COMP1006/1406 – Winter 2022

Submit a zipfile called `a1.zip` to Brightspace.

The assignment is out of 40 and is worth 10% of your final grade.
You can receive up to 44/40 on this assignment.

The objective of this assignment is to get you up to speed with basic Java syntax (control flow, branching, variables, methods, basic i/o, etc) with an emphasis on using arrays.

This assignment will mostly be graded for correctness.
Be sure that your code compiles and runs.
Test your code!

In this assignment, you can use the following classes/methods: `Math`, `String`, `StringBuilder`, `Double` (or any primitive wrapper class), any helper class you create yourself, and static helper method you write yourself.
You can use `System.in` and `System.out` for your own testing.

You are **NOT** allowed to use `Arrays`, `Array`, `ArrayList` (or any JCF class), or `System.arraycopy()`.
Essentially, using any other class that solves your problems for you is forbidden.
Using any of these may result in a grade of zero.

The assignment has two parts. The intention is that part one should be completed by Friday, January 28th, and that part two should be completed by Friday, February 4th. For this assignment, there will only be one final due date (Feb 4th) and one submission link in Brightspace.

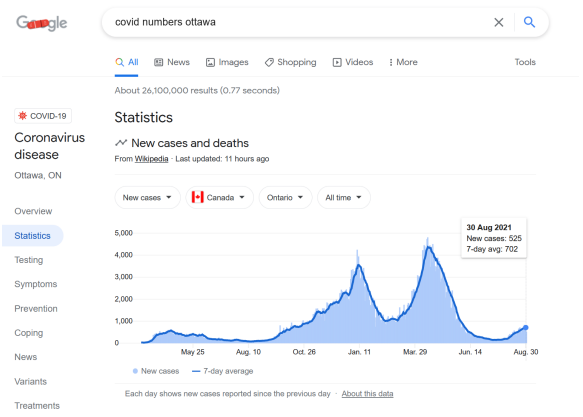
The assignment is due on Friday, February 4th, at 11:59PM. Please note that there is a 48-hour grace period in which submissions will be accepted without penalty. If need be, you can submit this assignment up to Sunday, February 6th, at 11:59pm without any penalty. However, there will not be any office hours and no guarantees that questions will be answered on discord over the weekend. Be sure to start early and submit often. Brightspace will keep your LAST submission.

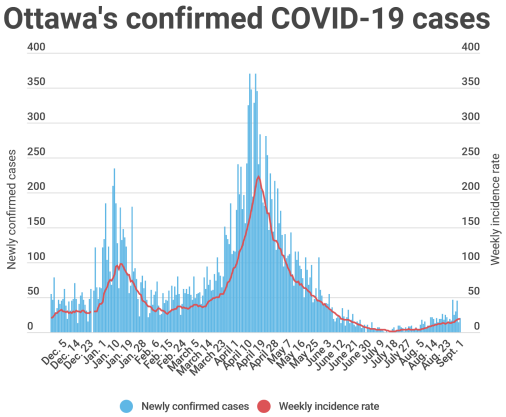
It is expected that ALL java files that you submit will have a completed comment header.
Use the empty headers as provided and fill in the relevant data.
Marks are not *given* for these headers, but marks will be *deducted* if they are missing or empty.

Part One

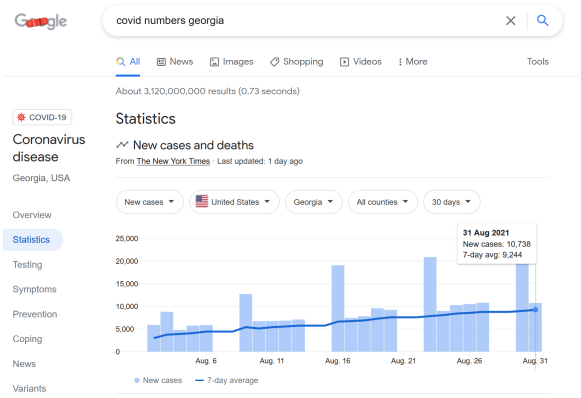
1 Rolling Average [10 marks]

In the past coupe of years, plots like the two shown below have become very common on the Internet. The left is the result of a Google search for "covid numbers ottawa", and the one the right is from cbc.ca.





In these plots, both the raw data and the rolling average of that data (taken over 7 days) is included. The rolling average is present to highlight the overall trends of the data that might not be clear from the raw data. For example, consider the data for one month shown below.



Looking carefully at the dates (they correspond to August 2021), you will see that the new case counts were zero over each weekend of the month (Saturday and Sunday) and that most Mondays had a significantly large count than the rest of the days of the week. The absence of cases on some days and then the jump in numbers after them make sense when you take some time to consider what is happening. The rolling average (using a 7-day average and shown as the continuous line), however, clearly shows a consistent slow increase in numbers over the month. The rolling average quickly communicates the overall trend in the data without having to consider what the gaps and jumps mean.

