

(continued)

Example



On the right is an image represented as a square grid of black and white cells. Two cells in an image are part of the same "blob" if each is black and there is a sequence of moves from one cell to the other, where each move is either horizontal or vertical to an adjacent black cell. For example, the diagram represents an image that contains two blobs, one of them consisting of a single cell.

Assuming the following `Image` class declaration, you are to write the body of the `eraseBlob` method, using a recursive algorithm.

```
public class Image
{
    private final int BLACK = 1;
    private final int WHITE = 0;
    private int[][] image;   //square grid
    private int size;        //number of rows and columns

    public Image()   //constructor
    { /* implementation not shown */ }

    public void display()   //displays Image
    { /* implementation not shown */ }

    /** Precondition: Image is defined with either BLACK or WHITE cells.
     *  Postcondition: If 0 <= row < size, 0 <= col < size, and
     *                 image[row][col] is BLACK, set all cells in the
     *                 same blob to WHITE. Otherwise image is unchanged.
     *  @param row the given row
     *  @param col the given column
     */
    public void eraseBlob(int row, int col)
    /* your code goes here */
}
```

Solution:

```
public void eraseBlob(int row, int col)
{
    if (row >= 0 && row < size && col >= 0 && col < size)
        if (image[row][col] == BLACK)
        {
            image[row][col] = WHITE;
            eraseBlob(row - 1, col);
            eraseBlob(row + 1, col);
            eraseBlob(row, col - 1);
            eraseBlob(row, col + 1);
        }
}
```

NOTE

1. The ordering of the four recursive calls is irrelevant.

(continued)

2. The test

```
if (image[row][col] == BLACK)
```

can be included as the last piece of the test in the first line:

```
if (row >= 0 && ...
```

If `row` or `col` is out of range, the test will short-circuit, avoiding the dreaded `ArrayIndexOutOfBoundsException`.

3. If you put the statement

```
image[row][col] = WHITE;
```

after the four recursive calls, you get infinite recursion if your blob has more than one cell. This is because, when you visit an adjacent cell, one of its recursive calls visits the original cell. If this cell is still `BLACK`, yet more recursive calls are generated, *ad infinitum*.

A final thought: Recursive algorithms can be tricky. Try to state the solution recursively *in words* before you launch into code. Oh, and don't forget the base case!

Sample Free-Response Question 1

Here is a sample free-response question that uses recursion in a two-dimensional array. See if you can answer it before looking at the solution.

A *color grid* is defined as a two-dimensional array whose elements are character strings having values "b" (blue), "r" (red), "g" (green), or "y" (yellow). The elements are called pixels because they represent pixel locations on a computer screen. For example,

```
b b g r      r r r r r      y g r
g r g r                     b y g
                            g r b
                            b b g
```

A *connected region* for any pixel is the set of all pixels of the same color that can be reached through a direct path along horizontal or vertical moves starting at that pixel. A connected region can consist of just a single pixel or the entire color grid. For example, if the two-dimensional array is called `pixels`, the connected region for `pixels[1][0]` is as shown here for three different arrays.

```
b b g r      y g r b      b b
g r g r      g g y g      b b
             b g r g
```

The class `ColorGrid`, whose declaration is shown below, is used for storing, displaying, and changing the colors in a color grid.

(continued)

```
public class ColorGrid
{
    private String[][] pixels;
    private int rows;
    private int cols;

    /** Creates numRows × numCols ColorGrid from String s.
     *  @param s the string containing colors of the ColorGrid
     *  @param numRows the number of rows in the ColorGrid
     *  @param numCols the number of columns in the ColorGrid
     */
    public ColorGrid(String s, int numRows, int numCols)
    { /* to be implemented in part (a) */ }

    /** Precondition:
     *   - pixels[row][col] is oldColor, one of "r", "b","g", or "y".
     *   - newColor is one of "r","b","g", or "y".
     *  Postcondition:
     *   - If 0 <= row < rows and 0 <= col < cols, paints the
     *     connected region of pixels[row][col] the newColor.
     *   - Does nothing if oldColor is the same as newColor.
     * @param row the given row
     * @param col the given column
     * @param newColor the new color for painting
     * @param oldColor the current color of pixels[row][col]
     */
    public void paintRegion(int row, int col, String newColor,
        String oldColor)
    { /* to be implemented in part (b) */ }

    //Other methods are not shown.
}
```

