PA3 Recursion

Due: Wednesday 2/9 by 11:30PM
Submission: Submit Recursion.java and RecursionTestClass.java to Gradescope. Do not submit the entire project.

Overview

The goal of this assignment is to practice recursion and testing. In a typical PA, you write one small recursive part of a larger assignment. I did not want you to waste your time on things you have already done in both PA1/PA2 (file reading, formatting output, etc...). So instead this assignment is all recursive functions.

Assignment

In this assignment, you will implement different recursive methods. Open the starter code to see the different methods. They are all commented describing the purpose of each function. **Note, the problems are not arranged in order of difficulty.** You may find the first function the hardest or easiest, who knows. The functions are reproduced here:

```
/**
 * Write a recursive function that finds the index of s2 in s1. Do not use any
 * string functions except for .length(), .equals(), and .substring(). Do not use
 * any loops, or any data structures.
 * @param s1
 * @param s2
 * @return Returns the index of the first time that
 *  s2 appears in s1 or -1 if s2 is not contained
 *  in s1.
 */
public static int indexOf(String s1, String s2) {}

/**
 * Write a recursive function that removes the first k even numbers
 * from the stack. If there are less than k even elements in the stack,
 * just remove all even elements. Do not use any loops or data structures
 * other than the stack passed in as a parameter.
 * @param stack
 * @param k
 * @return Returns the number of elements removed from the stack.
```

```
 */
public static int removeEvenNumbers(Stack<Integer> stack, int k) {}

/**
 * Write a recursive function that accepts an integer and
 * returns a new number containing only the even digits, in the same
 * order. If there are no even digits, return 0. Your function should
 * work for positive or negative numbers. You are NOT allowed
 * to use any data structures. You are not allowed to use Strings to
 * solve this problem either.
 * @param n
 * @return The input with only the even digits remaining in the same
 * order.
 */
public static int evenDigits(int n) {}

/**
 * Write a recursive function that evaluates a Queue<Character> as a mathematical
 * expression. This queue can have any of the following characters:
 * { '(', ')', '+', '*'} or any single digit number. Evaluate this expression and
 * return the result. For example, for the expression:
 * "(((1+2)*(3+1))+(1*(2+2)))", each of these characters would be in the
 * q. As you recursively evaluate characters from the expression, you will
 * remove the characters from the q. After evaluating the above expression,
 * you should return 16. You are guaranteed that there are NO two digit numbers,
 * and that the expression is well formed (parenthesis match, etc...). Do not use any
 * loops. Do not use any data structures besides the q passed in as a parameter.
 * @param q
 * @return The result of the mathematical expression.
 */
public static int evaluate(Queue<Character> q) {}

/**
 * Write a recursive function that accepts a stack of integers and
 * replaces each int with two copies of that integer. For example,
 * calling repeatStack and passing in a stack of { 1, 2, 3} would change
 * the stack to hold { 1, 1, 2, 2, 3, 3}. Do not use any loops. Do not use
 * any data structures other than the stack passed in as a parameter.
 * @param stack
 */
public static void repeatStack(Stack<Integer> stack) {}

/**
 * Write a recursive function that accepts a Queue<Integer>. It
 * should change every int in this queue to be double its original
 * value. You may NOT use loops or any other data structures besides
```

```
 * the queue passed in as a parameter. You may use a helper function.
 * @param q
 */
public static void doubleElements(Queue<Integer> q) {}
```

Sample Outputs

Below is some sample code that I wrote inside of RecursionTestClass.java. Below each print statement, I have written a comment showing what should be printed.

```
System.out.println(Recursion.indexOf("Hello", "lo"));
// 3

Stack<Integer> stack = new Stack<Integer>();
stack.push(2); stack.push(3); stack.push(4); stack.push(55);
stack.push(6); stack.push(17); stack.push(8);
System.out.println(Recursion.removeEvenNumbers(stack, 2));
// 2
System.out.println(stack);
// [2, 3, 4, 55, 17]

System.out.println(Recursion.evenDigits(-1364035));
// 640

String expr = "(((1+2)*(3+1))+(1*(2+2)))";
Queue<Character> q = new LinkedList<Character>();
for (char ch: expr.toCharArray()) {
    q.add(ch);
}
System.out.println(Recursion.evaluate(q));
// 16

System.out.println(stack);
// [2, 3, 4, 55, 17]
Recursion.repeatStack(stack);
System.out.println(stack);
// [2, 2, 3, 3, 4, 4, 55, 55, 17, 17]

Queue<Integer> q2 = new LinkedList<Integer>();
q2.add(34); q2.add(15); q2.add(0);
Recursion.doubleElements(q2);
System.out.println(q2);
[68, 30, 0]
```

Helper Functions

In this class we always strive to solve problems in the simplest way possible. As such, if you are using helper functions when they aren't required, you may be deducted points. Though if that is the only way you can think to solve it, then still submit that solution since you will gain some points for it. There is no penalty for using a helper function for doubleElements.