The rolling average (also called a *moving average* or *running average*) for any given day is the average of that day's count along with the 6 previous days for a 7-day average. (Note that there are other ways of computing a 7-day average in which you use the current day's value along with the values from the 3 previous days and the next 3 days. We will only use the current day and

some previous days in this assignment.) A rolling average does not have to use 7 values for the average in general but it is common when the data corresponds to days of the week.

In the provided `A1Q1.java` file, complete the `rollingAverage()` method. This method will compute the rolling average of some data using a specified width (number of data points used for the average). This method takes an array of floating point numbers (called `data`) and an integer (called `width`) as input. The method returns a new array that is the same size as the input array and will contain the rolling average of the input array data using `width` data points for each value.

```
public static double[] rollingAverage(double[] data, int width)
```

For each position where there is not enough data points (`width`) to compute the average, you will use the `NaN` (Not-A-Number) value. In particular, the first `width`−1 elements of the output array should have this value. Here are some examples:

```
A1Q1.rollingAverage(new double[]{1,2,4}, 1) -> [1.0, 2.0, 4.0]
A1Q1.rollingAverage(new double[]{1,2}, 2) -> [NaN, 1.5]
A1Q1.rollingAverage(new double[]{1,2,4,9}, 3) -> [NaN, NaN,  2.3333333333333335, 5.0]
A1Q1.rollingAverage(new double[]{1,2,4}, 4) -> [NaN, NaN,  NaN]
```

You will need to investigate and use Java's `Double` class to learn more about the value `NaN`.

In future assignments, restrictions on the input arguments passed to methods you create will be provided as preconditions for the methods. For this assignment, they are listed here (as well as in the provided java file). Note that no testing will involve input data that violates these restrictions.

```
data: 0 <= data.length <= Integer.MAX_VALUE/100
size: 1 <= width <= Integer.MAX_VALUE
```

2 Peaks [10 marks]

When analyzing data, such as the output of the rolling average method from the previous problem, it is sometimes useful to identify where the (local) maximum values are located. We'll call these maximal values *peaks* in the data. More formally, if we have a sequence of numbers $d_0, d_1, d_2, \ldots, d_{n-1}$, the peaks are identified as follows:

1. d_0 is a peak if $d_0 > d_1$,
2. d_{n-1} is a peak if $d_{n-2} < d_{n-1}$,
3. for any $1 \leq i \leq n-2$, d_i is a peak if $d_{i-1} < d_i$ and $d_i > d_{i+1}$

In the provided `A1Q2.java` file, you will complete the `peaks()` method which finds all peak values (and their positions) in a given array of numbers.

```
public static double[][] peaks(double[] data, double epsilon)
```

The input parameter `data` stores the data that we are looking for peaks in and `epsilon` is used as a tolerance value for floating point number equality. Because of the *representational* error of floating point numbers, we should never try to check if two floating point numbers are the same. Instead, we consider two floating point numbers to be the same number if they are close enough to each other. In particular, two numbers x and y are considered *equal* if and only if $|x - y| < epsilon$. While not necessary, it is highly recommended that you create and use a static helper function (in the same A1Q2 class) to determine if two floating point numbers are considered equal for a given epsilon. You must not use `equals()`, `compareTo()` or `compare()` from the `Double` class for this.

The method returns an array containing exactly two (2) arrays of doubles. The first array holds all the *peak* values and the second array holds the index positions of those peaks. For example,

```
// 1 peak in the data
double[]   data1  = {1.1, 2.2, 3.1, 4.2, 2.3};
double[][] peaks1 = A1Q2.peaks(data1, 0.0001);
// assert: peaks1 == { {4.2}, {3.0} }

// 2 peaks in the data
double[]   data2  = {1.1, 2.2, 1.1, 2.2, 3.3};
double[][] peaks2 = A1Q2.peaks(data2, 0.00001);
// assert: peaks2 == { {2.2, 3.3}, {1.0, 4.0} }

// NO peaks int the data
double[] data3 = {0.3, 1.1, 2.2, 2.2, 2.2, 1.1, 0.2};
double[][] peaks3 = A1Q2.peaks(data3, 0.00001);
// assert: peaks3 == { {} , {} }
```

Note that we are **not** considering what might be described as *plateaus* in the data. A plateau occurs when a local maximum value is repeated one or more times consecutively (the dataset `data3` from above has a plateau with value 2.2 that spans index values 2-4, inclusive).

