CSc 127B — Introduction to Computer Science II
Fall 2015 (McCann)

http://www.cs.arizona.edu/classes/cs127b/fall15/

Program #9: A Recursion Half-Dozen

Due Date: November 12th, 2015, at 9:00 p.m. MST

Overview: The construction of recursive solutions requires a different mind-set than does the construction of iterative (a.k.a. looping) solutions. For most people, the best way to develop the recursive mind-set is practice, and lots of it. We've got seven recursive tasks for you, ranging from the trivial to the (reasonably!) challenging.

Assignment: Write a complete, well–documented Java program named `Prog09` and a class named `Recursion` (stored in separate files). `Prog09a` will call static recursive methods from `Recursion`. `Recursion` holds a collection of recursive methods, one for each of the following six problems, named as indicated. `Prog09`'s `main()` is to test the correct operation of the solutions to these problems:

1. **Greatest Common Divisor (GCD)** (Method header: `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, GCD(7,14) = 7, and GCD(52,65) = 13. The general case of a recursive algorithm for computing GCDs is easily stated:

   ```
   gcd(x,y) =  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 header: `public static int ackermann (int m, int n)`)

   Wilhelm Ackermann created a function of three arguments in 1928. Rozsa Peter simplified that to a function of two arguments, and Raphael Robinson simplified it a bit further. The result, known as Ackermann's Function, 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.

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

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

3. **String Reversal** (Method header: `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 "stop" when reversed is "pots".

   The Java API provides a `reverse()` method in the `StringBuilder` class. You may **NOT** use it, or any other short-cuts, in your method; do all of the 'dirty work' yourself!

(Continued ...)

4. **Range Sum**

   (Method header: `public static double rangeSum (double [] aray, 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 header: `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.

   Please note: **The `pascalRow()` method itself must be recursive.** It is **NOT** acceptable to move the recursion out of `pascalRow()` to a 'helper' method.

   HINT: 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 to work on the elements of the current row.

6. **Horse Ride**

   (Method header: `public static void horseRide (int [][] board, int row, int col)`)

   On a chess board, the knight (which usually looks like a horse) has an L–shaped move, either 2 by 1 or 1 by 2. The knight has up to eight possible moves from a starting location, as illustrated below:

	●		●	
●				●
		N		
●				●
	●		●	

   Here's the problem: Given a *rows* × *columns* rectangular board and a starting square (`row`,`col`) on the board, is there a way to have the knight ride around 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:

	0	*1*	*2*	*3*
0	1	4	7	10
1	12	9	2	5
2	3	6	11	8

(Continued ...)

The numbers represent the sequence of visits; 1 is placed in the starting square ((0,0) in this example). 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 ride. Placing the 'move' number in the array is an easy way to accomplish this.

Your job is to write a recursive method that finds **all** of the possible rides from a given starting square. This means that when you find a solution, you need to display it and continue. The output format is shown below; each integer is in a field size of five, right-justified (easily accomplished with `printf()`).

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

You may write helper methods for `horseRide()` to call (such as for printing the array), but, as with `pascalRow()`, **the method `horseRide()` needs to be recursive**.

This problem, like the maze creation program you recently completed, is an example of a programming technique called *backtracking*. The difference is that this time recursion manages the stack for you; there's no reason for you to create one of your own (nor should you!).

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 `Prog09.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.

Output: For each of the recursive methods, the expected return types are given with the method headers, provided above. The output of `Prog09.java` will be determined by how you write its `main()` method; make sure that your output clearly shows the parameters used and the results produced by each invocation of each method.

Turn In: The files to submit are `Prog09.java` and `Recursion.java`. Use the 'turnin' utility on `lectura` to electronically submit them to the `cs127bsXp09` directory at any time before the stated due date and time.

Hints, Reminders, and Other Requirements:

- Because the recursive methods will be static and in the file `Recursion.java`, to invoke them from `Prog09.java` you'll have to prefix the method name with the class name (which is also the file name). For example, to call `gcd()`, you'll have to type `Recursion.gcd()`, just like you use `Math.sqrt()` to call the square root method.
- As mentioned in class, it's hard to "sanity-check" arguments to a recursive method. But, it's easy (if perhaps a bit rude) to throw exceptions! For this assignment, let's be rude: If one of your recursive methods is called with an argument that's invalid for that method, throw an `IllegalArgumentException`.
- 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 everyone else, this assignment could take quite a bit of time to complete, particularly latter methods. Remember to ask yourself, "What's (slightly) simpler than . . . ?"
- Unless you have a lot of free time, you might want to snoop around for some information on the behavior of Ackermann's Function before you try to compute it for $m > 3$...
- Most of these recursive problems are well–known. There are lots of web pages with information about them. Thus, we offer this friendly reminder: Programming assignments in this class are to reflect *your* work, not that of another person. The penalties for turning in someone else's work as your own is detailed on the class syllabus. Remember, we know how to use search engines, too . . .