CSDS 233 Assignment #1

Due September 22, 2024, before 11:59 pm. 100 points

Submission Instructions

- The submissions will be evaluated on completeness, correctness, and clarity.
- Please provide sufficient comments in your source code to help the TAs read it.
- Please generate a single zip file containing all your *.java files needed for this assignment (not .class) and optionally a README.txt file with an explanation about added classes and extra changes you may have done.
- Name your file P1_YourCaseID_YourLastName.zip for your **coding exercises**. Submit your zip file electronically to Canvas.
- Please submit a **PDF** for your answers to the written exercise **separate from the zip file**.

Office Hours for This Assignment

- September 13th from 12:00 pm - 1:00 pm: Zoom:https://cwru.zoom.us/j/98366092294?pwd=GolsUhCbaar14uALvDQPfd0oQOLdLa.1 Passcode 632227
- September 20th from 7:00 pm - 8:00 pm: In-person office hours will be held at Nord 204.

Contact Information

If you have questions outside of the scheduled office hours:

- For written questions, email: eme65@case.edu
- For coding problems, email: yxx914@case.edu

Written Exercise [50 points]

Answer the following questions. When asked to simplify, also be sure to convert logarithmic expressions to base 10. Show your work to be able to receive partial credit.

1. [10 points]

State whether the following assertions in the form "$f(N) = O(g(N))$" are true or false. Justify your answer by computing $\lim_{N \to \infty} \frac{f(N)}{g(N)}$.

Hint: Recall $f(N) = o(g(N))$ *if that limit is 0, and* $f(N) = \Theta(g(N))$ *if that limit is a nonzero constant. Consider using L'Hôpital's Rule:* $\lim_{N \to \infty} \frac{f(N)}{g(N)} = \lim_{N \to \infty} \frac{f'(N)}{g'(N)}$ *(to calculate* $f'(N)$ *or* $g'(N)$*, feel free to use an online derivative calculator).*

a. $2N + 5 = O(N)$

b. $0.01N = O\left(N^{0.99}\right)$

c. $2^N = O\left(2^{N/2}\right)$

d. $ln(N) = O(\sqrt{N})$

e. $Nln^2(N^2) = O(N^2 ln(N))$

2. [15 points]

Consider the following Java code snippets, which show an implementation of a method `func`. For each of them, what is the tightest big O (or equivalently, just big Θ) running time complexity of `func`, in terms of `n`? Simplify your answer.

a.

```
public static void func(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println(i + " " + j);
        }
    }
}
```

b.

```
public static void func(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            System.out.println(i + " " + j);
        }
    }
}
```

Hint: Compare how many of the pairs of "i j" will be printed in `func` *for this part as opposed to part (a). Alternatively, maybe you will find the series identity* $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ *useful?*

c.

```
public static void func(int n) {
    for (int i = 0; i < n; i++) {
        if (i == 0) {
            for (int j = 0; j < n; j++) {
                System.out.println(i + " " + j);
            }
        }
    }
}
```

3. [15 points]

Consider the following Java code snippet, which for non-negative inputs returns an integer between `low` and `high` that, when squared, is equal to `x`, and which returns `-1` if no such integer exists:

```
public static int sqrt(int x, int low, int high) {
    if (low > high || x < 0 || low < 0 || high < 0)
        return -1;
    int p = low + (high - low) / 2;
    if (p * p == x)
        return p;
    else if (p * p > x)
        return sqrt(x, low, p - 1);
    else
        return sqrt(x, p + 1, high);
}
```

a. [3 points]

In an arbitrary recursive call to `sqrt`, let `d1` equal `high - low`, and let `d2` equal what `high - low` will be for the next recursive call, assuming that both `d1` and `d2` are positive. What must $\lim_{d1 \to \infty} \frac{d1}{d2}$ be under these conditions?

Hint: Your answer should simplify to an integer. Think of this ratio as the number of approximately equal-sized partitions into which you're breaking the set of integers between low and high (inclusive) for this arbitrary recursive call, where in the next recursive call you're only considering values in one of these partitions.

b. [4 points]

Under the same assumptions listed in part (a), for the initial call to `sqrt`, we can express the difference `high - low` (the number of integers on which we are initially considering for the search) as approximately equal to a^r, where a is the number of approximately equal-sized partitions into which we break our consideration set (the value you calculated in part (a)) and r is the maximum number of recursive calls to `sqrt` we can go through (depending on `x`) before the desired integer is found or is determined to not exist. Solve the equation $high - low = a^r$ for the approximate value of r, based on the value you calculated for a, in terms of the arbitrary initial inputs `high` and `low`.

c. [3 points]

What is the tightest big O (or equivalently, just big Θ) worst-case running time complexity of `sqrt`, in terms of the input variables `x`, `high`, and `low` (assuming they are all positive)? Simplify your answer.

d. [5 points]

What is the tightest big O (or equivalently, just big Θ) space complexity of extra memory required when running `sqrt` (original implementation), in terms of the input variables `x`, `high`, and `low` (assuming they are all positive)? Simplify your answer.
Hint: Consider memory used on the call stack and space required by any new variables.

4. [10 points]

Consider the following Java code snippets, which show an implementation of a method `func`. For each of them, what is the tightest big O (or equivalently, just big Θ) running time complexity of `func`, in terms of the input variables? Simplify your answer.

a.

```
public static void func(int n) {
    for (int i = n; i > 0; i /= 3)
        System.out.println(i);
}
```

Hint: Consider how much `i` *is changing at each iteration. Might the formula* $n \approx a^r$ *which you used in a form (with* $n = high - low$*) for question 3 be helpful here?*

b.

