CSCI-1200 Data Structures — Fall 2024
Lab 2 — STL Strings and Vectors

This lab will give you a bit more practice with C++ compilation, command line arguments, and the STL `string` and `vector` classes. First, let's download the statistics code that we studied in Lecture 1:

http://www.cs.rpi.edu/academics/courses/fall24/csci1200/labs/02_strings_and_vectors/statistics.cpp
http://www.cs.rpi.edu/academics/courses/fall24/csci1200/labs/02_strings_and_vectors/input.txt

Go ahead and compile the code and then run it on the provided input:

```
clang++ -Wall -Wextra statistics.cpp -o stats.out
./stats.out input.txt
```

Checkpoint 1

estimate: 30-40 minutes

Your task for the first checkpoint is to modify the provided code to also compute the *mode*, that is the *most frequently occurring item* in the input. Continuing with the design of the provided code, let's add a new function to the file named `compute_mode`. You should modify the main function to call your new function and then add to the output to STDOUT to print this new calculation. Don't change any of the other functions in the file.

Think carefully about the prototype of your new function (return type and number and type of arguments). Should your argument(s) be pass-by-value (a.k.a. pass-by-copy) or pass-by-reference? For this checkpoint/lab we will only use the `vector` class (even if know other data structures that could help complete this task, don't use them now).

Test your function on the provided input, but also make several additional test input files. How does your function handle "the corner cases" for mode? A collection of data can have no mode – this happens if every item appears at most one time. Or it can have multiple modes – if two or more items both appear the same number of times and no other item appears more frequently. Update your implementation to handle these cases.

What assumptions are being made by this program about the data in input file? Does the program work for all inputs? Can you prepare an input file that breaks this program; e.g., giving a wrong answer, confusing answer, and/or crashing the program? NOTE: You don't need to change the provided code! Writing error-checking code and handling unusual/flawed input can be excessively time-consuming and distract us from our actual assignment/purpose. But let's document the expectations/requirements of this program.

To complete this checkpoint: Show your TA/mentor your tested and debugged function and discuss the testing you've done and how you would formally document the input requirements and the limitations of this program.

Checkpoint 2

estimate: 30-40 minutes

To get started with this checkpoint, make a copy of your code from the previous checkpoint and modify it to work with floating point input values. You can almost do a search-and-replace of `int` with `float`, but don't replace *all* of the `int`s! Compile and test the modified program and make sure it still behaves as expected.

Now let's add another function `closest_to_average` will determine which input value is closest to the average of all of the values.

For example, if the input file contains:

```
1.5 3.5 4.0
```

The average is `3.0` and the closest value to the average is `3.5`.

If the input file contains:

```
11.5 17.8 9.9 12.2 -1.7 89.2 65.1 27.1 40.2 9.4 -2.5 26.1 37.1 44.3 56.8
```

Then the average is `29.5` and the closest value to the average is `27.1`.

Other than switching to float, adding the `closest_to_average` function, and calling it from the `main` function (so you can print out the result), don't modify any of the other functions in the file. Again, think carefully about the prototype of your new function (return type and number and type of arguments). Should your argument(s) be pass-by-value (a.k.a. pass-by-copy) or pass-by-reference?

To complete this checkpoint: Show a TA/mentor your tested and debugged program.

Checkpoint 3 will be available at the start of Wednesday's lab.