COMP 2710 – Project 3: A SONG OF FILES & DATA

Points Possible: 100
Deadline: **11:59pm, Friday, Oct 25th, 2024 (Central Time)**

Goals:

- Able to understand coding requirements and have a logical workflow.
- Able to use input/output streams, file read/write tools and access methods.
- Able to detect illegal inputs.
- Able to work with C++ strings.
- Able to use arrays/arrays of structure to group data elements.
- Able to apply functional programming in a flexible manner.
- Able to create testcases and perform integration testing.
- Able to understand and design a simple sorting algorithm.
- Able to manage multiple project versions with git branching.

Short instruction:

To do this project, start by reading all the Parts and Sections thoroughly. Then, work on each part incrementally, moving on to the next once you've completed a section of the current part.

SCROLL DOWN TO GET STARTED

PART A: PHASE 1 – TIME-INDEPENDENT DATA

I. Description:

Jason, an Auburn student, is compiling results from his wet lab experiments. Each experiment generates a file with K raw measurement values, and he has N such files. Now, the goal is to merge these results and conduct an analysis on the combined data. Subsequently, the findings will be compiled into a single cohesive output. However, the platform used to conduct the experiments is a C++ environment and Jason is not a good coder. Therefore, he needs your help to create a C++ program to accomplish this task and will be eternally grateful.

The program is designed to accept N different text files as inputs. Each file includes a list of K signed decimal numbers in random order. Upon execution, it generates an output text file containing all of Jason's experimental values sorted from smallest to largest, along with statistics for the mean, median, and mode. The mean is the average of all values, the median is the middle value in the ordered list, and the mode represents the value that occurs most frequently.

Note 1: You must provide the following user interface with relatively similar output structure. You do not need to color the texts that are highlighted in red in the sample output. Study this and try to understand what the program is doing. Also, replace the word "Li" with your name.

```
*** Welcome to Li's Data Analyzer ***
Enter the number of files to read: 3

Enter the filename for file 1: input1.txt
The list of 6 values in file input1.txt is:
4.17
0.13
3.14
90.12
-23.66
89.999

Enter the filename for file 2: input2.txt
The list of 5 values in file input2.txt is:
-3.5
74.4
-93
4.46
90.12

Enter the filename for file 3: input3.txt
The list of 4 values in file input3.txt is:
```

```
-1182.33
90.43
1.2294
90.12


*** Summarized Statistics ***

The orderly sorted list of 15 values is:
-1182.33 -93 -23.66 -3.5 0.13 1.2294 3.14 4.17 4.46 74.4 89.999 90.12
90.12 90.12 90.43

The mean is -50.9448
The median is 4.17
The mode is 90.12

Enter the output filename to save: output_phase1.txt

*** File output_phase1.txt has been written to disk ***
*** Goodbye. ***
```

Note 2: If there are multiple mode values, you need to print the average value of all of them. Also, all metrics should be rounded to whatever nearest decimal point you choose.

Note 3: The program should be capable of reading inputs from any location and produce and output to any location, so long as the filename or path is correct. For instance, if `input1.txt` is located in a folder called `folderA`, the user can either type `folderA/input1.txt`, assuming the `.cpp` file containing the `main()` function is in the current working directory; or they can enter the full path from the root folder.

II. Input files:

Ideally, the input should only contain a list of K signed decimal numbers, each of them is separated by a new line, and the last item has a new line. However, since Jason's measuring instruments are not perfect, there are chances that malformed content exists inside, making the files corrupt. If the program comes across these files when reading, it should disregard them immediately and prompt for a newer working file. Here are three examples of such files:

```
453<<<392
74,392
3<>{{{{{kf

Rr,4
```

malform_1.txt

```
<this file is blank>
```

malform_2.txt

```
65,495
-45,466,000
74,39

944.48 58.593 -29.33 -192
```

malform_3.txt

If you feed either of those files into the program, it will ask for another appropriate input file:

```
*** Welcome to Li's Data Analyzer ***
Enter the number of files to read: 1

Enter the filename for file 1: malform_1.txt
Not an input file. Illegal content/structure detected. Please try again.

Enter the filename for file 1:
```

If instead, you entered a name of a non-existing file or path, it should prompt for a retry:

```
*** Welcome to Li's Data Analyzer ***
Enter the number of files to read: 1

Enter the filename for file 1: non_existing_file
File not found. Please try again.

Enter the filename for file 1:
```

A proper and legal input file should look like this:

```
-79
55.83919992
11.49
0.0001
4493.33
-44.5712
```

ideal.txt

Note 4: Keep in mind the content in these files is treated as C++ strings when you read them line by line. You need to find a way to convert these strings into meaningful numeric data. You can search for library functions that convert string-to-float or string-to-double values online.