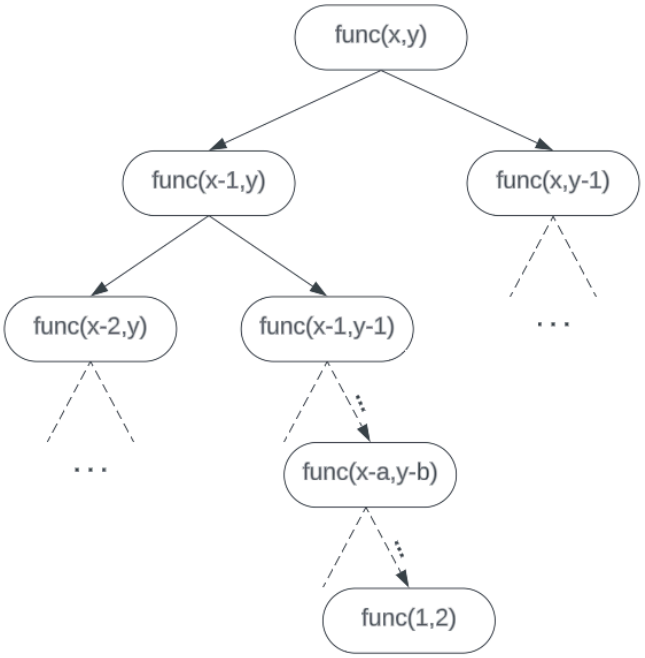
```
public static int func(int x, int y) {
    if (x <= 1 || y <= 1)
        return 1;
    return func(x - 1, y) + func(x, y - 1);
}
```

Hint: The formula $n \approx a^r$ *might still be helpful here, but you should analyze it differently. Rather than skipping some values between* `1` *&* `x` *or* `1` *&* `y`*,* `func` *considers some of these values more than once. Refer to the diagram below to understand what n, a, and r would represent in this example. Which of these can be used for the running time complexity?*

a = "how many recursive function calls directly result from any given function call (excluding base-case calls)?"

r = "how many vertical levels does it take to get from func(x,y) to func(1,2) (this is what happens in the worst case)?"

n = "how many total function calls will there be for arbitrary x and y?"



Programming Exercise [50 points]

Implement the classes and methods below in Java (make sure your code compiles for Java 8). We recommend using Visual Studio code for your IDE (tutorial for Java in Visual Studio Code), though you may use whichever you prefer. Use proper encapsulation practices when coding, and make sure your code is readable.

1. [20 points]

Write the methods required for parts (a) and (b) below in a class called MathFunction.

a. Exponential Method [5 points]

Given below is an iterative method that outputs an int with a value $2x^x$ based on the input variable x. Your task is to rewrite this method using a recursive approach.
Rewrite this method using recursion.

- Write a recursive version of the method named multiplyRecursive(int x).
- The recursive method should not use any loops and should be efficient like the iterative version. A helper method may be needed to do this (hint: make it private, and this should be the one with recursive calls).

```
public static int multiply(int x) {
    int y = 2;
    for (int i = 0; i < x; i++) {
        y *= x;
    }
    return y;
}
```

b. Square Root Method [15 points]

Given below is the same recursive method from question 3 of the written assignment.
Your task is to rewrite this method using an iterative approach.

- Implement an iterative version of the method named sqrtIterative(int x, int low, int high).
- Use a loop instead of recursive calls to sqrt. Make sure your implementation is also efficient like the recursive version.

```
public static int sqrt(int x, int low, int high) {
    if (low > high || x < 0 || low < 0 || high < 0)
        return -1;
    int p = low + (high - low) / 2;
    if (p * p == x)
        return p;
    else if (p * p > x)
        return sqrt(x, low, p - 1);
    else
        return sqrt(x, p + 1, high);
}
```

2: Implementing a Simple Bank Account Class [10 points]

- Task: Create a class named BankAccount that has the following attributes:
    - accountNumber (String)
    - balance (double)
- Include the following methods:
    - A constructor that initializes the account number and balance.
    - deposit(double amount): Method to add money to the account.
    - withdraw(double amount): Method to withdraw money from the account, ensuring that the balance does not become negative.
    - getBalance(): Method to return the current balance.

Instructions: Write a BankAccount class with the specified attributes and methods. Then, write a test class that creates a BankAccount object, makes some deposits and withdrawals, and prints the final balance.

3: Implementing a Student Class [10 points]

- Task: Create a class named Student with the following attributes:
    - name (String)
    - studentId (String)
    - grades (ArrayList<Integer>)
- Include the following methods:
    - A constructor to initialize the name and studentId.
    - addGrade(int grade): Method to add a grade to the student's list of grades.
    - getAverageGrade(): Method to compute and return the average of the grades.

Instructions: Write a Student class with the specified attributes and methods. Create a test class to demonstrate creating a Student object, adding some grades, and calculating the average grade.

4: Book and Library Classes [15 points]

- Task: Create two classes, Book and Library.
    - Book should have the following attributes:
        - title (String)
        - author (String)
        - isbn (String)
    - And the following methods:
        - A constructor to initialize all attributes.
        - getDetails(): Method to return a string with the book's details.
    - Library should have:
        - books (ArrayList<Book>)

- And the following methods:
  - addBook(Book book): Method to add a book to the library.
  - removeBook(String isbn): Method to remove a book from the library using the ISBN.
  - printBooks(): Method to print details of all books in the library.

Instructions: Implement the Book and Library classes. Then, create a test class that adds several books to a library, prints all book details, and removes a book by its ISBN.