(a) Write the implementation code for the `ColorGrid` constructor. The constructor should initialize the `pixels` matrix of the `ColorGrid` as follows: The dimensions of `pixels` are `numRows` × `numCols`. `String s` contains `numRows` × `numCols` characters, where each character is one of the colors of the grid—`"r"`, `"g"`, `"b"`, or `"y"`. The characters are contained in `s` row by row from top to bottom and left to right. For example, given that `numRows` is 3, and `numCols` is 4, if `s` is `"brrygrggyyyr"`, `pixels` should be initialized to be

```
b r r y
g r g g
y y y r
```

Complete the constructor below:

```
/** Creates numRows × numCols ColorGrid from String s.
 *  @param s the string containing colors of the ColorGrid
 *  @param numRows the number of rows in the ColorGrid
 *  @param numCols the number of columns in the ColorGrid
 */
public ColorGrid(String s, int numRows, int numCols)
```

(continued)

(b) Write the implementation of the `paintRegion` method as started below. **Note: You must write a recursive solution.** The `paintRegion` paints the connected region of the given pixel, specified by `row` and `col`, a different color specified by the `newColor` parameter. If `newColor` is the same as `oldColor`, the color of the given pixel, `paintRegion` does nothing. To visualize what `paintRegion` does, imagine that the different colors surrounding the connected region of a given pixel form a boundary. When paint is poured onto the given pixel, the new color will fill the connected region up to the boundary.

For example, the effect of the method call `c.paintRegion(2, 3, "b", "r")` on the `ColorGrid c` is shown here. (The starting pixel is shown in a frame, and its connected region is shaded.)

before

r	r	b	g	y	y
b	r	b	y	r	r
g	g	r	r	r	b
y	r	r	y	r	b

after

r	r	b	g	y	y
b	r	b	y	b	b
g	g	b	b	b	b
y	b	b	y	b	b

Complete the method `paintRegion` below. **Note: Only a recursive solution will be accepted.**

```
/** Precondition:
 *    - pixels[row][col] is oldColor, one of "r", "b","g", or "y".
 *    - newColor is one of "r","b","g", or "y".
 *  Postcondition:
 *    - If 0 <= row < rows and 0 <= col < cols, paints the
 *      connected region of pixels[row][col] the newColor.
 *    - Does nothing if oldColor is the same as newColor.
 *  @param row the given row
 *  @param col the given column
 *  @param newColor the new color for painting
 *  @param oldColor the current color of pixels[row][col]
 */
public void paintRegion(int row, int col, String newColor,
    String oldColor)
```

Solution

(a)
```
public ColorGrid(String s, int numRows, int numCols)
{
    rows = numRows;
    cols = numCols;
    pixels = new String[numRows][numCols];
    int stringIndex = 0;
    for (int r = 0; r < numRows; r++)
        for (int c = 0; c < numCols; c++)
        {
            pixels[r][c] = s.substring(stringIndex,
                stringIndex + 1);
            stringIndex++;
        }
}
```

(continued)

(b)
```
public void paintRegion(int row, int col, String newColor,
        String oldColor)
{
    if (row >= 0 && row < rows && col >= 0 && col < cols)
        if (!pixels[row][col].equals(newColor) &&
            pixels[row][col].equals(oldColor))
        {
            pixels[row][col] = newColor;
            paintRegion(row + 1, col, newColor, oldColor);
            paintRegion(row - 1, col, newColor, oldColor);
            paintRegion(row, col + 1, newColor, oldColor);
            paintRegion(row, col - 1, newColor, oldColor);
        }
}
```

NOTE

- In part (a), you don't need to test if `stringIndex` is in range: The precondition states that the number of characters in `s` is `numRows` × `numCols`.
- In part (b), each recursive call must test whether `row` and `col` are in the correct range for the `pixels` array; otherwise, your algorithm may sail right off the edge!
- Don't forget to test if `newColor` is different from that of the starting pixel. Method `paintRegion` does nothing if the colors are the same.
- Also, don't forget to test if the current pixel is `oldColor`—you don't want to overwrite *all* the colors, just the connected region of `oldColor`!
- The color-change assignment `pixels[row][col] = newColor` must precede the recursive calls to avoid infinite recursion.

Sample Free-Response Question 2

Here is another sample free-response question that uses recursion.

