Indiana University
CSCI-C212: Introduction to Software Systems (F25)
Joshua Crotts

Problem Set 3
September 10, 2025

Standard and Tail Recursion

Important Dates:

- Assigned: September 10, 2025
- Deadline: September 17, 2025 at 11:59 PM EST

Objectives:

- Students learn to design more complex methods.
- Students understand and describe the differences between standard recursion and tail recursion.
- Students design methods that call `private` helper methods to solve a problem.

What To Do:

For each of the following problems, create a class named `ProblemX`, where `X` is the problem number. E.g., the class for problem 1 should be `Problem1.java`. Write (JUnit) tests for each method that you design in corresponding test files named `ProblemXTest`, where `X` is the problem number. Additionally, write Javadoc comments explaining the purpose of the method, its parameters, and return value. **Do not round your solutions!**

What You Cannot Use:

You cannot use any content beyond Chapter 2.2. This includes loops, arrays, regular expressions, data structures, and so forth. Please contact a staff member if you are unsure about something you should or should not use.

Any use of anything in the above-listed forbidden categories will result in a **zero** (0) on the problem set.

Problem 1:

Design the `isPalindromeTR` tail recursive method, which receives a string and determines if it is a palindrome. Recall that a palindrome is a string that is the same backwards as it is forwards. E.g., "racecar." **Do not** use a (character) array, `StringBuilder`, `StringBuffer`, or similar, to solve this problem. It *must* be naturally recursive.

Problem 2:

This problem has two parts.

(a) The *hyperfactorial* of a number, namely $H(n)$, is the value of $1^1 \cdot 2^2 \cdot \ldots \cdot n^n$. As you might imagine, the resulting numbers from a hyperfactorial are outrageously large. Therefore we will make use of the `long` datatype rather than `int` for this problem. Design the standard recursive `hyperfactorial` method, which receives a long integer n and returns its hyperfactorial.

(b) Then, design the `hyperfactorialTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

Problem 3:

This problem has two parts.

(a) The *subfactorial* of a number, namely $!n$, is the number of permutations of n objects such that no object appears in its natural spot. For example, take the collection of objects $\{a, b, c\}$. There are 6 possible permutations (because we choose arrangements for three items, and $3! = 6$): $\{a, b, c\}, \{a, c, b\}, \{b, c, a\}, \{c, b, a\}, \{c, a, b\}, \{b, a, c\}$, but only two of these are *derangements*: $\{b, c, a\}$ and $\{c, a, b\}$, because no element is in the same spot as the original collection. Therefore, we say that $!3 = 2$. We can describe subfactorial as a recursive formula:

$$!0 = 1$$
$$!1 = 0$$
$$!n = (n - 1) \cdot (!(n - 1) + !(n - 2))$$

Design the standard recursive `subfactorial` method, which receives an long integer n and returns its subfactorial. Because the resulting subfactorial values can grow insanely large, we will use the `long` datatype instead of `int`.

(b) Then, design the `subfactorialTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

Problem 4:

This problem has two parts.

(a) Design the standard recursive `collatz` method, which receives a positive integer and returns the Collatz sequence for said integer. This sequence is defined by the following recursive process:

`collatz(1)` = `1`
`collatz(n)` = `collatz(3 * n + 1)` if `n` is odd.
`collatz(n)` = `collatz(n / 2)` if `n` is even.

The sequence generated is the numbers received by the method until the sequence reaches one (note that it is an open research question as to whether this sequence converges to one for every positive integer). So, `collatz(5)` returns the following `String` of comma-separated integers: `"5,16,8,4,2,1"`. **The last number cannot have a comma afterwards**.

(b) Then, design the `collatzTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

Problem 5:

This problem has two parts.

A *parenthesized string* is a string enclosed by parentheses. For example, the string `"(abc)pqr(de)"` contains two parenthesized strings: `"abc"`, and `"de"`.

For the following problems, you may assume that there are no nested parentheses, all parentheses are balanced, and if there is a parenthesized string, it contains at least one character.

(a) Design the standard recursive `String collectParenthesizedStrings(String s)` method, which receives a `String` *s* and returns a `String` of all the parenthesized strings in *s* as one giant string.

(b) Design the `String collectParenthesizedStringsTR(String s)` method, which solves the same problem as `collectParenthesizedStrings` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

Problem 6:

Design the tail recursive `char findLastNonDigitCharTR(String s)` method that returns the last character in the string that is not a digit. You will need to design a helper method. Remember to include the relevant access modifiers!