Lab 05 Computer Science 1027B Summer 2025

Introduction

The purpose of these labs is to provide practice problems for you to get used to the concepts being covered in class. It is very important that you do these labs on your own without relying on available AI tools. Questions on exams are often very similar to exercises presented in these labs, so understanding them and solving these problems gives you a huge advantage for the exams. Make sure that you complete both the code as well as provide written answers to the blue questions within the boxes.

When you have concluded this lab, you should submit the following files to OWL Brightspace:

- GCD.java
- SumOfDigits.java
- call_stack_images.pdf
- palindromeCounter.java
- written_answers.txt (written answers to all questions in the lab)

Topics Covered

- Recursion (just recursion, that's it for this lab)

Exercise 01

Objective:

Write a recursive method inside of a class called GCD to calculate the greatest common divisor (GCD) of two positive integers using Euclid's algorithm (click here if you'd like a refresher). The GCD of two integers is the largest integer that divides both without leaving a remainder.

Instructions:

1. Implement a method named 'gcd' that takes two integer parameters, 'a' and 'b'.
2. Use recursion to find the GCD.
3. The method should handle the base case where the second parameter 'b' is zero.
4. Test your method with various pairs of integers to ensure it works correctly.

Method signature:

```
public static int gcd(int a, int b)
```

Example:

- gcd(48, 18) should return 6
- gcd(56, 98) should return 14
- gcd(101, 103) should return 1

Exercise 02

Let's look at the problem of finding the sum of digits given a positive integer. The problem is defined as follows:

Given a positive integer, write a recursive method, **sumOfDigitsRecursive** in a class '**SumOfDigits** to compute the sum of its digits. For instance, if the input is 1234, then the function should return 10, because 1+2+3+4 = 10.

Hint: The base case must be when there is a single integer < 10 remaining, correct? Then, the recursive step must sum the last digit and the sum of the remaining digits (how could you use the modulo operator and floor division here?)

Next, let's write at an iterative approach to the same problem called **sumOfDigitsIterative** that takes one non-negative integer as input.

Once written, replace the empty methods in the following code with your code, and compile and execute the file.

```
public class SumOfDigits {

    public static int sumOfDigitsRecursive(int n) {
        // your code here
    }

    public static int sumOfDigitsIterative(int n) {
        // your code here
    }

    public static void main(String[] args) {
        int number = 1234567890;

        // measure the time for the recursive method
        long startTime = System.nanoTime();
        int recursiveResult = sumOfDigitsRecursive(number);
        long recursiveTime = System.nanoTime() - startTime;

        // measure the time for the iterative method
        startTime = System.nanoTime();
        int iterativeResult = sumOfDigitsIterative(number);
        long iterativeTime = System.nanoTime() - startTime;

        // print the results
        System.out.println("The sum of digits of " + number + " (recursive) is " + recursiveResult);
        System.out.println("Time taken by recursive method: " + recursiveTime + " nanoseconds");

        System.out.println("The sum of digits of " + number + " (iterative) is " + iterativeResult);
        System.out.println("Time taken by iterative method: " + iterativeTime + " nanoseconds");
    }
}
```

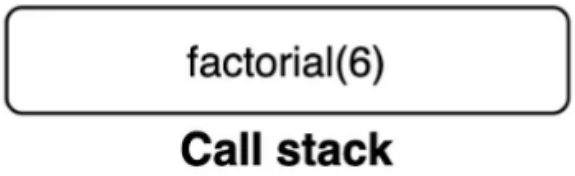
Q 02.1

Which method is faster? Is this surprising? Why do you think this might be?

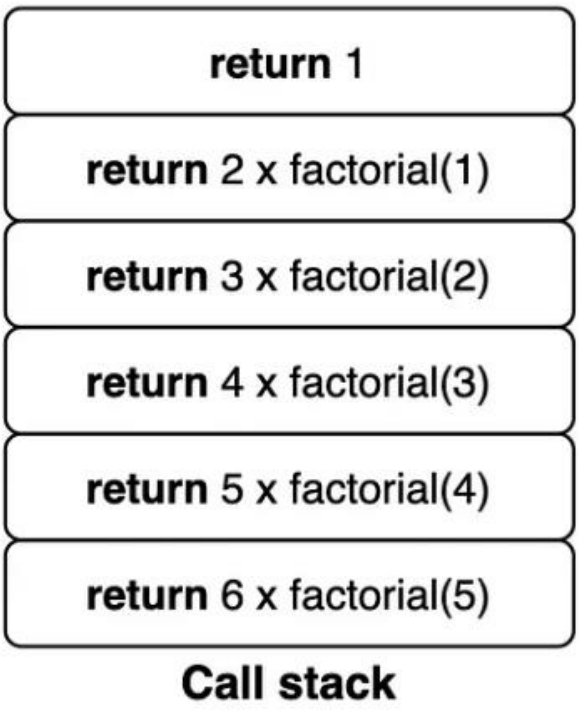
Q 02.2

Why is the base case crucial in a recursive approach? What could happen if the base case is not properly defined or is missing?

Consider the factorial example that we talked about in the lecture. Consider that we wanted to find the value of **factorial(6)**. Once we call our function **factorial(6)**, it is pushed to the call stack, visually represented like this:

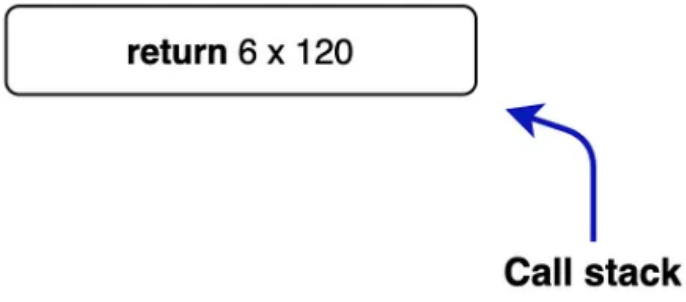


Great! When we hit our base case, factorial(1), our call stack will look like this:



. . .

Finally, when we have made it to the bottom of our call stack after resolving every other method call that was above it in the call stack, we can return our original function call of factorial(6):



Similar to the process described above for the recursive factorial method that we discussed in the lecture videos, draw (preferably using draw.io) the call stack on the first call to sumOfDigitsRecursion, the call stack when we hit our base case, and the call stack when we have made it to the bottom of the call stack after resolving every other method call that was above it. These images should reflect the call to **sumOfDigitsRecursive(1234)**. Once finished, save all images in a PDF called "call_stack_images.pdf".

Exercise 03

Write a recursive method, **isPalindrome**, to check if a given string is a palindrome.

Base case: The string is empty or has only one character. In both cases, the string is a palindrome.

Recursive step: Compare the first and last characters of a string. If they are the same, recursively check the substring that excludes these characters.

Paste the following code into a file named 'PalindromeCounter.java' and put it in the same location as the 'potentialPalindromes.txt' file provided with this lab.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class PalindromeCounter {

    public static boolean isPalindrome(String s) {
        // your code here
    }

    public static void main(String[] args) {
        String fileName = "potentialPalindromes.txt";
        int palindromeCount = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
            String line;
            while ((line = br.readLine()) != null) {
                // check if the word is a palindrome
                if (isPalindrome(line)) {
                    palindromeCount++;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Number of palindromes found: " + palindromeCount);
    }
}
```

Q 03.2

What is printed when the PalindromeCounter.java file is compiled and run?