WIT COMP1000 Computer Science I

Prof. Thai

Student:

**Lab9: Fractions**

1. **Fractions** (Fractions.java)

Write a program that works with fractions. You are to implement three methods called subtractFractions(), multiplyFructions() and divideFractions() to perform subtraction, multiplication and division, respectively, on a pair of fractions. You are given two fractions as arguments for each function; each of them is a two-element array (the numerator is at index 0, the denominator is at index 1). The methods are to return a resulting, simplified fraction as a new two-element array (again, with the numerator at index 0, and denominator at index 1).

You have been provided an add fraction method as an example. You must compute the resulting fraction using fraction-based math (working with numerators and denominators) – do **not** convert the fractions to double values (like 1.5), do the math, and convert back to a fraction. You have been provided a method to simplify a fraction using the **gcd** method from a previous lab. Here are the common fraction manipulation techniques:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Once the operation methods are complete, focus your attention on the **main** method. You first need to input the two fractions from the keyboard (numerator then denominator for each operation. You can assume they are all integers) as well as one of the four valid operations (+, **-**, **\***, **/**). Then validate the inputs: make sure a valid operation was input, neither of the denominators are zero, and the numerator of the first fraction is not zero if the operation is division (error messages have been provided for each of these situations). Finally, compute the result of the operation and output the answer. Note that if the denominator of the answer is 1, you should just output the numerator (this includes if the answer is 0). Here are two sample test cases of the program:

Enter the numerator for the first fraction: 1

Enter the denominator for the first fraction: 2

Enter the numerator for the second fraction: -4

Enter the denominator for the second fraction: 8

Enter the operation (+, -, \*, /): +

1/2 + -4/8 = 0

Enter the numerator for the first fraction: 7

Enter the denominator for the first fraction: 8

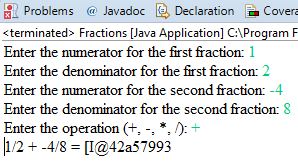
Enter the numerator for the second fraction: 1

Enter the denominator for the second fraction: 3

Enter the operation (+, -, \*, /): -

7/8 - 1/3 = 13/24

**TAKE A SCREENSHOT** of the above sample runs and paste them here. Do not paste your source code in this document.



1. **AboveAverage** (AboveAverage.java)

Write a program that reads a stream of integers from the console and stores them in an array. The array is then analyzed to compute the average of all the values in the array, and finally all the values that are above the average should be printed out to the screen. Specifically, you must write three methods: **main()**, **readIntoArray()**, and **printAboveAverage()**.

**main()** creates a scanner and an array of 100 elements of integer type, and outputs a message to the screen asking for a sequence of numbers. **readIntoArray()** is then called to read values from the scanner and store them in the array. It must be passed two arguments: the scanner and the array objects. You should only store as many integers as the array can handle. Note, however, that there might be fewer than 100 values typed at the console – store whichever is *fewer*. This method must return how many integers, up to the length of the array, were read into the array. The **hasNextInt()** method of the scanner will be useful to determine if there are additional integers to read from the console.

Finally, **printAboveAverage()** should be called to read through the array, compute the average, and then print out all values in the array that are above the average. For each value above the average it should print the index in the array, as well as the value itself. **printAboveAverage()** should take two arguments: the array and the actual number of values in the array. Note that the second argument is not the total number of elements that the array can hold but is instead the number of values that are valid (i.e. populated in the **readIntoArray()** method). For example, the array should be able to hold up to 100 values, but there might have only been 15 values typed at the console.

**TAKE A SCREENSHOT** of your input and output and paste them here. Do not paste your source code in this document.

1. **Median** (Median.java)

Write a program to help students practice median. Write a program to generate five two-digit random numbers. Store the integers in an array. Verify all numbers are unique, no duplications.

Display the numbers and ask the student to find the median of the generated numbers. Note that the median is the middle number of a sorted list. You can use the following statement to sort the array:

java.util.Arrays.*sort*(myArray);

Here are sample test cases:

What is the median of these numbers: 53, 20, 15, 45 and 90: 20

Your answer is not correct. Do you want to try again? (Y/N): Y

What is the median of these numbers: 53, 20, 15, 45 and 90: 90

Your answer is not correct. Do you want to see the answer? (Y/N): N

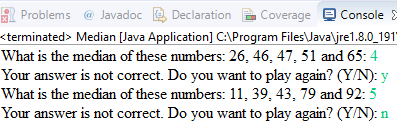
What is the median of these numbers: 53, 20, 15, 45 and 90: 20

Your answer is not correct. Do you want to see the answer? (Y/N): Y

The correct answer is 45. Do you want to play again? (Y/N): Y

What is the median of these numbers: 63, 22, 15, 35 and 83: 35

Your answer is correct. Do you want to play again? (Y/N): N

**TAKE A SCREENSHOT** of your input and output, and paste them here. Do not paste your source code in this document.

Submit your source code and this document to Blackboard for grading.