This question refers to the `Sentence` class below. Note: A *word* is a string of consecutive nonblank (and nonwhitespace) characters. For example, the sentence

"Hello there!" she said.

consists of the four words

```
"Hello    there!"    she    said.
```

(continued)

```
public class Sentence
{
    private String sentence;
    private int numWords;

    /** Constructor. Creates sentence from String str.
     *                Finds the number of words in sentence.
     *  Precondition: Words in str separated by exactly one blank.
     *  @param str the string containing a sentence
     */
    public Sentence(String str)
    { /* to be implemented in part (a) */ }

    public int getNumWords()
    { return numWords; }

    public String getSentence()
    { return sentence; }

    /** @param s the specified string
     *  @return a copy of String s with all blanks removed
     *  Postcondition: Returned string contains just one word.
     */
    private static String removeBlanks(String s)
    { /* implementation not shown */ }

    /** @param s the specified string
     *  @return a copy of String s with all letters in lowercase
     *  Postcondition: Number of words in returned string equals
     *                 number of words in s.
     */
    private static String lowerCase(String s)
    { /* implementation not shown */ }

    /** @param s the specified string
     *  @return a copy of String s with all punctuation removed
     *  Postcondition: Number of words in returned string equals
     *                 number of words in s.
     */
    private static String removePunctuation(String s)
    { /* implementation not shown */ }
}
```

(a) Complete the `Sentence` constructor as started below. The constructor assigns `str` to `sentence`. You should write the subsequent code that assigns a value to `numWords`, the number of words in `sentence`.

Complete the constructor below:

(continued)

```
/** Constructor. Creates sentence from String str.
 *                Finds the number of words in sentence.
 *  Precondition: Words in str separated by exactly one blank.
 *  @param str the string containing a sentence
 */
public Sentence(String str)
{
    sentence = str;
```

(b) Consider the problem of testing whether a string is a palindrome. A *palindrome* reads the same from left to right and right to left, ignoring spaces, punctuation, and capitalization. For example,

A Santa lived as a devil at NASA.
Flo, gin is a sin! I golf.
Eva, can I stab bats in a cave?

A public method `isPalindrome` is added to the `Sentence` class. Here is the method and its implementation:

```
/** @return true if sentence is a palindrome, false otherwise
 */
public boolean isPalindrome()
{
    String temp = removeBlanks(sentence);
    temp = removePunctuation(temp);
    temp = lowerCase(temp);
    return isPalindrome(temp, 0, temp.length() - 1);
}
```

The overloaded `isPalindrome` method contained in the code is a private recursive helper method, also added to the `Sentence` class. You are to write the implementation of this method. It takes a "purified" string as a parameter, namely one that has been stripped of blanks and punctuation and is all lowercase letters. It also takes as parameters the first and last index of the string. It returns true if this "purified" string is a palindrome, false otherwise.

A recursive algorithm for testing if a string is a palindrome is as follows:

- If the string has length 0 or 1, it's a palindrome.
- Remove the first and last letters.
- If those two letters are the same, and the remaining string is a palindrome, then the original string is a palindrome. Otherwise it's not.

Complete the `isPalindrome` method below:

```
/** Private recursive helper method that tests whether a substring
 *  of string s is a palindrome.
 *  @param s the given string
 *  @param start the index of the first character of the substring
 *  @param end the index of the last character of the substring
 *  @return true if the substring is a palindrome, false otherwise
 *  Precondition: s contains no spaces, punctuation, or capitals.
 */
private static boolean isPalindrome(String s, int start, int end)
```

(continued)

Solution

(a)
```
public Sentence(String str)
{
    sentence = str;
    numWords = 1;
    int k = str.indexOf(" ");
    while (k != -1)   //while there are still blanks in str
    {
        numWords++;
        str = str.substring(k + 1); //substring after blank
        k = str.indexOf(" ");       //get index of next blank
    }
}
```

(b)
```
private static boolean isPalindrome(String s, int start,
        int end)
{
    if (start >= end)  //substring has length 0 or 1
        return true;
    else
    {
        String first = s.substring(start, start + 1);
        String last = s.substring(end, end + 1);
        if (first.equals(last))
            return isPalindrome(s, start + 1, end - 1);
        else
            return false;
    }
}
```