One final thing is, Jason isn't sure how many files will be produced during the experiment. He may want to input 3 files, but only 1 is available at that time per se. Therefore, the program should allow the user to quit at any time during input and proceed with the calculation. For example, Jason can type "quit" when asked for a filename and the program will move on:

```
*** Welcome to Li's Data Analyzer ***
Enter the number of files to read: 3

Enter the filename for file 1: input1.txt
The list of 6 values in file input1.txt is:
4.17
-23.66
90.12
90.12
-23.66
89.999

Enter the filename for file 2: quit
Input cancelled. Proceeding to calculation...

*** Summarized Statistics ***

The orderly sorted list of 6 values is:
-23.66, -23.66, 4.17, 89.999, 90.12, 90.12

The mean is 37.848
The median is 47.0845
The mode is 33.23

Enter the output filename to save:
```

Now obviously, if he decides to quit on the first input, it should terminate without doing anything:

```
*** Welcome to Li's Data Analyzer ***
Enter the number of files to read: 3

Enter the filename for file 1: quit
Input cancelled. Proceeding to calculation...

*** Goodbye. ***
```

Note 5: Do It Yourself – You should handle the input for the number of files to read. Identify any invalid inputs and display your own message to prompt for a valid entry.

III. Output file:

You only need to produce one output file, and it should have the results after you successfully run the program and instruct it to save the results to a folder. Let's say you name the file as `output.txt`, the content inside should look like the one below.

```
*** Summarized Statistics ***

The orderly sorted list of 11 values is:
-93 -23.66 -3.5 0.13 3.14 4.17 4.46 74.4 89.999 90.12 90.12

The mean is 21.489
The median is 4.17
The mode is 90.12
```

output_phase1.txt

Note 6: Recall that you can quit the program at any time by typing "quit". If you happen to quit on the first input, then no output file should be generated.

IV. The sorting procedure:

After you've read all the data from the files, you'll need to implement a sorting algorithm to arrange the values. This will also help you gather the necessary statistical information thereafter. You can choose whatever type of algorithm that can sort the values from smallest to largest.

Note 7: You are not allowed to use any prebuilt sorting functions from C++ libraries. It is strongly recommended that you implement a basic sorting algorithm instead of more complex ones.

One of the most widely used sorting algorithms is MergeSort, which has a time complexity of $O(N * logN)$, where N is the number of elements in the array. If you are not familiar with it, that just means the time used to sort the array is a multiple of $logN$ to the array size, i.e., if you have 5 elements, then around $5 * \log(5) = 3.49$ elements are accessed during the sorting routine.

It is a highly efficient divide-and-conquer algorithm. The array is initially divided into smaller chunks until it can no longer be divided, after which sorting occurs starting from the smallest arrays and then combining and sorting the larger ones. This approach will be beneficial when you progress to phase 2, where you'll need to sort a more complex structure.

Visit this link on how to do it: https://www.geeksforgeeks.org/merge-sort/

V. Remarks:

Now you are ready to help Jason out! Make sure everything works well before moving on to **Phase 2** of the project. You are suggested to read the entire project description first before doing anything, because you will be asked to version control your code in a more advanced way. It is best to get a grasp of how you should build a structure for this project, so things will go smoothly.

Note 8: Keep in mind that writing your entire project in a single `main()` function is a bad practice. Even Jason will have a hard time going through your code explanation if you cramp everything in. Instead, use what you've learned to implement functional programming effectively. Here are some suggested functions to get started:

1. `array_size <- readfile(inputArray[], instream)`
2. `outputArray_size <- sort(inputArray1[], inputArray1_size, inputArray2[], inputArray2_size, outputArray[])`
3. `void <- writefile(outputArray[], outputArray_size)`

PART A: PHASE 2 – CHRONOLOGICAL DATA

I. Description:

Thank you for helping Jason out! You are truly a saint! Although he needs a little bit more help to upgrade the program that would be compatible with his new sets of experimental data files.

