COMP1006/1406 – Fall 2021

Submit all of your files to Brightspace. Only submit your `.java` and `.txt` files. Do not submit any `.class` files. Do not submit a zip file.

The assignment is out of 100 and is worth 10% of your final grade.

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.

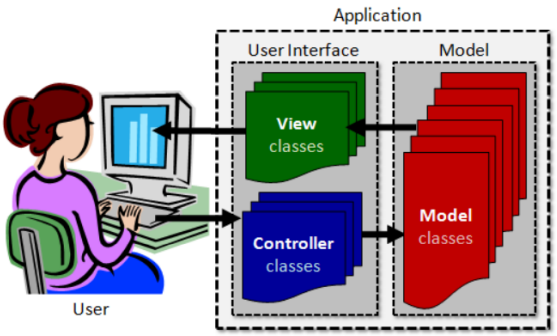
This assignment will mostly be graded for correctness. If one of your Java files does not compile, you will receive zero marks for that question. If your file does compile, then your grade will be determined by the test cases and how your code performs with them.

In this assinment, you can use the following classes/methods: `Math`, `String`, `StringBuilder`, `Double` (or any primitive wrapper class) any class you create 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), `System.arraycopy()`, or any other class that solves your problems for you. Using any of these will result in a grade of zero.

Note that Brightspace only keeps your **last** submission.
Since you are submitting multiple files, be sure to submit **all** files you have ready each time you submit.

It is good programming practise to break your code down into smaller sub-programs that each perform a single task. The actual program will then execute the different parts when needed. In COMP 1005/1405, we tried to do this by writing functions. In COMP 1006/1406, we will use classes. Each class should do one thing (and hopefully, do it well).

A common approach when developing interactive software is the model-view-controller (MVC) architecture. The `model` contains the logic and data for the application. This is sometimes called the *business logic* of the application. The view and controller combined provide the *user interface* for the application. The `view` displays (some of) the data from the model in some way. The `controller` takes input from the user and sends to the model to be used accordingly. Each component might be made up of several or many different classes.

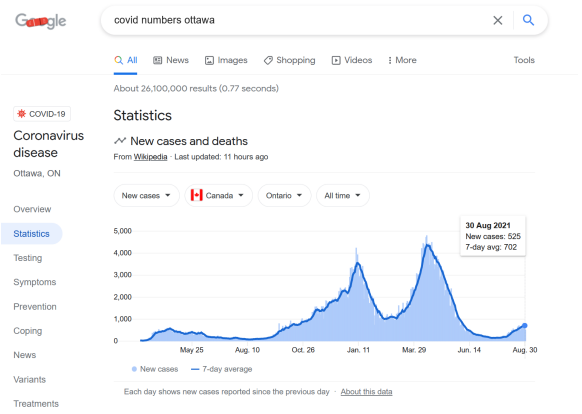


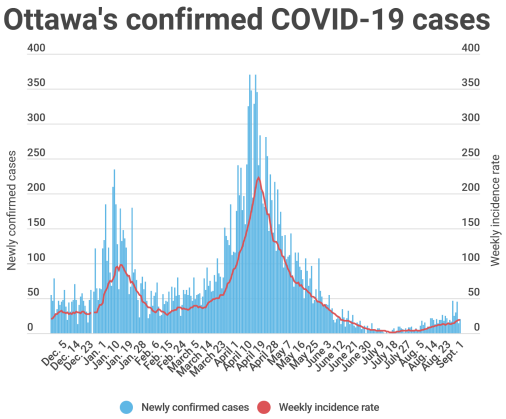
The textbook covers the MCV architecture in Chapter 6. We will discuss this further in the semester and not worry about applications with a GUI (graphical user interface) for now. You are not expected to read Chapter 6 for this assignment.

For now, it is important to realize that the majority of time when working on a software project, you are working on one single component of that project. For example, you might be a front-end web developer. In this case, you are not working with the model. Similarly, back-end developers only work with the model. In this assignment, the first two problems have you working on a model and the last question has you create a simple view (for the data in the model).

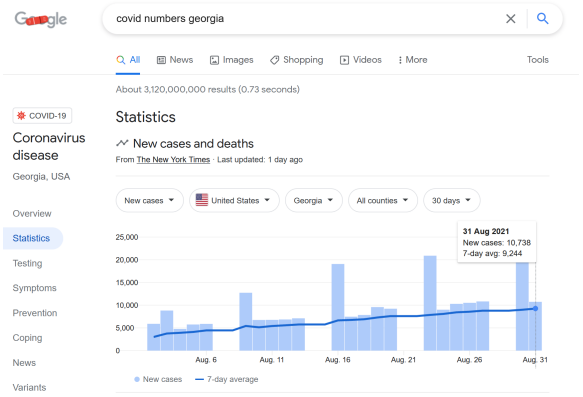
1 Rolling Average [10 marks]

In the past year, 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)[1]. A rolling average

[1]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.

