113 Project Report
Nasif Rahman

- **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.

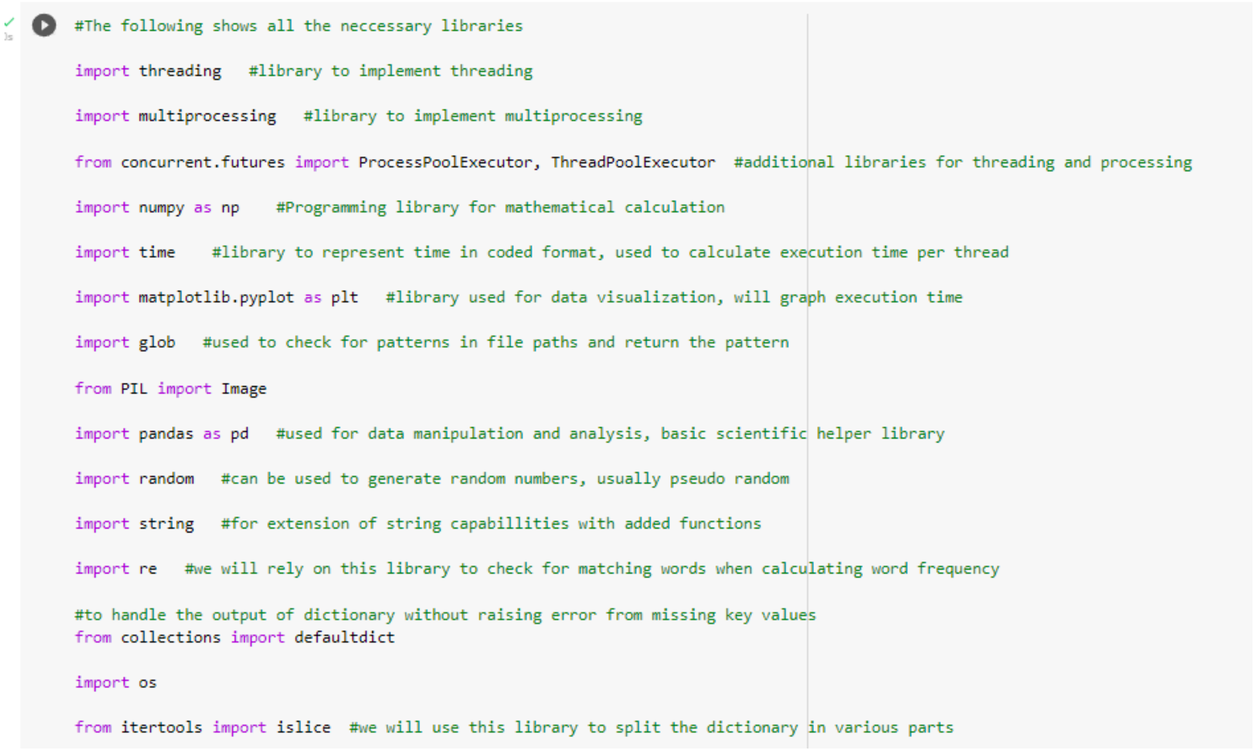
- **Design of the program:**

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:

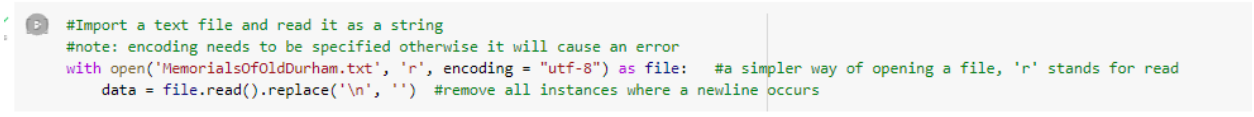
a. **Report the word frequent analysis in a text file:**

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. The following code snippet shows how this action was done:



```
#The following shows all the neccessary libraries

import threading   #library to implement threading

import multiprocessing   #library to implement multiprocessing

from concurrent.futures import ProcessPoolExecutor, ThreadPoolExecutor  #additional libraries for threading and processing

import numpy as np    #Programming library for mathematical calculation

import time    #library to represent time in coded format, used to calculate execution time per thread

import matplotlib.pyplot as plt   #library used for data visualization, will graph execution time

import glob   #used to check for patterns in file paths and return the pattern

from PIL import Image

import pandas as pd   #used for data manipulation and analysis, basic scientific helper library

import random   #can be used to generate random numbers, usually pseudo random

import string   #for extension of string capabillities with added functions

import re   #we will rely on this library to check for matching words when calculating word frequency

#to handle the output of dictionary without raising error from missing key values
from collections import defaultdict

import os

from itertools import islice  #we will use this library to split the dictionary in various parts
```