NOTE

- In part (a), for every occurrence of a blank in `sentence`, `numWords` must be incremented. (Be sure to initialize `numWords` to 1!)
- In part (a), the code locates all the blanks in `sentence` by replacing `str` with the substring that consists of the piece of `str` directly following the most recently located blank.
- Recall that `indexOf` returns `-1` if its `String` parameter does not occur as a substring in its `String` calling object.
- In part (b), the `start` and `end` indexes move toward each other with each subsequent recursive call. This shortens the string to be tested in each call. When `start` and `end` meet, the base case has been reached.
- Notice the private static methods in the `Sentence` class, including the helper method you were asked to write. They are static because they are not invoked by a `Sentence` object (no dot member construct). The only use of these methods is to help achieve the postconditions of other methods in the class.

Chapter Summary

On the AP exam you will be expected to calculate the results of recursive method calls. Recursion becomes second nature when you practice a lot of examples. For the more difficult questions, untangle the statements with either repeated method calls (like that shown in the solution to Question 5 on p. 319), or box diagrams (as shown in the solution to Question 12 on p. 320).

You should understand that recursive algorithms can be *very* inefficient.

MULTIPLE-CHOICE QUESTIONS ON RECURSION

1. Which of the following statements about recursion are true?

   I Every recursive algorithm can be written iteratively.
   II Tail recursion is always used in "divide-and-conquer" algorithms.
   III In a recursive definition, a process is defined in terms of a simpler case of itself.

   (A) I only
   (B) III only
   (C) I and II only
   (D) I and III only
   (E) II and III only

2. Which of the following, when used as the `/* body */` of method `sum`, will enable that method to compute $1+2+\cdots+n$ correctly for any $n > 0$?

```
/** @param n a positive integer
 *  @return 1 + 2 + ... + n
 */
public int sum(int n)
{
    /* body */
}
```

I
```
return n + sum(n - 1);
```

II
```
if (n == 1)
    return 1;
else
    return n + sum(n - 1);
```

III
```
if (n == 1)
    return 1;
else
    return sum(n) + sum(n - 1);
```

   (A) I only
   (B) II only
   (C) III only
   (D) I and II only
   (E) I, II, and III

3. Refer to the method `stringRecur`:

```
public void stringRecur(String s)
{
    if (s.length() < 15)
        System.out.println(s);
    stringRecur(s + "*");
}
```

When will method `stringRecur` terminate without error?
(A) Only when the length of the input string is less than 15
(B) Only when the length of the input string is greater than or equal to 15
(C) Only when an empty string is input
(D) For all string inputs
(E) For no string inputs

4. Refer to method `strRecur`:

```
public void strRecur(String s)
{
    if (s.length() < 15)
    {
        System.out.println(s);
        strRecur(s + "*");
    }
}
```

When will method `strRecur` terminate without error?
(A) Only when the length of the input string is less than 15
(B) Only when the length of the input string is greater than or equal to 15
(C) Only when an empty string is input
(D) For all string inputs
(E) For no string inputs

Questions 5 and 6 refer to method `result`:

```
public int result(int n)
{
    if (n == 1)
        return 2;
    else
        return 2 * result(n - 1);
}
```

5. What value does `result(5)` return?
(A) 64
(B) 32
(C) 16
(D) 8
(E) 2

6. If $n > 0$, how many times will `result` be called to evaluate `result(n)` (including the initial call)?
   (A) 2
   (B) 2^n
   (C) n
   (D) $2n$
   (E) n^2

7. Refer to method `mystery`:

```
public int mystery(int n, int a, int d)
{
    if (n == 1)
        return a;
    else
        return d + mystery(n - 1, a, d);
}
```

   What value is returned by the call `mystery(3, 2, 6)`?
   (A) 20
   (B) 14
   (C) 10
   (D) 8
   (E) 2

8. Refer to method `f`:

```
public int f(int k, int n)
{
    if (n == k)
        return k;
    else
        if (n > k)
            return f(k, n - k);
        else
            return f(k - n, n);
}
```

   What value is returned by the call `f(6, 8)`?
   (A) 8
   (B) 4
   (C) 3
   (D) 2
   (E) 1

9. What does method recur do?

```
/** @param x an array of n integers
 *  @param n a positive integer
 */
public int recur(int[] x, int n)
{
    int t;
    if (n == 1)
        return x[0];
    else
    {
        t = recur(x, n - 1);
        if (x[n-1] > t)
            return x[n-1];
        else
            return t;
    }
}
```

(A) It finds the largest value in x and leaves x unchanged.
(B) It finds the smallest value in x and leaves x unchanged.
(C) It sorts x in ascending order and returns the largest value in x.
(D) It sorts x in descending order and returns the largest value in x.
(E) It returns x[0] or x[n-1], whichever is larger.