Now, Jason explains that all generated N files will be richer in information if the data from the experiment includes timestamps for each measured K value, as the measurements are taken at different times each time. The program will still accept N different text files as inputs, but now, along with a list of K signed decimal numbers in random order, each file also contains timestamps of when those values are recorded.

Note 9: You should study the following user interface and must have yours in a relatively similar structure, which has been slightly modified from phase 1 to accommodate Jason's request.

```
*** Welcome to Li's Data Analyzer ***
Enter the number of files to read: 2

Enter the filename for file 1: input1.txt
The list of 6 values in file input1.txt is:
4.17      14.22.32
0.13      15.33.38
3.14      19.15.17
90.12     23.23.28
-23.66    10.27.04
89.999    09.33.39

Enter the filename for file 2: input2.txt
The list of 5 values in file input2.txt is:
-3.5      12.24.11
74.4      12.03.12
-93       12.33.13
4.46      12.21.14
90.12     12.49.15

*** Summarized Statistics ***

The orderly sorted list of 11 values is:
-93 -23.66 -3.5 0.13 3.14 4.17 4.46 74.4 89.999 90.12 90.12

The mean is 21.489
The median is 4.17
The mode is 90.12
```

```
The chronologically sorted list of 11 values is:
89.999 -23.66 74.4 4.46 -3.5 -93 90.12 4.17 0.13 3.14 90.12

The mean is 21.489
The median is -93
The mode is 90.12

Enter the output file name to save: output_phase2.txt

*** File output_phase2.txt has been written to disk ***
*** Goodbye. ***
```

Notice that only the median statistic for chronologically sorted data has changed, the rest of the statistics remain the same.

II. Input files:

The updated structure for the input files now includes timestamp data for each value. It has a format of `HH.MM.SS`, where `H` represents the 24-hour, `M` the minute, and `S` the second. This format uses leading zeros for single-digit values, for example, `05.03.01`. The raw values and timestamps are separated by a tab character (`\t`). So, if you see an input file like this:

```
4.17        14.22.32
0.13        15.33.38
3.14        19.15.17
90.12       24.23.28
-23.66      10.27.04
89.999      09.33.39
```

input_1.txt

In the vision of C++, it will be read like this:

```
4.17\t      14.22.32\n
0.13\t      15.33.38\n
3.14\t      19.15.17\n
90.12\t     24.23.28\n
-23.66\t    10.27.04\n
89.999\t    09.33.39\n
```

not an actual file

Note 10: You need to separate and convert the C++ strings into different type of meaningful numeric values when you read the file line by line. Use the separation characters as anchor points to determine which is which, for example, the tab character (\t) or the period (.).

Of course, your program should also deal with illegal inputs where files have malformed contents. The followings are three extra examples involving the timestamps. When you come across these files or any type of errors, you should prompt for another file just like in Phase 1.

```
40          14:22.32
50          15:33.38
60dmd       19:15.17
90          24.23:28
-70         10:27.04
-100        09:33.39
```

malform_4.txt

```
40                  1422
50[these are spaces]15.33.38
60                  -99.34
90[these are spaces]24:23
-70                 10:27.04
-100                09.39
```

malform_5.txt

```
4.17      14.28.000283
0.13      156.54.39
3.14      194.15.17
90.12     -24.235.28
-23.66    10.270392.04
89.999    09.33.39.584

264.38    12.12.12 -99.12     12.13.14 264.38    12.12.12 -99.12
12.13.14
```

malform_6.txt

III. Output file:

Like Phase 1, one output file is all you need, but it should contain the updated results that Jason needs in Phase 2. Hence, it should look something like this:

```
*** Summarized Statistics ***

The orderly sorted list of 11 values is:
-93 -23.66 -3.5 0.13 3.14 4.17 4.46 74.4 89.999 90.12 90.12
```

```
The mean is 21.489
The median is 4.17
The mode is 90.12

The chronologically sorted list of 11 values is:
89.999 -23.66 74.4 4.46 -3.5 -93 90.12 4.17 0.13 3.14 90.12

The mean is 21.489
The median is -93
The mode is 90.12
```

output_phase2.txt

IV. Guide to approach Phase 2:

You might be thinking that you can use as many arrays as you like to store timestamps or any other information. However, this approach won't work as expected. Imagine sorting one array while needing to ensure the others are linked to it—that could be a nightmare. Instead, consider these questions to help you develop a better strategy for Phase 2:

1. You need to store multiple pieces of information—such as the raw value, hour, minute, and second—within a single "object." What C++ data structure do you think would be suitable for this?
2. Can you create an array-like structure from multiple such objects, like the array of values in Phase 1?
3. With that structure in place, can you apply the same sorting algorithm used in Phase 1 using it? How about the way you calculate the statistical values?
4. Think about the sorting order based on the timestamp values. What should you sort first, next, and last? Can you incorporate that order into your sorting algorithm?

