ECE 150 CODING LAB 3: MEDIAN

SUBMISSION DUE *at the end of the Lab Period*

perfer et obdura; dolor hic tibi proderit oli

Statistics: Median

Write a program, `median`, that accepts several (one or more) command-line arguments that will be floating-point numbers representing a series of voltage readings. Valid voltage readings are positive numbers less than 150. The list of readings will be terminated by a negative number. Your program should compute the median of the *valid* voltage readings.

The program will be executed as follows:

```
./Median <num_1> <num_2> ... <num_k> -1
```

where "`<num1>`", "`<num2>`", ... are valid floating-point numbers within the range `[-FLT_MAX, FLT_MAX]` and you may use "`atof`" (ASCII-to-float, requires `<stdlib.h>`) to convert the arguments.

The output should be of the form:

```
Number of voltage readings: ...
Median voltage: ...
```

If any voltage readings are invalid then you should output, to `cout`, the warning message:

```
Warning: invalid voltage reading <...> ignored in calculation
```

where `<...>` is whatever the invalid value was. If there are any invalid voltage readings, you should return a positive integer when your program completes.

You may compute the median using whatever algorithm you prefer, though a relatively straightforward technique is to sort the data using your favourite sorting algorithm (there should be sample sorting code on Learn for at least bubblesort of an `int` array) and then select the middle element. If the dataset has an even number of elements, then the median is the average of the middle two elements. You will need to copy the data from `argv[]` into an array if you use this method.

If there are any errors in the input data set, you should output, to `cerr`, the message:

```
Error: Unable to compute median over data set because ...
```

and exit with a negative return code (*i.e.*, `return -1;`).

Submit your work to http://marmoset01.shoshin.uwaterloo.ca project **L3-Median**. Your submission file must be named `Median.cpp`. Note that the filename is *case sensitive*.

Additional knowledge required to complete this exercise:

- Creation and use of arrays
- Sorting

ECE 150 HOMEWORK 3: STATISTICS AND ROOT FINDING

SUBMISSION DUE WEDNESDAY, SEPTEMBER 26th AT 10:00 PM

Why are all the quotes in Latin?

1. Statistics: Standard Deviation [1 marks]

While minimum, maximum, average, and median are useful, they are far from the complete picture. For example, if we have the datasets:

```
0 12 15 25 33 87 95 95 98 100
```

and

```
0 38 57 60 60 60 60 62 63 100
```

they have the same maximum (100), minimum (0), average (56), and median (60) but they are clearly quite different. In the next few exercises you will write programs to compute other statistics that will help to capture this difference in distribution.

The most common measure of data spread is standard deviation, which comes in two forms: *population standard deviation*, σ, and *sample standard deviation*, s.The difference between these two is that the first assumes that we have data for the entire population while the second assumes that we have just a sample of data from the entire population. For example, if we compute the standard deviation of grades for ECE 150-3 students in Fall 2018, since we have all of the grades for all of the students, we can compute a population standard deviation. Conversely, if we wanted some idea of the standard deviation of grades of first-year engineering students in Fall 2018 at Waterloo, and all we have is the ECE 150 grades, then we can compute a sample standard deviation.

While the specific computation depends on the distribution of the dataset, for this question, we will assume that the data is normally distributed (the so-called bell-curve), even though that is often not the case in natural and engineering phenomena (and is certainly not the case in the first of the above examples). Under this assumption, the population standard deviation, σ, is computed according to the equation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where N is the number of elements in the dataset, x_i is the i^{th} element in the dataset and $\overline{x}$ is the average of the elements in the dataset.

The sample standard deviation, s, is:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where the meanings of terms is the same as the previous case.

For this question, you are required to write a program, `StdDev`, that accepts several (one or more) command-line arguments that will be floating-point numbers representing a series of voltage

readings. Valid voltage readings will be positive numbers less than 150. The list of readings will be terminated by a negative number. Your program should store *valid voltage readings* in a floating-point array; any invalid voltage readings should be rejected with a warning message to `cout`. Your program should then compute the population standard deviation and sample standard deviation of the valid voltage readings.