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 number of data points 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 Not-A-Number value. In particular, the first `width`−1 elements of the output array should have this value. Here are some examples:

```
rollingAverage(new double[]{1,2,4}, 1) -> [1, 2, 4]
rollingAverage(new double[]{1,2}, 2) -> [NaN, 1.5]
rollingAverage(new double[]{1,2,4,9}, 3) -> [NaN, NaN,  2.3333333333333335, 5.0]
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. Note that no testing will involve input data that violates these restrictions.

```
data: 0 <= data.length <= Integer.MAX_VALUE
      for each value in data : -101.0 <= value <= 101.0
size: 1 <= width <= Integer.MAX_VALUE
```

Note: There will be some marks allocated for the efficiency of your code. In particular, your code should not take very long to compute the rolling average of a large array with a large width. If you are computing the average for each position in the output array from scratch then your code will be inefficient.

2 Peaks (and Plateaus) [20 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 identifies 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 addition to these rules, we also consider what might be described as *plateaus* to be peaks as well. A plateau occurs when a local maximum value is repeated one or more times consecutively. For example, in the numbers $\{1, 2, 8, 1\}$, the value at index position 2 is a peak with value 8. In the numbers $\{1, 3, 7, 7, 1\}$, the values at index positions 2 and 3 are a plateau (with value 7).

In the provided `A1Q2.java` file, you will complete the `peaks()` method which finds all peak (and *plateau*) values and 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[2]. 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 the peaks in the input array. For a *plateau*, the value and *starting* index position of the plateau will be used. For example,

```
// 2 peaks in the data
double[] data1 = {1.1, 2.2, 1.1, 2.2, 3.3};
double[][] peaks1 = peaks(data1, 0.00001);
// assert: peaks1 == { {2.2, 3.3}, {1.0, 4.0} }

// 2 peaks and a plateau in the data
double[] data2 = {1.3, 1.1, 2.2, 2.2, 2.2, 1.1, 2.2, 3.3, 1.2, 1.2, 1.1};
double[][] peaks2 = peaks(data2, 0.00001);
// assert: peaks2 == { {1.3, 2.2, 3.3} , {0.0,2.0,7.0} }

// 1 wide plateau because epsilon is so big! (plateau is still a peak!)
double[] data3 = {1.3, 10.1, 12.2, 12.2, 12.2, 11.1, 12.2, 13.3, 11.2, 11.2, 1.1, 1.0};
double[][] peaks3 = peaks(data3, 5);
// assert: peaks3 == { {10.1}, {1.0}}
```

When identifying a plateau, you should test for equality (using `epsilon`) of values against the *first* value of the plateau and not neighbouring data values as you move through a plateau. In the last example above, this means using the value at index 1 to test for equality with all values from index 2 to 9 in the plateau and the value with 10 to identify the end of the plateau.

In a plain text file called `plateaus.txt`, write a brief explanation of why you should *not* use neighbouring values when finding a plateau. Include example input (data and epsilon) to illustrate this.

Note: finding the *peaks* 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()` method does. In practise, it can be more complicated than as described here because the uncertainty in the measured data must also be considered.

[2]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 $|x - y| < epsilon$.

3 View [20 marks]

In this problem, you will implement a simple console view to display some data. In the provided `A3Q3.java` file, there are two plotting methods: one is already implemented for you and the other is not. Each takes an array of exactly 20 integers and returns a String that when printed shows a (bar) plot of the data. All values in the input arrays are integers between 0 and 20 (inclusive).

```
// plot the data in a rotated orientation (bars are left-right)
// this method is already implemented for you
public static String plotHorizontal(int[] data)

// plot the data in the usual orientation with bars shown up-down
public static String plotVertical(int[] data)
```

It is important that the output string be *exactly* as described as follows. When testing your code, we will be comparing the String that your method outputs with the expected String and they must be exactly the same.

`plotHorizontal` This method is provided for you. You can use it as an exmaple of working Java code. Here is the description of it. The output strings corresponds to 22 lines that are each newline terminated. When printed, the output will look like the following:

```
--+--------------------+
01|####                |
02|###########         |
03|####################|
04|####################|
05|#################   |
06|#####               |
07|################### |
08|######              |
09|#################   |
10|#                   |
11|#                   |
12|##########          |
13|############        |
14|#                   |
15|                    |
16|######              |
17|###########         |
18|                    |
19|###                 |
20|#                   |
--+--------------------+
```

where the first and last lines are "–+——————————+\n", and each of the twenty lines in between show numbers 01 to 20 followed by a pipe "|", between zero and twenty number signs (#), another pipe and then ending with a newline character. The number of number signs is given by the input array. For example, line 01 has `data[0]` number signs, line 02 has `data[1]` number signs, ..., line 20 has `data[19]` number signs.

Note: The plotHorizontal will always return a string that has length 550.

`plotVerticle` **This is the method that you must implement**. Here, the bars in the plot are vertical (as they normally are when you make a bar plot in excel, for example). The output string corresponds to 24 lines that are each newline terminated. When printed, the output will look like the following:

```
+--------------------+
|  ##                |
|  ##  #             |
|  ##  # #           |
|  ### # #           |
|  ### # #           |
|  ### # #           |
|  ### # #           |
|  ### # #   #       |
|  ### # #   #       |
| #### # #   #   #   |
| #### # #  ##   #   |
| #### # #  ##   #   |
| #### # #  ##   #   |
| #### # #  ##   #   |
| #### ###  ##  ##   |
| ########  ##  ##   |
|#########  ##  ##   |
|#########  ##  ## # |
|#########  ##  ## # |
|############### ## ##|
+--------------------+
|00000000011111111112|
|12345678901234567890|
```

where the first line is "+——————————+\n"; the next 20 lines start with a pipe, have between zero and twenty number signs, another pipe and then end with a newline character; the 22nd line is the same as the first; and the last two lines show the numbers 01 to 20 as shown above. Just as with the plotHorizontal method, the number of number signs comes from the input array.

Note: The plotVertical method must return a string of length 552. You can use the `.length()` method to check when testing your code.

Note: The two example plots above are for the same input array.

Notice that the input to both of these methods are integer arrays of a fixed length with a strong restriction on the integer values [0-20]. In order to be able to plot arbitrary data (like the data from the rolling average, for example), you will also implement another method that takes an array of floating point numbers and returns an array of integers that is an averaged and scaled version of the input array. The output of this method will then become the input for the plotting methods.

```
public static int[] averageAndScale(double[] data)
```

A Python implementation of this method is provided for you use as a guide for your solution. In particular, this part of the assignment is asking you to translate working Python code into working Java code.

`averageAndScale` This method will average neighbouring data values so that there are at most 20 values (averages) and then scale those averages so that the largest is 20. All of the original numbers will non-negative.

Suppose first that the length of the input array was exactly 20. The method would simply scale the data so that the largest value is scaled to 20 and all other values are modified by the same scaling factor and then rounded to the nearest integer. These integers would make up the new integer array to be returned. If the length of the input array was less than 20, then you would follow the same procedure for as many values as there are in the input array and then simply leave the last values in the array to be zeros (remember the length of the output array must be exactly 20).

If the length of the input array is a multiple of 20, then we would first compute averages using length/20 neighbouring data values (making bins for a bar plot) and then scale as above. For example, if the length of the array was 80, then you would use the first 4 values for the first average, then the next four for the second average, etc., to create 20 averages and then scale these as above. In this case there will be exactly 20 averages to work with.

If the length is greater than 20 but is not a multiple of 20, then more care must be taken. We'll illustrate this with an example. Suppose the length of the input array is 53. Since $53.0/20 = 2.65$, we need to average more than 2 values together. Since 3 is the smallest integer that is larger than 53.0/20, we would compute as many averages with 3 neighbouring values as the data allows. We can create 17 averages (requiring 17*3 = 51 of the values from the input array). This leaves only two values remaining, so we compute the average of these two values to give us our 18th average. We now have 18 values (the averages) and proceed to scale them as we did in the cases above. Notice that we will have one or more zeros at the end of the output array and that the last average will be computed with a different number of values than the first ones.

Submission Recap

A complete assignment will consist of three `.java` files (`A1Q1.java`, `A1Q2.java`, and `A1Q3.java`) and a single `.txt` file (`plateaus.txt`).

`A1Q1.java`: 20 marks for `rollingAverage`
`A1Q2.java`: 30 marks for `peaks`
`plateaus.txt`: 10 marks
`A1Q3.java`: 30 marks for `averageAndScale`, 10 marks for `plotVertical`

Remember that Brightspace will only keep your **last** submission.
Be sure to submit **all** files you have worked on **each** time you submit.