If you can answer those questions, congratulation, you have done a marvelous job! Jason is eternally grateful for your kindness!

PART B: GETTING FURTHER INTO THE GITTY BIT

In the previous project, you have learned the basics of using the `git` command to track and commit two simple snapshots of your code in your repository, or repo. In this project, you will learn to use the concept of `git` branching and track your whole project in subsequent phases.

Here is one resource we think will be extremely helpful to understand the concept: https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

Turn your project folder into a `git` repo and follow the instructions below.

Note 11: Make sure there is a `[.git]` folder in your repository before attempting to do anything because this will be where all `git` information are stored. Use `ls -a` command to see whether it is there. If not, reinitialize your project folder to a `git` repo by using `git init`.

I. Your `master` branch:

When you start committing a file to a repository, you'll be committing to the `master` branch by default. This will be where you'll complete Phase 1 and do all related code commits.

You're required to make at least 5 commits here. While you can choose any content for these commits, we suggest organizing your project into subphases so you can commit them like this:

Commit 1. Reading inputs from a single file.

Commit 2. Reading multiple files.

Commit 3. Merging and sorting values.

Commit 4. Calculate relevant statistics.

Commit 5. Output all results to the terminal.

Commit 6. Writing your outputs to a file.

If you do not know how to do these commits, ***you might have been cheating*** on Project 2 and need some refresher on how to do this.

Once Phase 1 is completed, review your commit history and ensure it includes everything you've documented. Here's how to check and see the changes: https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

Note 12: Please be honest with your work. Since you can view your commit history and track changes, we can do the same. If you comment "change this" in your commits but actually "changed that," it will be considered as academic dishonesty.

II. Let's get branchin':

Since you need to update your program to match Jason's request and you do not want to screw up your Phase 1 code, you'll need to create a new branch to transition your work safely.

Feel free to name this branch whatever you like and check it out. Now, you can work on the same `.cpp` file(s) while having `git` help you to stay off the `master` branch.

In your Phase 2 branch, you're also required to make at least 5 commits. We won't provide specific examples for this branch, as we encourage you to exercise your creative freedom. Document your progress with your own comments as you work through Phase 2 of the project.

Note 13: Be careful! Ensure that you're on the correct branch before continuing your work on the same file(s). If you mistakenly complete your Phase 2 work in the master branch and make a commit, you'll be in a long ride to restore Phase 1 code.

III. Testing on `master`:

Now, return to your `master` branch. If you've done everything correctly, you'll see that your code file(s) have reverted to Phase 1, and it is untouched. At this point, you may proceed with the testing for Phase 1, which will be covered in **Part C**.

Note 14: Do not add your testcases to your `git` repository! Git manages versions of your code, not data. If you do this, you could lose your testcases during branch merges.

IV. Testing on Phase 2 branch:

After you are done testing on Phase 1 branch, go back to the branch containing your Phase 2 codes and proceed with the testing.

V. Merging the two branches:

Congratulations! You've tested your program and ensured everything is running smoothly. Your Phase 2 code is now ready to be merged with Phase 1 to create the complete program that Jason

has requested. Proceed with the merge and ensure that `git` is now recognizing your `master` branch as the main branch, and it has your updated Phase 2 code.

PART C: TESTING YOUR PROGRAM

For this project, you will be testing your program using **Integration Testing (a.k.a., Integration and Testing)**, in which individual software modules are combined and tested as a group. The goal is to create as many testcases as possible to cover all potential inputs.

Before proceeding with this section, you should have fully read Part A and B to understand the project. You should consider all edge/corner inputs to thoroughly test your program, as how much you cover will contribute to your grade for this section.

The following guide will help you complete this part.

