CSc 227 — Program Design and Development
Spring 2014 (McCann)

http://www.cs.arizona.edu/classes/cs227/spring14/

Program #11: Recursionfest!

Due Date: May 6th, 2014, 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. You've got fourteen days; we've got seven recursive tasks for you, ranging from the trivial to the challenging.

Assignment: Write two complete, well–documented Java programs, `Prog11a.java` and `Prog11b.java`. `Prog11a.java` will call static recursive methods from a class named `Recursion` stored in a third file, `Recursion.java`.

Prog11a.java and *Recursion.java*: `Recursion.java` holds a class (`Recursion`) containing a collection of recursive methods, one for each of the following six problems, named as indicated. `Prog11a.java`'s `main()` will just test the correct operation of the solution 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 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 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.

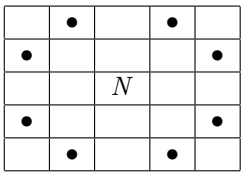
   Please note: **The `pascalRow()` method itself must be recursive.** It is **NOT** acceptable to move the recursion to a 'helper' method.

   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. **Knight's Tour**

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

   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:



   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 "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 'move' 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 creation program you recently completed, is an example of a programming technique called *backtracking*. The difference is that this time you can let recursion manage the stack for you; there's no need for you to create one of your own (nor should you!).

7. **Building a Binary Tree from its Traversals**

(Method header: `public static TreeNode reinflateTree (String preorder, String inorder)`)

Note: You'll probably want to wait to try this method until after we cover binary tree traversals in class.

Given just the pre-order and in-order traversals of a binary tree, we can build the unique tree which generated that pair of traversals. For example, consider this small binary tree and its traversals:

```
    R
   / \          Pre-order:  RWTE
  W   E
 /              In-order:   TWRE
T
```

Because a pre-order traversal always 'visits' the root first, R must be the tree's root node's data value. All data ahead of R in the in-order traversal must be from R's left subtree, and all data after R must be from the right subtree. Each of those subtrees is also a binary tree, meaning that we can do the same thing with each subtree, thanks to the fact that their traversals are found within the full tree's traversals. (In this example, notice that the data values from R's left subtree, T and W, are grouped together in both traversals.) When we have the subtrees built, we can attach them as the left- and right-subtrees of R, and the tree has been 'reinflated.'

Write the `reinflateTree()` method recursively (duh!), assuming that the tree data is of type `char`. You may assume that the given strings are the correct traversals. A traversal with no data will be given to the method as an empty string (that is, a String object of length zero).

To support this method (and your testing of it), you will need to supply one class and two additional recursive methods:

(a) You will need to add a `TreeNode` class to the `Recursion.java` file. Modify your `Node` class from Program #10 and use it here, but be sure to rename it `TreeNode`. You'll have to adjust its instance methods so that there are getters and setters for the left- and right-child references (`TreeNode getLeftChild()`, `setLeftChild(TreeNode)`, etc.), change the node's data from being a `Card` reference to being a `char` value, etc.

(b) Add public static methods `String preorder (TreeNode)` and `String inorder (TreeNode)` to your `Prog11a.java` file. These methods should return String objects containing the traversal data, in the format used with the sample tree above, for the (sub)tree rooted at the given `TreeNode` reference. As we'll be giving the code for the traversals in class (albeit in a form that just displays the traversals to `System.out`), these methods should be easy to code.

(Continued ...)

Prog11b.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 former 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: (a) previous sequence, (b) a left turn, and (c) the previous sequence again, but in reverse order, and 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:

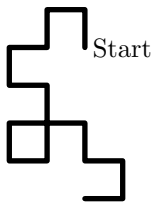
```
Iteration #1:  L
Iteration #2:  LLR
Iteration #3:  LLRLLRR
Iteration #4:  LLRLLRRLLLRRLRR
```

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 header for your recursive turn-generating method:

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

To produce the image, we want you to use the `Turtle.java` file, available from the class web page. "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 units, turn left, go forward 5 units'). It's a graphical abstraction that fits this task well. It is also very easy to understand, which is why it is often used to introduce children to computational thinking.

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 several iterations without the turtle moving out of the display area.) Tell it to move forward a short distance. Then, for each turn in the turn sequence you recursively generated, turn the turtle and have it move forward the same distance. 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:



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 `Prog11a.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 `Prog11b.java`, get the number of iterations from the command line. If no integer value is provided on the command line, have the program prompt the user to enter the number of iterations desired. (That is, do not stop the program when the iterations aren't given on the command line.) When testing this program, remember that you can hold off on adding the graphics commands to the program until you are confident that your recursive dragon curve generator is working correctly.

(Continued ...)

Output: For each of the recursive methods, the expected return types are given with the method headers, which are given 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: (`Prog11a.java`, `Recursion.java`, and `Prog11b.java`) Use the 'turnin' utility on `lectura` to electronically submit your `Prog11a.java`, `Recursion.java`, and `Prog11b.java` files to the `cs227p11` 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 `Prog11a.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()`.
- As mentioned in lecture, 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.
- A sample program that shows `Turtle.java` 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 everyone else, this assignment could take quite a bit of time to complete, particularly latter methods of `Prog11a`. 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 ...