10. Which best describes what the printString method below does?

```
public void printString(String s)
{
    if (s.length() > 0)
    {
        printString(s.substring(1));
        System.out.print(s.substring(0, 1));
    }
}
```

(A) It prints string s.
(B) It prints string s in reverse order.
(C) It prints only the first character of string s.
(D) It prints only the first two characters of string s.
(E) It prints only the last character of string s.

11. Refer to the method power:

```
/** @param base a nonzero real number
 *  @param expo an integer
 *  @return base raised to the expo power
 */
public double power(double base, int expo)
{
    if (expo == 0)
        return 1;
    else if (expo > 0)
        return base * power(base, expo - 1);
    else
        return /* code */;
}
```

Which `/* code */` correctly completes method power?
(Recall that $a^{-n} = 1/a^n$, $a \neq 0$; for example, $2^{-3} = 1/2^3 = 1/8$.)

(A) `(1 / base) * power(base, expo + 1)`
(B) `(1 / base) * power(base, expo - 1)`
(C) `base * power(base, expo + 1)`
(D) `base * power(base, expo - 1)`
(E) `(1 / base) * power(base, expo)`

12. Consider the following method:

```
public void doSomething(int n)
{
    if (n > 0)
    {
        doSomething(n - 1);
        System.out.print(n);
        doSomething(n - 1);
    }
}
```

What would be output following the call `doSomething(3)`?

(A) 3211211
(B) 1121213
(C) 1213121
(D) 1211213
(E) 1123211

13. A user enters several positive integers at the keyboard and terminates the list with a sentinel (`-999`). A `writeEven` method reads those integers and outputs the even integers only, in the reverse order that they are read. Thus, if the user enters

```
3 5 14 6 1 8 -999
```

the output for the `writeEven` method will be

```
8 6 14
```

Assume that the user enters at least one positive integer and terminates the list with −999. Here is the method:

```
/** Postcondition: All even integers in the list are output in
 *                 reverse order.
 */
public static void writeEven()
{
    int num = IO.readInt();    //read user input
    if (num != -999)
    {
        /* code */
    }
}
```

Which `/* code */` satisfies the postcondition of method `writeEven`?

```
I   if (num % 2 == 0)
        System.out.print(num + " ");
    writeEven();

II  if (num % 2 == 0)
        writeEven();
    System.out.print(num + " ");

III writeEven();
    if (num % 2 == 0)
        System.out.print(num + " ");
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

14. Refer to the following recursive method.

```
public int mystery(int n)
{
    if (n < 0)
        return 2;
    else
        return mystery(n - 1) + mystery(n - 3);
}
```

What value is returned by the call `mystery(3)`?

(A) 12
(B) 10
(C) 8
(D) 6
(E) 4

Questions 15 and 16 refer to method t:

```
/** @param n a positive integer */
public int t(int n)
{
    if (n == 1 || n == 2)
        return 2 * n;
    else
        return t(n - 1) - t(n - 2);
}
```

15. What will be returned by t(5)?
    (A) 4
    (B) 2
    (C) 0
    (D) −2
    (E) −4

16. For the method call t(6), how many calls to t will be made, including the original call?
    (A) 6
    (B) 7
    (C) 11
    (D) 15
    (E) 25

17. This question refers to methods f1 and f2 that are in the same class:

```
public int f1(int a, int b)
{
    if (a == b)
        return b;
    else
        return a + f2(a - 1, b);
}

public int f2(int p, int q)
{
    if (p < q)
        return p + q;
    else
        return p + f1(p - 2, q);
}
```

What value will be returned by a call to f1(5, 3)?
    (A) 5
    (B) 6
    (C) 7
    (D) 12
    (E) 15

18. Consider method foo:

```
public int foo(int x)
{
    if (x == 1 || x == 3)
        return x;
    else
        return x * foo(x - 1);
}
```

Assuming no possibility of integer overflow, what will be the value of z after execution of the following statement?