[OPTIONAL] For up to 4 bonus marks, complete the `plateaus()` method in the `A1Q2` class. You'll need to comment out the `throw` line in the method if you wish to complete this. This method identifies all plateaus in a given data set. The output is an array with THREE arrays: the plateau values, the starting index of a plateau and the length of a plateau, respectively. Note that a *peak* is just a *plateau* with length 1 and so all peaks are also included. For example,

```
// 1 plateau in the data
double[] data3 = {0.3, 1.1, 2.2, 2.2, 2.2, 1.1, 0.2};
double[][] peaks3 = A1Q2.plateaus(data3, 0.00001);
// assert: peaks3 == { {2.2} , {2.0}, {3.0} }

// 1 peak and 1 plateau in the data
double[] data4 = {3.3, 1.1, 2.2, 2.2, 2.2, 1.1, 0.2};
double[][] peaks4 = A1Q2.plateaus(data4, 0.00001);
// assert: peaks4 == { {3.3, 2.2} , {0.0, 2.0}, {1.0, 3.0} }
```

Note: finding the *peaks* (which include the *plateaus*) in scientific data is very common. For example, analyzing data from various spectrometers in chemistry and physics labs often involves identifying the peaks in the data first. Tools like Matlab and Octave have built-in functions to find peaks just like your `peaks()`/`plateaus()` methods. In practise, it can be more complicated than as described here because the uncertainty in the measured data must also be considered (among other things).

Part Two

3 Formatting Data [20 marks]

For this part of the assignment, you will make a program that will ask a user for input (some data) and will then display a text-based plot of that data similar to the plots in Question 1 (Rolling Average), except the plot will be rotated. The plot will show the data as well as the rolling average. Each piece of data will consist of a label and a numerical value.

The program must begin by first asking the user how many data points will be entered, and the width to use when computing the rolling average. It then allows the user to enter the data. Once all the data is entered, a plot of the data will be displayed to the screen.

Here is an example run of the program. Note that the red text shows the user input in the program and the blue text denotes what the program displays. Your program will will NOT have red/blue text.

```
enter number of data points : 6
enter width for rolling average : 3
enter data one line at a time as  label , value
January 1, 30
2, 20
3,40
    04,  0
5   , 20
6th, 20

January 1 |-----------------------------              30.000
        2 |--------------------                       20.000
        3 |-----------------------------*----------  40.000
       04 |                    *                       0.000
        5 |--------------------*                      20.000
      6th |------------*-------                       20.000
```

Here are some details that you must follow for this problem.

- The labels must be all right-justified (as shown above). The largest label (length) will have no leading spaces before it when displayed.
- There is one blank space between the labels and a pipe character (vertical bar, `|`).
- The LARGEST value in the data set will have 40 dashes (`-`) and all other values will be scaled accordingly. When scaling, you will need to round to the nearest number of dashes to show. It is fine to have NO dashes for a value of 0 or when scaled is less than 1/2.
- The rolling average value will be denoted by the asterisks character `*`. Note that the first two values in the example above do not show the rolling average because the width is 3. You'll have to scale this value and round to find the correct place to display this. If there is already a dash (from the value) then you will display the asterisk instead of the dash.
- the data values are also displayed using 3 decimal places to the right of the plots. The numbers must be lined up so that the decimal place is in the same position. There must be FIVE spaces separating the plot area and the numerical values. The plot area is ALWAYS 40 characters wide.
- The label and value of a data point are entered in a single line and they are separated by a comma. No other commas are allowed to be entered (for the label). It does not matter how much whitespace is around the comma. See the example above for different examples of this. All whitespace at the front and end of the label must be removed.

- When showing the 3 decimal places for the values, they do NOT have to be rounded. Just use string formatting to achieve this.

Create a class called `PartTwo` that has your program in it. Your class must have a `main()` method (which makes it a program) and a static method that generates a `String` of the entire plot. The method must look like

```
public static String plot(String[] labels,
                          double[] values,
                          double[] rollingAves)
```

Here, `labels` are all the labels of the data points, `values` are the numerical values, and `rollingAves` are the rolling averages. All the arrays must have the same length. Some of the first values in `rollingAves` might be `NaN` and these are NOT displayed in the plot. The output of the method is single String that when printed will display the plot (like in the example above). Note that the String in the example above starts with `"J"` and ends with `"0"`. Each line, except the LAST in the plot must have a newline character at the end of it (`"\n"`).

You can use your `A1Q1.rollingAverage()` method to compute the rolling average in your program. Same sample arrays (labels, values, rollingAves) will be provided for you to test your `plot()` method.

We will test your program and your `plot()` method separately. So, when we run your program, if your `A1Q1.rollingAverage()` is not computing the right values that is fine. We are testing the overall behaviour of the program. When we test the method directly, will will expect exact string outputs for a given labels/values/rollingAves arrays.

Submission Recap

A complete assignment will consist of three `.java` files (`A1Q1.java`, `A1Q2.java`, and `PartTwo.java`).

`A1Q1.java`: 10 marks for `rollingAverage`
`A1Q2.java`: 10 marks for `peaks`, [bonus `plateaus`: 4 marks]
`PartTwo.java`: 20 marks

Start early. Submit early. Submit often. Brightspace will only keep your **last** submission.