Indiana University
CSCI-C212: Introduction to Software Systems (S25)
Joshua Crotts, Muazzam Siddiqui

Problem Set 2
February 5, 2025

Standard Recursion, Tail Recursion, Loops

Important Dates:

- Assigned: February 5, 2025
- Deadline: February 19, 2025 at 11:59 PM EST

Objectives:

- Students learn to design more complex methods.
- Students understand and describe the differences between standard recursion, tail recursion, and iteration.
- Students understand the direct correspondence between iteration and tail recursive methods.
- 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!**

This assignment contains fourteen required problems, meaning the maximum possible score on this assignment is 100%.

You must write sufficient tests and adequate documentation.

Problem 1:

This question has two parts.

(a) 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.

(b) Design the `isPalindromeLoop` method that solves the problem using a loop. The same restrictions from the previous problem hold true for this one.

Problem 2:

This question has three 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.

(c) Finally, design the `hyperfactorialLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Problem 3:

This question has three 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.

(c) Finally, design the `subfactorialLoop` method that solves the problem using a loop.

Problem 4:

This question has three 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.

(c) Finally, design the `collatzLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Problem 5:

The C programming language contains the `atoi` "ascii-to-integer" function, which receives a string and, if the string represents some integer, returns the number converted to an integer. Design the `int atoi(String s)` method that, when given a string s, returns its value as an integer if it can be parsed as an integer. An integer may contain a sign as the first character, that being + or −. If the integer does not have a sign, it is assumed to be positive. Ignore all leading zeroes and leading non-digits. Upon reading a sign, the remaining characters must be digits for the number to be a valid integer. Upon finding the first non-zero digit (after the sign), if one exists, begin interpreting the string as a number. At any point thereafter, if a non-digit is encountered, return the number parsed up to that point. You may assume that the bounds of an integer are never exceeded, i.e., all valid integers will be within the bounds of $[-2^{31}, 2^{31} - 1]$. Writing enough tests is *crucial* to correctly solving this exercise! We provide some examples below. **You cannot use methods that trivialize the problem, e.g., `Integer.parseInt`.**

```
atoi("ABCD")           => 0
atoi("42")             => 42
atoi("000042")         => 42
atoi("004200")         => 4200
atoi("ABCD42ABCD")     => 42
atoi("ABCD+42ABCD")    => 42
atoi("ABCD-42ABCD")    => -42
atoi("000-42000")      => -42000
atoi("000-ABCD")       => 0
atoi("-+-+1234")       => 0
atoi("-A1234")         => 0
atoi("000+42ABCD")     => 42
atoi("8080*8080")      => 8080
```

Problem 6:

Wordle is a game created by Josh Wardle, where the objective is to guess a word with a given number of turns, inspired by Mastermind. In this exercise, you will implement a stage of the Wordle game.

Design the `String guessWord(String W, String G)` method that, when given a string W and a "guess" string G, returns a new string with the following properties:

- If $|W| \neq |G|$, return `null`.
- For every index i, if $W_i = G_i$, append W_i to the output string. If $W_i \neq G_i$ but $G_i \in W$, append an asterisk to the output string. Otherwise, output a dash.

Below are some example inputs and outputs.

```
guessWord("PLANS", "TRAP")  => null
guessWord("PLANS", "TRAIN") => "--A-*"
guessWord("PLANS", "PLANE") => "PLAN-"
guessWord("PLANS", "PLANS") => "PLANS"
guessWord("PLANS", "SNLPA") => "*****";
```

Problem 7:

Design the `String substring(String s, int a, int b)` method, which receives a string and two integers a, b, and returns the substring between these indices. If either are out of bounds of the string, return `null`. You **cannot** use the `substring` method(s) provided by the `String` class.

Problem 8:

File names are often compared lexicographically. For example, a file with name `"File12.txt"` is greater than `"File1.txt"` because `'.'` is less than `'2'`. Design the `int compareFiles(String f1, String f2)` method that would fix this ordering to return the more sensible ordering. That is, if a file has a prefix and a suffix, where the only differing piece is the number, then make the file with the lower number return a negative number. Take the following examples as motivation.

```
compareFiles("File12.txt", "File1.txt")  => 1
compareFiles("File10.txt", "File11.txt") => -1
compareFiles("File1.txt", "File12.txt")  => -1
compareFiles("File1.txt", "File1.txt")   => 0
```

Problem 9:

Design the `int strSumNums(String s)` that computes the sum of each *positive integer* (≥ 0) in a string s. See the below test cases for examples. You may assume that each integer in s, should there be any, is in the bounds of a positive `int`, i.e., 0 and $2^{31} - 1$. Hint: use `Character.isDigit` to test whether a character c is a digit, and `Integer.parseInt` to convert a `String` to an `int`.

```
assertEquals(100, strSumNums("hello50how20are30you?"));
assertEquals(10, strSumNums("t1h1i1s1i1s1e1a1s1y1!"));
assertEquals(0, strSumNums("there are no numbers :("));
assertEquals(0, strSumNums("still 0 just 0 zero0!"));
assertEquals(500000, strSumNums("500000"));
```

Problem 10:

The *definite integral* of a function f, defined as $\int_a^b f(x)\,\mathrm{d}x$, produces the area under the curve of f on the interval $[a, b]$. The thing is, though, integrals are defined in terms of *Riemann summations*, which provide estimations on the area under a curve. Riemann sums approximate the area by creating rectangles of a fixed width Δ, as shown in 4 for an arbitrary function f. Left-Riemann, right-Riemann, and midpoint-Riemann approximations define the focal point, i.e., the height, of the rectangle. Notice that, in Figure 4, we use a midpoint-Riemann sum with $\Delta = 0.2$, in which the collective sum of all the rectangle areas is the Riemann approximation. Your job is to use this idea to approximate the area of a circle.

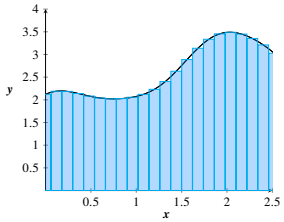

Figure 4: Midpoint-Riemann Approximation of a Function

Design the `double circleArea(double r, double delta)` method, which receives a radius r and a delta Δ. It computes (and returns) a **left**-Riemann approximation of the area of a circle. Hint: if you compute the **left**-Riemann approximation of one quadrant, you can very easily obtain an approximation of the total circle area. We illustrate this hint in Figure 5 where $\Delta = 0.5$ and its radius $r = 2$. Note that the approximated area will vary based on the chosen Riemann approximation.[1] Further note that no calculus knowledge is necessary to solve this exercise.

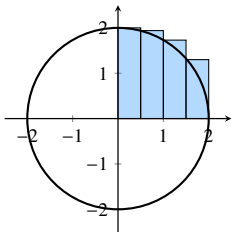

Figure 5: Left-Riemann Approximation of a Function

[1] A left-Riemann sum over-approximates the area, whereas a right-Riemann sum provides an under-approximation. A midpoint approximation uses the average between the left and right approximations.