I. PHASE 1 – the `master` branch:

1. Create a folder named "`regular_test`." Within this folder, create 5 subfolders and populate each with several legal input text files in increments of 5. For example, the first subfolder will have 5 text files, the second will have 10, the third will have 15, and so on. You can name them whatever you like but make it simple and do not include spaces.

   Each of these files should have at least 5 legal values of whatever range you choose, but they should prove that your program does the job. You can search online for a number generator that can help you out.

2. Now test your program using the files in the subfolders. Remember any valid inputs of the path/direct path to the file + the filename itself should be accepted ---> see **Note 3**.

   Verify the accuracy of the outputs and their relevant statistics.

   While you are testing, create another folder to store the outputs in each subfolder. Name and save your output file in that folder for each testing session.

3. Pick any of the subfolder and rerun the program.
   At any time, abruptly type “quit” when it asks for filename.

   Verify that it stops asking for input and proceed to calculation if it has files inputted prior to exiting; or quit entirely if there aren’t any. Name and save the output file in the same folder as **Step 2**. Be careful not to overwrite your **Step 2** output file.

4. Now create a second folder called "`corrupt_test`." Again, populate it with 5 subfolders and name them whatever you like. But now, for each subfolder, you will include different types of malformed/corrupted text files along with some legal working ones (however many you choose). Clearly identify the illegal files with appropriate names.

There is no minimum number of raw values in each legal file for this step, but they should be legal, nonetheless. You can copy some from **Step 1**.

5. Now test your program with these files. Verify that it can detect these corrupted files, and the outputs and their relevant statistics are correct.

   While you are testing, create another folder to store the outputs in each subfolder. Name and save your output file in that folder for each testing session.

6. Create a third folder called "`sorting_test`" and add 5 subfolders into this folder. Name them whatever you like. Then, populate these subfolders with input values that can corner test your sorting function. For example, you can include cases where all values are the same, or all are negative, all are positives, etc.

7. Repeat **Step 5**.
   Verify that it sorts correctly, and the outputs and their relevant statistics are correct.

8. Now put all these files into a folder called "`phase_1_test`" and you are done.

II. PHASE 2 – the other branch:

1. Create a folder called "`phase_2_test`" and go into this folder.

2. Do the same procedure in **Section 1**, **Step 1** and **2**. But now, each of these files should have at least 5 legal values along with randomly generated legal timestamps.

3. Do the same procedure in **Section 1**, **Step 3**. Be careful not to overwrite your previous step output file.

4. Do the same procedure in **Section 1**, **Step 4** and **5**. But now, add in a mixture of corrupted timestamps and/or corrupted values.

5. Do the same procedure in **Section 1**, **Step 6** and **7**. But now, add several edge examples that can sort the raw values chronologically. For example, same timestamps, same hour, same minute, and/or same second values, etc.

Gather all these testcases and neatly organize them in different folders and name them appropriately. Congratulation, you have finished testing your program! Remember that we use no other than these testcases you have made to grade your project.

PART D: SAMPLE CODES

1. The following code shows you how to use arrays and their sizes as input parameters of functions.

```
//Sample code for Project 3
#include <fstream>
#include <iostream>
using namespace std;

const int MAX_SIZE = 100;

//Input: (1) Array storing data retrieved from the file (i.e.,
instream)
//       (2) input file stream object
//Output: Size of array. Note: you need to use this parameter to
control the array size.
int readfile(int inputArray[], ifstream& instream);

int main() {
    ifstream inStream1;

    int iArray1[MAX_SIZE];
    int iArray1_size;
    int iArray2[MAX_SIZE];
    int iArray2_size;

    inStream1.open("input1.txt");

    iArray1_size = readfile(inputAry, inStreamFirst);

    inStreamFirst.close( );

    return 0;
}

int readfile(int inputArray[], ifstream& inStream) {
    int index;

    inStream >> inputArray[index];
    while (! inStream.eof()) {
        cout << inputArray[index] << endl;
        index++;
```

```
        inStream >> inputArray[index];
    }

    return index;
}
```

