113 Project Report
By: Ayan Das

- **Project Topic Introduction:**

The purpose of this project was to implement two features. The first feature was to implement an algorithm which will read a given text file and output the frequency of the word, as in the number of times they appear throughout the text file. The second feature was to implement either multithreaders or multiprocessors to determine the execution speed. We were provided with various resources and references through class notes to understand the concept of multithreading/multiprocessing and various scientific libraries used in data analysis and visualization. This project was completed in a group of three with roles split amongst each student.

- **Design of the program:**

The collaboration for implementing the code and researching methods to implement the two features was done primarily through "Google Collab". However, outside of google collab, the primary form of IDE used for the project was "Jupyter notebook" provided in the "Anaconda Navigator". For all of us, it was a completely new software but the learning curve was fairly simple. Google collab is simply a cloud form of Jupyter Notebook, with added features and less download requirements.

As stated earlier, the project is split into two parts:

a. **Report the word frequent analysis in a text file:**

The project requirement asked us to specifically use a text file. The source of the textfile is the following link: https://www.gutenberg.org/ebooks/69079. The link contains the specific text file we used for the project. Using the text file, we implemented a function named: "WordFrequency" which will take in one parameter to calculate the frequency of the words throughout the text. The data structure we decided best suitable to serve to represent the word frequency would be a dictionary. With the keys being the word and the values representing the number of times each specific word appears throughout the text. However, prior to implementing the function, we needed to convert the text file into readable python data, therefore, we converted the text file into a string data type and stored it into a variable.

b. Report the word count-time using multiple threads/processes from 1-8 (My Contribution)

The first part of the project was completed primarily through my group members, they implemented the function to calculate the word frequency, although I did instruct them on how the function implementation would be done and which library would be best suitable for calculating the word analysis in the text. As for my part, using Nasif and Adeeb, my two group member's code as reference for the threading, I implemented thread 5-8 and plotted the visualization for threads 2-8 using the matplotlib graph. The runtime execution was pushed onto empty lists so I have the data to plot in the graph, with the x-axis representing the number of threads and y-axis representing the runtime. For the control flow structure for the threading function, it was primarily a set of sequential instructions, the instructions executed one after another. For the functions implemented to calculate the general word frequency and highest word frequency, the "wordFrequency" function utilized a for loop repetition, "highestWordFrequency" utilized a combination of selector control flow statement such as "if" as well as repetition focused statement such as "for" to calculate the word that appeared the most. The data structure we used to store the list of words and their frequency was a dictionary, also known as a "hashtable" and to store the runtime execution we used a list. Which served as data on the y-axis for graphing the runtime visualization based on the number of threads, also known as "workers".

Based on my observation of executing the threads, what I have noticed is that while the general trend of each passing thread, the runtime execution speed decreases, the overall time it takes with more threads for the program to execute increases. This means while the last thread execution out of 8 threads may run the fastest, the overall time from start to finish for the execution of 8 threads is higher in comparison to a single thread, double thread and even seven threads. The distinguishing feature between single threads and multi threads is that the dictionary size was split into smaller sizes as the number of threads increased, this was achievable using the **islice()** function found in the **itertools** library, similar to how a string slicing method, it splits the dictionary into various smaller pieces, which then serves as values for the "args" parameter for the "threading.thread" function. For example, executing a program with 3 threads means splitting the dictionary into 3 parts, if the length of the dictionary is evenly divisible by 3, the dictionary will be split into 3 equal parts, if the dictionary isn't evenly divisible, the first two part will be equal, but the third part will be smaller in length in comparison to the other two, a smaller dictionary size for the thread to work with will also lead to a faster execution time, since there's less data to parse through.

As for the visualization of the runtime, given the function on the notes, I ran into various errors, so I implemented my own visualization function to visualize the thread execution speed and they all follow the similar pattern of decreasing with each passing thread.

Lastly, my last contribution was writing up the readme file on the github account with instructions on how to run the program, a lot of the information given in the report is an extensive form of the instructions written in the Github ReadMe file for the shared repository. I learned several things working on this project, such as various kinds of dictionaries, methods of splitting a dictionary beyond the conventional means, how multithreading works, how it can be used to optimize programs, how to read text file data using python and apply it, as well as various data visualization techniques available using matplotlib.

An extension of this project can be to review and modify our code so as to make it more scalable. At the present, the code only calculates the word frequency of a single text file which we imported and converted to a string. But we can extend the function in such a way that we can traverse multiple text files and output separately their own wordFrequency, we can also check to see how such a function affects the runtime execution speed for multi threaders.