The program will be executed as follows:

```
./StdDev <num1> <num2> ... -1
```

where "`<num1>`", "`<num2>`", ... will be replaced with floating-point values. You may assume that the arguments will be valid floating-point numbers within the range `[-FLT_MAX, FLT_MAX]` and you may use "`atof`" (ASCII-to-float, requires `<stdlib.h>`) to convert the arguments.

The output should be of the form:

```
Number of voltage readings: ...
Population standard deviation: ...
Sample standard deviation: ...
```

where "..." are the results of your calculations. If any of your calculations require a non-zero finite number to be divded by zero, the "`...`" should be output as "`infinity`" (positive or negative, according to the sign of the numerator). If any of your calculations require zero divded by zero, the "`...`" should be output as "`undefined`"

If there are any errors in the input data set, you should output the message:

```
Error: Unable to compute standard deviations over data set because ...
```

to `cerr`.

Submission: You should submit the result of this exercise to the project **H3-StdDev**. Your submission file must be named `StdDev.cpp`. Note that the filename is *case sensitive*.

Knowledge required to complete this exercise:

- Creation and use of arrays

2. Statistics: Histogram [1 marks]

In Question 1 you calculated the standard deviation on the assumption that the data set you received was normally distibuted (*i.e.,* a bell-curve distribution). If the data is not normally distributed, the calculated deviation can be at best of limited value, and quite possibly extremely misleading. For example, the first sample data set in Question 1 was bimodally distributed: when the data is considered as a whole, the deviation is greater than 40; if the data is subdivided into its two natural modes, one mode has a deviation of 12 around a mean of 17, while the other has a deviation of 5 around a mean of 95. This begs the question: how can you determine if data is normally distributed? While there are several techniques, one of the simplest techniques is to create a histogram of the data and visually compare it to a bell curve.[1]

For this question, you are required to write a program, `Histogram`, that accepts several (one or more) command-line arguments that will be floating-point numbers representing a series of voltage readings. Valid voltage readings will be positive numbers less than 150 terminated by a negative number. The voltage readings we are most interested in are those between 106 V and 127 V,[2] and so we will create buckets for every 3 V increments in that range: [106,109), [109,112), [112,115),[115,118),[118,121), [121,124),[124,127]. Note that the last bucket includes 127. In addition to these seven buckets, the following buckets should capture additional possible valid voltage inputs: (0,106) and (127,150). Finally, you should have a bucket for any invalid voltage readings (0 or $\geq$ 150), for a total of 10 buckets.

Your program should create an `int` array of ten elements: the first element should count valid voltages less than 106; the next seven are the seven buckets that each cover the 3 V range; the ninth element should be valid voltages greater than 127 V. Finally, the last element should contain invalid voltages.

The program will be executed as follows:

```
./Histogram <num1> <num2> ... -1
```

where "`<num1>`", "`<num2>`", ... will be replaced with floating-point values. You may assume that the arguments will be valid floating-point numbers within the range `[-FLT_MAX, FLT_MAX]` and you may use "`atof`" (ASCII-to-float, requires `<stdlib.h>`) to convert the arguments.

The output should be of the form:

```
Number of voltage readings: ...
Voltage readings (0-106): ...
Voltage readings [106-109): ...
Voltage readings [109-112): ...
Voltage readings [112-115): ...
Voltage readings [115-118): ...
```

[1] See `https://en.wikipedia.org/wiki/Normality_test`

[2] Why this range?

```
Voltage readings [118-121): ...
Voltage readings [121-124): ...
Voltage readings [124-127]: ...
Voltage readings (127-150): ...
Invalid readings: ...
```

where "..." are the results of your calculations.

If there are any errors in the input data set, you should output the message:

```
Error: Unable to compute histogram over data set because ...
```

to `cerr`.

Submission: You should submit the result of this exercise to the project **H3-Histogram**. Your submission file must be named `Histogram.cpp`. Note that the filename is *case sensitive*.

