

Program #9: Recursion

Due Date: April 9th, 2007, at 10:00 p.m. MST

Overview: The construction of recursive solutions requires a different mindset than does the construction of iterative solutions. For most people, the best way to develop that mindset is practice, and lots of it. You've got seven days; we've got seven recursive tasks for you, ranging from the trivial to the challenging.

Assignment: Write two complete, well-documented Java programs, `Prog09a.java` and `Prog09b.java`. `Prog09a.java` will call static recursive methods from a class named `Recursion` in a third file, `Recursion.java`.

Prog09a.java and *Recursion.java*: `Recursion.java` holds a class (`Recursion`) containing a collection of recursive methods, one for each of the following problems, named as indicated. `Prog09a.java`'s `main()` will just test the correct operation of the solution to these problems:

1. Greatest Common Divisor (GCD)

Method signature: `public static int gcd (int x, int y)`

The GCD of two positive integers is the largest integer value that divides both evenly. For example, the GCD of 12 and 15 is 3, $\text{GCD}(7,14) = 7$, and $\text{GCD}(52,65) = 13$. The general case of a recursive algorithm for computing GCDs is easily stated:

$$\text{gcd}(x,y) = \text{gcd}(y,x\%y)$$

Eventually, the remainder will be zero, and the value of y that produced the zero remainder is the GCD. That's the base case of the recursion.

2. Ackermann's Function

Method signature: `public static int ackermann (int m, int n)`

Wilhelm Ackermann created a function of three arguments in 1928. Rozsa Peter simplified that function to one of two arguments, and Raphael Robinson simplified it a bit further. The result is now known as Ackermann's Function, and is famous as a simple example of a recursive function that is not primitive recursive. It also grows very, very quickly for most values of the first argument. Play around with different input values, and you'll see for yourself.

$$A(m,n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

For example: `ackermann(2,4) = ackermann(1,ackermann(2,3)) = ... = 11`.

3. String Reversal

Method signature: `public static String reverse (String str)`

Given a string, return a new string that has the same content as the given string but in reverse order. For example, the string "top" when reversed is "pot".

The Java API provides a `reverse()` method in the `StringBuilder` class. You may **NOT** use it; do all of the 'dirty work' yourself!

(Continued ...)

4. Range Sum

Method signature: `public static double rangeSum (double [] array, int lower, int upper)`

Given an array of `double` and two indices within the array (`lower` and `upper`), return the sum of the elements of the array from index `lower` through index `upper`. For example, consider this array:

0	1	2	3	4	5	6
7	-2	4	0	8	-1	2

Based on this content, `rangeSum(1,4) = 10`, `rangeSum(5,5) = -1`, and `rangeSum(6,5) = 0`.

5. Pascal's Triangle

Method signature: `public static int [] pascalRow (int row)`

This is the top of Pascal's Triangle (rows 0 through 6):

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
```

Each row's values, other than the 1's on the ends, are the sums of the values of the two numbers just above it on the left and right. The 20, for example, is the sum of the two 10's.

Write a recursive method that accepts a row index and returns an array of `int` that contains that row of the triangle, starting at index 0 of the array. For example, if the parameter is 4, the returned array's value at index 0 would be 1, at index 1 would be 4, etc.

Note that we want the `pascalRow()` method to be recursive. It is **NOT** acceptable to move the recursion to a helper method. Also note that, to be able to use the slightly-simpler solution to form the bigger problem's solution, you'll likely want to include a loop in this recursive method.

6. Knight's Tour

Method signature: `public static void knightsTour (int [][] board, int x, int y)`

The Knight's Tour problem is very old (it dates back over 1100 years), but it's easy to describe if you know how the knight moves on a chess board. To get everyone up to speed, the knight has an L-shaped move, either 2 by 1 or 1 by 2. The knight has up to eight possible moves, as illustrated below:

	•		•	
•				•
		N		
•				•
	•		•	

Here's the problem: Given a `rows × columns` rectangular board and a starting square on the board, is there a way to have the knight "tour" the board and visit each square exactly once? There are multiple ways to tour a 3×4 board; here's one of them as an example:

1	4	7	10
12	9	2	5
3	6	11	8

The numbers represent the sequence of visits; 1 is the starting square. This is why the board array is of type `int`: When the knight moves to a new square, the square needs to be marked as having been visited to prevent future visits on the same tour. Placing the 'hop' number in the array is an easy way to accomplish this.

(Continued ...)

Your job is to write a recursive method that finds **all** of the possible tours from a given starting square. This means that when you find a solution, you need to display it and continue. The output format can be trivial; for example, this is fine:

```
1  4  7 10
12 9  2  5
3  6 11  8
```

This problem, like the maze program you did earlier, is an example of backtracking. The difference is that this time you can let recursion manage the stack for you; there's no need for you to (and you should not!) create one of your own.

Prog09b.java:

As part of your second program, you are to write a method that recursively computes the sequence of turns needed to generate a fractal known as the Heighway Dragon (after NASA physicist John Heighway). Then, the program is to display a graphical representation of those turns.

Here's how to generate the list of turns needed to compute the dragon: The simplest form is just a left turn (L). The next simplest form (we'll call it the next iteration) is always the concatenation of three sequences of turns: The previous sequence, a left turn, and the previous sequence, in reverse order, with the Ls replaced by Rs and Rs replaced by Ls. For example, assume that the second iteration is LLR (which it is!). To form the third iteration, we concatenate the previous iteration (LLR), a left turn (L), and the reversed/replaced version of the previous iteration (LRR). The result is the sequence of turns LLRLLRR. Here are the first four iterations:

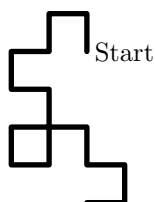
```
Iteration #1: L
Iteration #2: LLR
Iteration #3: LLRLLRR
Iteration #4: LLRLLRLLRLLRR
```

Your tasks for this program are (a) to figure out how to view this recursively, (b) to write a recursive method that accepts the number of the iteration desired and returns the corresponding sequence of left and right turns, and (c) to convert those turns into an image. Use this signature for your recursive turn-generating method:

```
public static String dragonCurve(int iterations)
```

To produce the image, we want you to use the `GTurtle` class from the `acm.graphics` package. "Turtle graphics" is a simplified view of drawing in which an imaginary turtle has a pen and draws with it by following our instructions (such as 'go forward 10 pixels, turn left, go forward 5 pixels'). It's a graphical abstraction that fits this task well, and is very easy to understand, which is why it is often used to introduce children to computer programming.

Initially, place the turtle in the window a quarter of the way across the window and a third of the way down, and have it face the top of the window. (This placement will allow you to display dragons through iteration 14 without the turtle having to move out of the display area.) Tell it to move forward three pixels. Then, for each turn in the turn sequence you recursively generated, turn the turtle and have it move forward three more pixels. Continue until all of the turns have been drawn. The result will be a picture of your dragon curve. Here's a magnified view of what the fourth iteration should look like:



(Continued ...)

Data: There is no mandatory data to use to test the methods of the `Recursion` class. The necessary parameters for each method are given above. Include with `Prog09a.java` an appropriate `main()` method that adequately tests your methods to ensure that they work correctly for all reasonable input values. You'll want your tests to demonstrate that those methods function correctly in a wide variety of situations.

For `Prog09b.java`, prompt the user to enter the number of iterations desired. When testing this program, remember that you can hold off on adding the graphics code to the `run()` method until you know that your recursive turn generator is working correctly.

Output: For each of the recursive methods, the expected return types are given with the method signatures, above. The output of the first program will be determined by how you write `main()`; make sure that your output clearly shows the parameters used and the results produced by each invocation of each method. The second program's output will be primarily graphical, of course.

Turn In: Electronically submit all three of your files (`Prog09a.java`, `Recursion.java`, and `Prog09b.java`) using the turnin program and the usual

`[section leader first name]_S[your section #]_P[program #]_[your last name]_[your first name]` naming convention.

Hints, Reminders, and Other Requirements:

- A sample program that shows `GTurtle` in action is available from the class web page. It's called `Spiral.java`.
- This isn't a particularly difficult assignment ... if you already understand recursion or are one of those lucky people who pick it up quickly. For the rest, this assignment could take quite a bit of time to complete, particularly the Knight's Tour method. Try to remember to ask yourself, "What's simpler than ...?"
- You might want to snoop around for some information on Ackermann's Function before you try to compute it for $m > 3$...
- The Knight's Tour problem is well-known. There are lots of web pages with information about it. A friendly reminder: Programming assignments in this class are to reflect *your* work, not that of another person or that person's web site. The penalty for turning in someone else's work as your own is detailed on the class syllabus.