CSCI 1933 Lab 3
Working with Arrays and Scanner

Lab Rules

You may work individually or with *one* partner in lab. We suggest that you have a TA evaluate your progress on each milestone before you move onto the next. The labs are designed to be completable by the end of lab. If you are unable to complete all of the milestones by the end of lab, you have **until the last office hours on Monday** to get those checked off by **any** of the course TAs. We suggest you get your milestones checked off as soon as you complete them since Monday office hours tend to become crowded. If you worked with a partner, both of you must be present when getting checked off to receive credit. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Canvas for this lab.

Attendance

Per the syllabus, students with 4 unexcused lab absences over the course of the semester will automatically fail the course. Meaning if you have missed 3 labs without excuses you are OK, but if you miss a fourth lab (without a formal excuse) you would fail the course. The TAs will take attendance during lab; it is expected that students will be actively working on the lab material. *Your physical presence in lab is not sufficient to be marked as present for the purposes of attendance.*

1 Fibonacci Sequence

The Fibonacci sequence is named after the Italian mathematician, Leonardo of Pisa, also known as Fibonacci. To calculate the Fibonacci sequence, you start with the 0^{th} and 1^{st} terms, which are 0 and 1 respectively. **To calculate the next term you sum the previous two terms.** Using the initial numbers $0 + 1 = 1$ the 2^{nd} term in the Fibonacci sequence is 1! Now that you know the pattern, here are the 0th through 8th numbers in the Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21$.

Now let's write a **Fib** class to help us find the n^{th} Fibonacci number. It will have the method `public static int fibonacciRecursive(int n)` which will return the n^{th} Fibonacci number as an int using a recursive approach. Here are some test cases:

Example Test Cases:

- `fibonacciRecursive(3)` will return 2
- `fibonacciRecursive(5)` will return 5
- `fibonacciRecursive(8)` will return 21
- `fibonacciRecursive(10)` will return 55

Once you have your method working, use `Scanner` to prompt for input.

Hint: For a reminder on how to use scanner follow this link to the Scan.java file from lecture. The first scanner example explains how to take ints as input.

Reflection: When you run your method with large values like 80 or 200 what happens? Are there redundant calculations that are slowing down your method? How might you speed up your method by using an array, or using an iterative approach?

Milestone 1:
What values need to be tested? How does your Fibonacci method handle a negative value for `n`?
Show a TA your method and have them input numbers into your `Scanner` prompt to show correctness.

2 Find the Maximum Digit in a Number

In this section, you will find the maximum digit in a given **positive** `int` both recursively and iteratively. Write a **Max** class which will have the following methods:

- `public static int recursiveMaxDigit(int num)` - This will find the maximum digit in num **recursively**
- `public static int iterativeMaxDigit(int num)` - This will find the maximum digit in num **iteratively**

Hint: Use integer division [`/`] and the modulo (remainder) operation [`%`] to separate out the individual digits.

Write a main method to test your methods. We have included some example return values below for you to get started, but please also **write 3 of your own test cases for each method.**

Example Test Cases:

- `recursiveMaxDigit(578)` will return 8
- `recursiveMaxDigit(10)` will return 1
- `iterativeMaxDigit(9999)` will return 9
- `iterativeMaxDigit(13442)` will return 4

Milestone 2:
Show a TA your working test cases and code for `recursiveMaxDigit(int)` and `iterativeMaxDigit(int)`

3 Histogram

Recall that a histogram is a visual representation of a distribution of discrete data. For example, the data set $\{3, 2, 1, 2, 3, 0, 1, 5, 3\}$ over the range $[0, 5]$ will have the following histogram:

```
0:*
1:**
2:**
3:***
4:
5:*
```

Notice that each array element contains the frequency of its index value in the data set. When programming it is important to break your problems down into small pieces. Here is a breakdown for this section.

1. Create a `Histogram` class with a way to store the information in a histogram.
   - Add a constructor `public Histogram(int lowerbound, int upperbound)` which will initialize your histogram by setting the range.
   - Add a method `public boolean add(int i)` - If `i` is between `lowerbound` and `upperbound` inclusive, then add `i` to the histogram and return `true`. Otherwise, return `false`.
   - Add a method `public String toString()` - This will *return a `String`* formatted as in our example above. That means you are not printing anything in this method. Make sure your data points are in order from `lowerbound` to `upperbound`.
   - Create the main method. Inside, implement some tests on an instance of `Histogram` to confirm that everything is working correctly.
2. Next, create a driver class called `HistogramApp` which will utilize `Scanner` to collect input from the user and run a more thorough test of `Histogram`.
   - Start by prompting the user to enter `lowerbound` and `upperbound` for their histogram.
   - Create an instance of `Histogram` that matches the user's desired range.
   - Ask for user input until the program ends. This can be accomplished with a `while` loop. Implement the following commands (i.e. what should be done when the user types in this String):
     - **add** - this command will prompt the user for more numbers for the histogram. This should involve calling your `add` method from `Histogram` and will require you to receive input again using `Scanner`.
     - **print** - this should give the user a view of their current histogram by printing the `Histogram` instance. This command relies on your `toString()` method to work properly (but note that `System.out.println()` will call `toString()` for you).
     - **quit** - calling this command should cause the program to end.

Remember to only accept the commands listed above. If the user types in something else, simply remind them of their available options. An example output for the `HistogramApp` program can be found below. Remember that green text is typed by the user.

```
---Histogram Console---
Options
add - used to add numbers to the histogram
print - prints the histogram to the screen
quit - leaves the program

Enter Range to get started: 10 13

>add
>Enter number(s): 10
>add
>Enter number(s): 9
9 is not in the range
>add
>Enter number(s): 10 12 13 11 13 13 25 10 77
25 is not in the range
77 is not in the range
>print
10:**
11:*
12:*
13:***
>quit
Bye!
```

Hints: In the constructor if `upperbound` is smaller than `lowerbound`, then simply swap the bounds. Also, your data range should not include negative numbers or fractions. In the `toString()` method, the character `\n` adds newlines. You may need to refer back to the Scan.java file, because now you will need to take both ints and Strings as input.

Milestone 3:
Show a TA your `Histogram` and `HistogramApp` classes once they are successfully tested.

4 Honors Section-Fibonacci Memoize

This section is aimed at the Honors section, however everyone is encouraged to do it. As you may have noticed, the recursiveFibonacci method is slow for large inputs. This is because many calculations are repeated, when calculating the 5th Fibonacci number, the 4th and 3rd numbers are calculated separately. However one will notice that in order to calculate the 4th Fibonacci number we must know the 3rd number too, and so on.

Memoization lets us store the Fibonacci numbers that we have already calculated in an array; this greatly speeds up our calculations.
Add a method `public static int fibonacciMemoized(int n)` which will return the n^{th} Fibonacci number using memoization, while maintaining a recursive approach. When a Fibonacci number is requested, the array will be checked first. If the number is present, simply return it. If it is not present, recursively calculate it. Make sure to check the array first before trying to calculate the number.

Hints: Create a single dimension array called table and store the first two Fibonacci numbers in it. Every successive number can be filled in when calculated.

Milestone 4:
For Honors students: Explain how your method works and how it speeds up the calculation. How is performance related to the number of Fibonacci numbers you calculate?