Knowledge required to complete this exercise:

- Creation and use of arrays

3. Statistics: Mode [1 marks]

While deviation captures the spread of a dataset, we would like to be able to capture the common case of a set of data. While this appears to be similar to the average or median, and in a normal distribution it is likely to be similar, in many cases it is quite different.[3] To do this, we compute the *mode* of our data set.

Mode is not suited to floating-point datasets[4] and so for this question we will assume the data set is of type `int`. Specifically, you will be given a template C++ file that accepts several (one or more) command-line arguments that will be floating-point numbers representing a series of voltage readings. It will convert these readings into their closest `int` equivalent and then call the `mode()` function that you will write and output to cout the results of that function.

The signature of your `mode()` function should be:

```
int mode(const int dataset[], const int size, int modes[]);
```

The `dataset[]` parameter is the array of integers that the template code will pass to your function, while the `size` parameter is the number of items in that dataset.

Since we do not know in advance how many modes there will be, the return value of the `mode()` function will be the number of modes that you find. To return the actual modes, the function will accept the parameter `modes[]` and it should fill in the array `modes[]` with the values of the modes that the function found. The modes should be stored in `mode[]` in order of increasing size. For example, with the dataset:

```
7 5 2 5 7 0 4 2 7 0 5 9 2
```

your function should return 2 and fill in `modes[0]` and `modes[1]` with the values `2` and `7`, respectively. The `modes[]` array is guaranteed to have sufficient space for all modes you could find.

[3] For example, the mode of the first dataset in Question 1 is 95, which is nowhere near the average or median. If if you want a more vivid example, consider the *fact* that the average person has less than two legs! Fortunately, that is not the common case.

[4] Technically, floating-point numbers present a large number of discrete choices, rather than a continuous range, and so in principle mode is possible over a floating-point dataset. In fact, on a typical four-byte machine, there are no more floating point numbers than there are integers (both fit in a 32-bit quantity, and so there are precisely 2^{32} possible choices for each datatype!), and so it may seem strange to use mode with integers but not with floating-point numbers. The deeper reason is that we rarely, if ever, would use mode if we actually had a large range in our integer set. For example, even if our range is just 100 (say, grades in a course from 0 to 100), mode is of somewhat limited value unless we had a very sizable dataset (In ECE 150-3, if grades were uniformly randomly distributed over 0 to 100, then the expected number of datapoints for each grade would be slightly more than 1! Even allowing for a more plausible distribution curve, the likelihood that the mode represented a reasonable measure of the typical student performance is low.). Typically if dealing with either floating-point data or integer data over a large range, we would first histogram the data into a more manageable number of buckets (maybe 10 to 20; for grades, either every 10% or every 5%) and then compute the mode of those buckets.

There can never be a negative number of modes, and so your `mode()` function should return `-1` if it cannot compute the mode(s) for what every reason.

You may need/want to sort the dataset to help you in computing the modes. If you do this, please note that the dataset passed to the `mode()` function is declared `const` and so you cannot change that dataset.

Given that you are implementing a function, we will provide a template `main()` function in the file `Mode.cpp` that you may edit as you see fit to test your functions.

All code will be subject to style checking. Your function should not output anything to either `cout` or `cerr`.

Submission: You should submit the result of this exercise to the project **H3-Mode**. Your submission file must be named `Mode.cpp`. Note that the filename is *case sensitive*.