```
int z = foo(foo(3) + foo(4));
```

(A) $(15!)/(2!)$
(B) $3!+4!$
(C) $(7!)!$
(D) $(3!+4!)!$
(E) 15

Questions 19 and 20 refer to the IntFormatter class below.

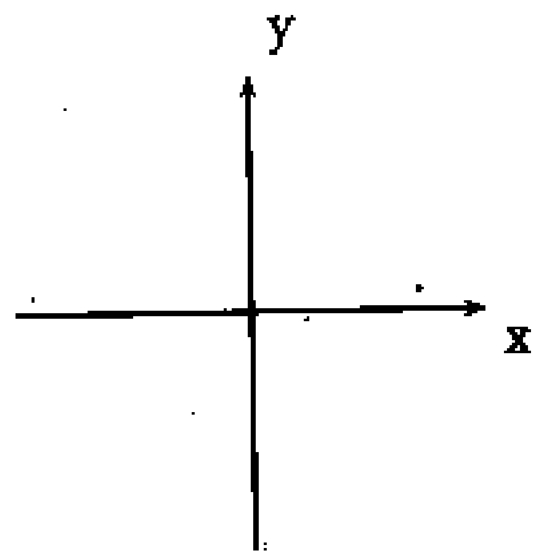
```
public class IntFormatter
{
    /** Write 3 digits adjacent to each other.
     *  @param n a nonnegative integer
     */
    public static void writeThreeDigits(int n)
    {
        System.out.print(n / 100);
        System.out.print((n / 10) % 10);
        System.out.print(n % 10);
    }

    /** Insert commas in n, every 3 digits starting at the right.
     *  @param n a nonnegative integer
     */
    public static void writeWithCommas(int n)
    {
        if (n < 1000)
            System.out.print(n);
        else
        {
            writeThreeDigits(n % 1000);
            System.out.print(",");
            writeWithCommas(n / 1000);
        }
    }
}
```

19. The method `writeWithCommas` is supposed to print its nonnegative `int` argument with commas properly inserted (every three digits, starting at the right). For example, the integer 27048621 should be printed as 27,048,621. Method `writeWithCommas` does not always work as intended, however. Assuming no integer overflow, which of the following integer arguments will *not* be printed correctly?
    (A) 896
    (B) 251462251
    (C) 365051
    (D) 278278
    (E) 4

20. Which change in the code of the given methods will cause method `writeWithCommas` to work as intended?
    (A) Interchange the lines `System.out.print(n / 100)` and `System.out.print(n % 10)` in method `writeThreeDigits`.
    (B) Interchange the lines `writeThreeDigits(n % 1000)` and `writeWithCommas(n / 1000)` in method `writeWithCommas`.
    (C) Change the test in `writeWithCommas` to `if (n > 1000)`.
    (D) In the method `writeWithCommas`, change the line `writeThreeDigits(n % 1000)` to `writeThreeDigits(n / 1000)`.
    (E) In the method `writeWithCommas`, change the recursive call `writeWithCommas(n / 1000)` to `writeWithCommas(n % 1000)`.

21. Consider the following method:

