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.

Assignment: Write a complete, well–documented Java program named Prog10 and a class named Recursion (stored in separate .java files). Recursion holds a collection of static recursive methods, one for each of the problems detailed below. Prog10 will call the recursive methods in Recursion for the purpose of testing them. (Yes, you need to write your own tests and test cases for these recursive methods.)

Reminder: GCD, Range Sum, and Reversing Alternate Characters were on the preview version of this assignment. You need to submit those methods as part of this assignment, whether or not you completed them as practice for Midterm Exam #2.

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 (that is, \(\text{gcd}(12,15) = 3\)), \(\text{gcd}(7,14) = 7\), and \(\text{gcd}(52,65) = 13\). The general case of a recursive algorithm for computing GCDs is easily stated:

\[
gcd(x,y) = gcd(y,x\mod y)
\]

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 quickly for most values of the first argument (e.g., values of \(m > 3\) are generally not useful for testing). 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: \(\text{ackermann}(2,4) = \text{ackermann}(1,\text{ackermann}(2,3)) = \ldots = 11\).

3. Range Sum

(Method header: 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:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
7 & -2 & 4 & 0 & 8 & -1 & 2 \\
\end{array}
\]

Based on this content, \(\text{rangeSum}(1,4) = 10\), \(\text{rangeSum}(5,5) = -1\), and \(\text{rangeSum}(6,5) = 0\).

(Continued ...)

(Continued ...)
4. Reversing Alternate Characters

(Method header: public static String backSkip (String message))

The backSkip() method uses recursion to construct a String object containing every-other character of the given string, but in reverse order. Specifically, the returned string has the original’s last character, followed by the 3rd-to-last character, etc.

For example, if the call to backSkip() is:

```java
String aphorism = "Anything you do in life will be insignificant, but" + 
    " it is very important that you do it. Mahatma Gandhi";
String result = Recursion.backSkip(aphorism);
```

The content of result should be:

```
ida maa t duyth ntom rvs itb,ncfnin bli fln duygitn
```

5. Reversed Strings of a Reversed Linked List (RSOARLL)

(Method Header: public static String rsoarll(LLNode<String> head))

Given a reference to the first node of a linked list of String objects, write a recursive method that returns a String object containing the list’s strings in reverse order, with each string also in reverse order, each separated by a single space. For example, given this linked list:

```
start
   ___
  /   
"stressed" "diaper" "palindromes"
```

rsoarll(start) would return the string “semordnilap repaid desserts”.¹

To write your method, add the LLNode<E> class from Section 9 (available from the class web page) to your Recursion.java file. (Section 10’s TestSequence.java file also included the LLNode<E> class.)

The rsoarll()’s recursion is to handle recursing on the linked list of nodes. To build the reversed strings, rsoarll() will need to call another recursive method that you also need to write. We aren’t supplying the header for this one; we’re leaving it for you to design. NOTE: The Java API provides a reverse() method in the StringBuilder class. You may NOT use it in your reversed-string-building method!

6. 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
```

¹Semordnilap was suggested by famous puzzle creator Martin Gardner as the term for a word that, when reversed, spells a different word. A palindrome is a word or phrase that is spelled the same forwards as it is backwards.
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 the recursive method pascalRow() that accepts a row index and returns a reference to an array of int that contains the values of 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.

7. Equine Adventure

(Method header: public static void equineAdventure (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 (the •s) from a starting location (the ‘N’), as illustrated below:

Here’s the question: Given a rows × columns rectangular board and a starting square (row,col) on the board, is there a way to have the knight travel around the board and visit each square exactly once? Believe it or not, there are multiple ways to tour a 3 × 4 board; here’s one of them, that starts in the upper-left corner, as an example:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 visit numbers 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, follow it with a blank line, and continue. The output format is shown below; each integer is in a field size of five, right-justified (easily accomplished with printf()).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>9</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

You may write helper methods for equineAdventure() to call (such as for printing the array), but, as with pascalRow(), the method equineAdventure() needs to be recursive. Should there be no complete tours possible for the given board and/or starting position, print “No tours found.”

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 a stack of your own (nor should you!).

(Continued ...)

3
Data: We are not supplying any test cases for this assignment. The necessary parameters for each method are given above. Include with Prog10 an appropriate main() method that adequately tests your methods to ensure that they work correctly for all reasonable input values. You’ll want to choose your tests to demonstrate that those methods function correctly in a wide variety of situations.

Note that it is perfectly acceptable for Prog10 to be written with a testing method for each of the recursive methods, and to have main() call them in turn. That is, you don’t need to write a massive main() method.

Output: For each of the recursive methods, the expected return types are given with the method headers, provided above. The output of the execution of Prog10.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: For this assignment, the files to submit via `turnin` are Prog10.java and Recursion.java. Be sure to submit them all to the cs127b002sXp10 directory at any time before the stated due date and time. Of course, you can turn them in late if you still have late days to use, or don’t mind losing 20% per day if your late days are exhausted.

Hints, Reminders, and Other Requirements:

- Because the recursive methods will be static and in the file Recursion.java, to invoke them from Prog10.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 as you use Math.sqrt(...) to call the static square root method.

- As mentioned in class, it’s difficult 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.

- Remember to ask yourself: (1) “What’s ‘slightly’ simpler than . . .?”, (2) “How can I use the solution to that ‘slightly’ simpler problem to solve the original problem?”, and (3) “What’s the most trivial case of this problem?” The answers to those questions will help you begin to identify the general and base cases.

- Some 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 are detailed on the class syllabus.