Testing

You are responsible for testing your code (as it should be). The testing code that you will write is in RecursionTestClass.java. We have given you one example of the format of a test. You can also refer to any of the drills, we are using the same style of testing as is done for the drills.

Remember you need to test the normal case, and the edge cases. What are all the weird (but valid) inputs that you might receive? What sorts of inputs might cause issues with your code?

For this assignment, we recommend you use test-driven development. Write your tests first, before your code! This requires an understanding of the problem, the valid inputs, and expected outputs. Understanding this will make coding the methods a whole lot easier. Let us know what you thought about this approach compared to what (I assume) you have done thus far up to this point. That is, write all your code and then figure out how the heck to test it all.

Hints

- Refer back to the recursive examples done in class.
- Start early!
- There are 6 problems. You have roughly one week to do the assignment. Just like we learn about decomposing a large problem into smaller tasks, why don't you set yourself a goal of doing roughly one of the problems every day? That way you are not stuck with doing a whole bunch of tough recursive problems the night before it is due.
- Remember that often recursive functions are very little code, but it is difficult to figure out what that code is. Spend a lot of time designing an algorithm before jumping into the code. Use a pen and paper!
- The key insight you need to find in these recursive problems is how are they self-similar? Look for patterns. Look for the simplest case, and then try to figure out how to do a tiny bit of work to make a complicated case just a little bit simpler.
- For the evaluate function, you might find

  `Character.isDigit(char ch)` and `Character.getNumericValue(char ch)`

  helpful.

- For the doubleElements function, it might be useful to have another parameter. This might mean creating a helper function.
- Take the recursive leap of faith. Trust your function to do what it should.
- The expressions that are passed into the evaluate function have to follow a strict pattern. For instance, ((4)+(3)) is **not** a valid expression since there are parenthesis around just a single number. (1 + 3 + 5) * 3) is also invalid since there are spaces and/or there are three values inside one set of parentheses. Lastly you are guaranteed that every expression must be in parenthesis. 1+5 is **not** valid, (1+5) is valid. To put it as simply as possible, expressions must be of the form (expr op expr) without spaces, but I needed to provide spaces to make it clear. expr = expression and op = operation.
- Students often struggle most with evaluate. Remember that recursive functions are hard to solve, but once solved can be short and elegant. If your function is getting very long and complex, rethink your solution. Everyone will have different solutions but as an example, my solution is 13 lines long including the method header and curly braces.

Grading Criteria

Unless otherwise specified, you are not allowed to use any data structures or loops in solving these problems.

For this PA, we have only provided one testcase so that you can see the format of how one is written. We will be testing your code on a lot of different testcases, make sure you do the same.

Since this PA is more like a drill than a traditional programming assignment, it will be graded slightly differently. There will be no deductions pertaining to a clean main() method, as well as other traits of a larger program. Instead, we will be grading your testing coverage. So your grade will consist of functionality of your code, style/code clarity, and how well your tests cover different possible scenarios. Of course we still care about many pieces of code clarity, like the below:

- Should use only static methods.
- Each static method should be short and concise. A method should only perform one task. If you have a method that is performing multiple tasks, break it into smaller methods. We think a good rule of thumb for method length is 30 lines.
- Make things as simple as possible. This means avoiding nested loops, nested conditionals, and too many levels of user-defined methods calling other user-defined method (chaining) whenever possible.
- You should be able to read, understand, and explain your own code weeks after you wrote it.
- Use meaningful variable names.

- Remember that indentation and spacing are great tools to make your code more readable. Eclipse should automatically indent the proper levels for you. As long as you stick with those defaults you should be fine.

Note: you do not need to have a file header comment for either file. You **do need to comment your testcases though.** These comments can be brief, but they should describe what that specific test is testing.

Write your own code. We will be using a tool that finds overly similar code. Do not look at other students' code. Do not let other students look at your code or talk in detail about how to solve this programming project. Do not use online resources that have solved the same or similar problems. It is okay to look up, "How do I do X in Java", where X is indexing into an array, splitting a string, or something like that. It is **not** okay to look up, "How do I solve this programming assignment from CSc210" and copy someone else's hard work in coming up with a working algorithm.