```
public static void sketch(int x1, int y1, int x2, int y2, int n)
{
    if (n <= 0)
        drawLine(x1, y1, x2, y2);
    else
    {
        int xm = (x1 + x2 + y1 - y2) / 2;
        int ym = (y1 + y2 + x2 - x1) / 2;
        sketch(x1, y1, xm, ym, n - 1);
        sketch(xm, ym, x2, y2, n - 1);
    }
}
```

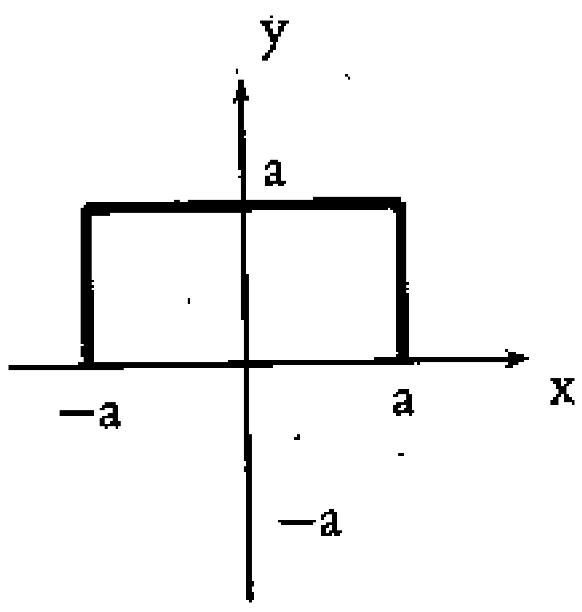


Assume that the screen looks like a Cartesian coordinate system with the origin at the center, and that `drawLine` connects (x1,y1) to (x2,y2). Assume also that x1, y1, x2, and y2 are never too large or too small to cause errors. Which picture best represents the sketch drawn by the method call

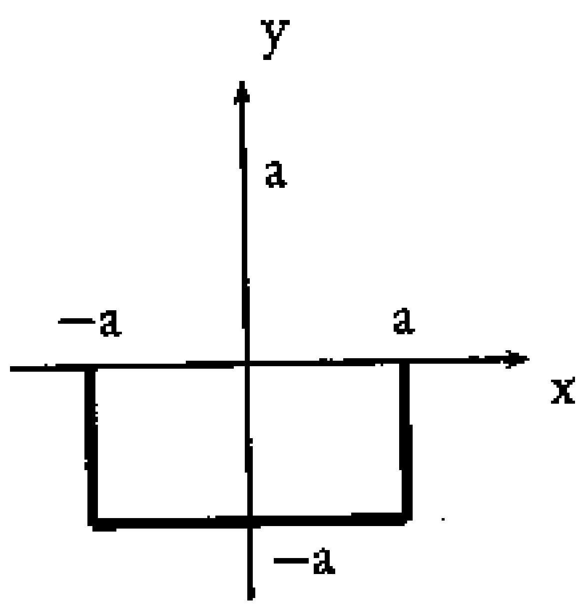
```
sketch(a, 0, -a, 0, 2)
```

where a is a positive integer?

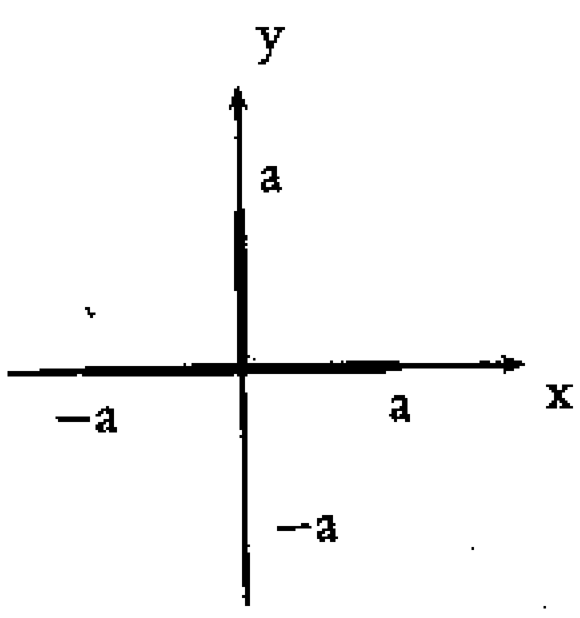
(A)



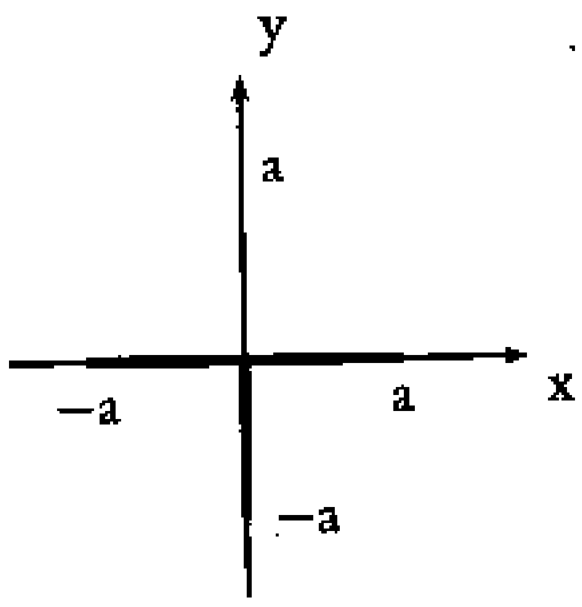
(B)



(C)



(D)



(E)