Knowledge required to complete this exercise:

- Creation of a function
- Passing data to a function in an array
- Returning a value from a function
- Returning array data from a function

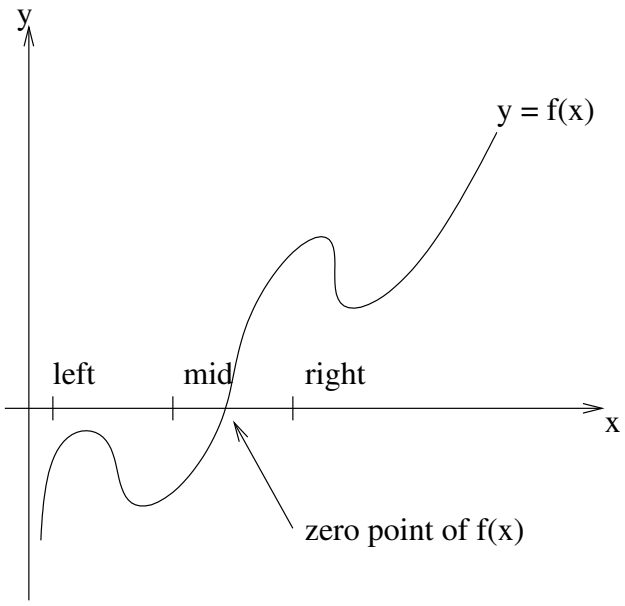
4. Bisection Root Finding [1 marks]

In Homework 1, Question 3, when calculating the square-root of a value we used the Babylonian method. This method is a specific instance of the Newton-Raphson method for finding the zero points of a function $f(x)$ (*i.e.,* the values of x for which $f(x) = 0$), applied to the function:

$$f(x) = S - x^2$$

This method works well for calculating the square-roots, or even N^{th}-roots, because the equation is relatively stable and well behaved. For other functions, the method is very sensitive to the choice of the initial guess and if that is poorly chosen, the method will fail badly.

An alternate technique that is extremely robust is the Bisection Method which is described briefly in Wikipedia.[5] This method starts with two initial values, x_l and x_r, representing an interval $[x_l, x_r]$, where the function crosses the x-axis (*i.e.,* $f(x_l)$ and $f(x_r)$ have opposite signs). The value of the function at the interval midpoint is calculated and then the interval size is reduced to the half that contains the zero-crossing point. For example, in the figure below:



the function is below the x-axis at the "left" point and above it at the "right" point. At the middle of that interval, the function is below the x-axis, and so we can reduce the interval from [left,right] to [mid,right]. This process continues until the interval size has reached a sufficiently small range.[6] Given that this method is a progressive reduction of interval, it can be implemented recursively roughly as follows:

[5] See `https://en.wikipedia.org/wiki/Root-finding_algorithm`

[6] Question: Why do we want to specify an interval size? Why don't we simply require that the value of the function at the interval midpoint is within a desired precision?

```
float bisection(const float left, const float right,
                const float minIntervalSize) {
  if (invalid parameters)
    return NaN;
  determine midpoint;
  if (interval is less than minIntervalSize)
    return midpoint;
  if (midpoint replaces left)
    bisection(midpoint, right, minIntervalSize);
  else
    bisection(left, midpoint, minIntervalSize);
}
```

However, this approach is still not quite sufficient. It is possible that the minimum interval size is too tightly specified, and therefore not achievable. As such, you need to also implement a termination criteria that will prevent the `bisection()` function from looping infinitely.[7] To do this, you must implement a bisection helper function:

```
float bisectionHelper(const float left, const float right,
                      const float minIntervalSize,
                      ... ) {
  // Implement above pseudocode in this function
  // NOT in bisection function
  // Add in necessary termination capacity
}

float bisection(const float left, const float right,
                const float minIntervalSize) {
  // invoke bisectionHelper() with extra parameter(s)
}
```

where "`...`" is the additional parameter needed to support termination when the requested interval is not achievable.

For this question you are required to implement both the `bisection()` function and the recursive `bisectionHelper()` function. Your bisection method may assume that the template and testing code will define the function whose root you are computing as follows:

```
float f(float x) {
  return ...
}
```

[7] It will not actually loop indefinitely under this condition. It will crash instead. Why?

Given that you are implementing functions, we will provide a template `main()` function in the file `Bisection.cpp` that you may edit as you see fit to test your functions.

Submission: You should submit the result of this exercise to the project **H3-Bisection**. Your submission file must be named `Bisection.cpp`. Note that the filename is *case sensitive*.