2. The following code shows you (1) how to retrieve a file name from your keyboard, (2) how to open a file, and (3) how to read data from the file.

```
#include <fstream>
#include <iostream>
#include <cstdlib>  //for exit()
using namespace std;

int main() {
    ifstream inStream;
    int data;

    cout << "file name:";
    cin >> filename;
    cout << "entered filename is:" << filename << endl;

    // Pass the file name as an array of chars to open()
    // inStream.open(filename);
    inStream.open((char*)filename.c_str());

    if (inStream.fail()) {
        cout << "Input file opening failed." << endl;
       exit(1);
    }

    inStream >> data;
    while (!inStream.eof()) {
        cout << data << endl;
       inStream >> data;
    }
    inStream.close();

    return 0;
}
```

PART E: GRADING AND POLICIES

I. Grading:

100 points maximum, criteria ordered by parts, and from lowest to highest in point value.
All these criteria work in harmony, so if you miss one or two, you might be doing so on others.

PART A: (35 points), of which

PHASE 1 (20 points, all carry equal point value).

1. Your program can read multiple text files from any legal location/path.
2. Your program can detect non-existing files, malformed/corrupted files, and illegal input for the number of files.
3. Your program can quit at any time, even on the first input, and it behaves correctly.
4. You must close files after using them.
5. You can write a working sorting algorithm by yourself. It should work on virtually any range of values.
6. You can apply functional programming into your project.
7. Your program can write to a file, and it has the correct output contents.
8. Your program calculates all statistical metrics accurately.

PHASE 2 (15 points, all carry equal point value).

1. You can define a proper structure to hold additional data instead of arrays.
2. Your program can also read text files with timestamps.
3. Your program can also handle malformed/corrupted timestamped files.
4. Your program can also sort the raw data chronologically using any legal values.
5. Your program can write to a file, and it has the correct and updated output contents.
6. Your program calculates all additional statistical metrics correctly.

PART B: (25 points, all carry equal point value)

1. You have at least 5 commits in your master branch.
2. You have at least 5 commits in your branch holding **Phase 2** codes.
3. You can test your master branch using your testcases.
4. You can test your branch holding **Phase 2** codes using your testcases.
5. You can merge the two branches together.

PART C: (25 points)

1. **(5 points)** Your program works with any of your testcases for both sections.
2. **(10 points)** Follow all steps in **Section 1** for your master branch and have all required testcases neatly organized in their respective folders.
3. **(10 points)** Follow all steps in **Section 2** for your other branch holding **Phase 2** codes and have all required testcases neatly organized in their respective folders.

MISCELLANEOUS: (15 points)

1. **(2 points)** Use comments to provide a heading at the top of your code containing your name, Auburn Banner ID, filename, and how to compile your code for both phases. Also describe any help or sources that you used.
2. **(2 points)** Your zipped tarball file should be named like this:

   `project2_LastName_UserID.tgz`

   For example, `project2_King_pzk0039.tgz`.

   You will not lose any point if Canvas automatically changes your file name (e.g., `project2_LastName_UserID-1.tgz`) due to your resubmissions.
3. **(3 points)** Your source code is of good quality, easy to read and well-organized.
4. **(8 points)** You must follow the specified user interface/example outputs for each phase.

Note 15: You will automatically lose at least 40 points if there are compilation errors or warning messages when we compile your source code for both phases. You will lose points if you don't use the specific program file name, or don't have comments, or don't have a comment block on EVERY program you hand in. These will be deducted from your final score, after accounting all other requirements listed above.

II. Programming environment:

Write your program in C++. Compile and run it using AU server (no matter what kind of text editor, IDE or coding environment you use, please make sure your code could run on AU server, the only test bed we accept is the AU server).

III. Deliverables:

You must submit a tarred and compressed file – called a tarball, which contains all your .cpp files, header files if you have any, all testcases, and your `.git` folder. Assuming your repo root is

`~/comp2710/project2_King_pzk0039`, use these commands below to create a compressed file for submission:

```
cd ~/comp2710
tar vfzc project2_King_pzk0039.tgz project2_King_pzk0039
```

Note 16: Again, make sure you have all necessary files, including your `.git` folder. You will lose a big chunk of points if you forget to add any of them to your submission.

IV. Late submission penalty:

- Late submissions are not accepted and will results in a **ZERO** without valid excuses, in which case you should talk to Dr. Li to explain your situation.
- GTA/Instructor will not accept any late submission caused by internet latency.

V. Rebuttal period:

- You will be given a period of **2 business days** to read and respond to the comments and grades of your homework or project assignments. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.

Good luck y'all! And WAR EAGLE!