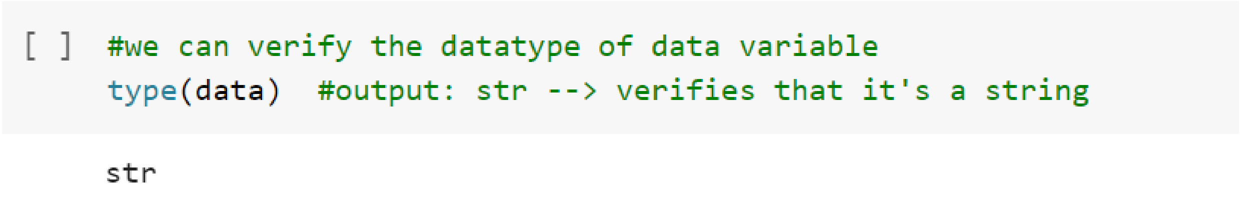
Fig 1: *Shows the list of libraries used to implement the code*



```
#Import a text file and read it as a string
#note: encoding needs to be specified otherwise it will cause an error
with open('MemorialsOfOldDurham.txt', 'r', encoding = "utf-8") as file:   #a simpler way of opening a file, 'r' stands for read
    data = file.read().replace('\n', '')  #remove all instances where a newline occurs
```

Fig 2: *Python Code snippet with additional comments explaining how to read a file and store the file in a variable*

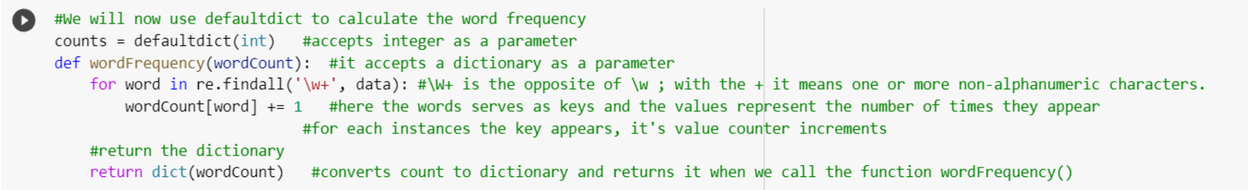
In this case, we replace all instances of newline with an empty string, to remove line breaks and storing the converted text file into the string variable named "data".



```
[ ]  #we can verify the datatype of data variable
     type(data)  #output: str --> verifies that it's a string

     str
```

Fig 3: *The following code verifies that variable data is of type string, comments have been added for explanation*

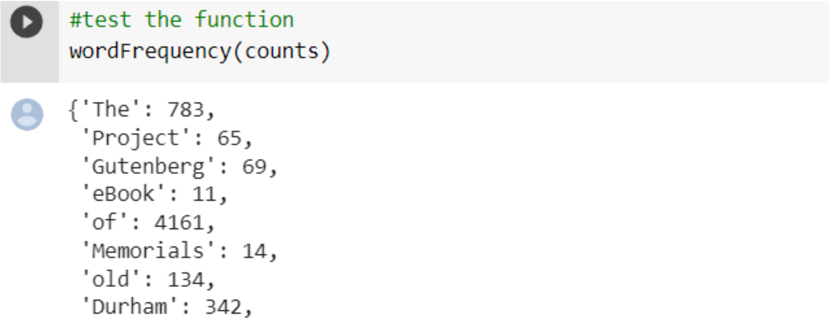
The **type()** function in Python returns the variable type. Here, we are using the **type()** function to verify that our data is of string (str) type.



```python
#We will now use defaultdict to calculate the word frequency
counts = defaultdict(int)   #accepts integer as a parameter
def wordFrequency(wordCount):  #it accepts a dictionary as a parameter
    for word in re.findall('\w+', data): #\W+ is the opposite of \w ; with the + it means one or more non-alphanumeric characters.
        wordCount[word] += 1   #here the words serves as keys and the values represent the number of times they appear
                            #for each instances the key appears, it's value counter increments
    #return the dictionary
    return dict(wordCount)   #converts count to dictionary and returns it when we call the function wordFrequency()
```

Fig 4: *Function definition for wordFrequency to calculate the word frequency in data*

We needed a specific kind of dictionary for this function. “DefaultDict” helps us handle missing key values without raising any errors. Whereas, if we stored the result in a normal dictionary, it will cause the function to raise an error. The “wordFrequency” function takes a parameter, and the parameter it expects is a “defaultdict” type dictionary. The “re.findall” function is used to traverse through the string data, and checks for all matching patterns, in this case, all the unique words and how many times they appear throughout the text. And for each instance of word, which serves as the key, we increment it by 1 for each matching word found in the string variable data.

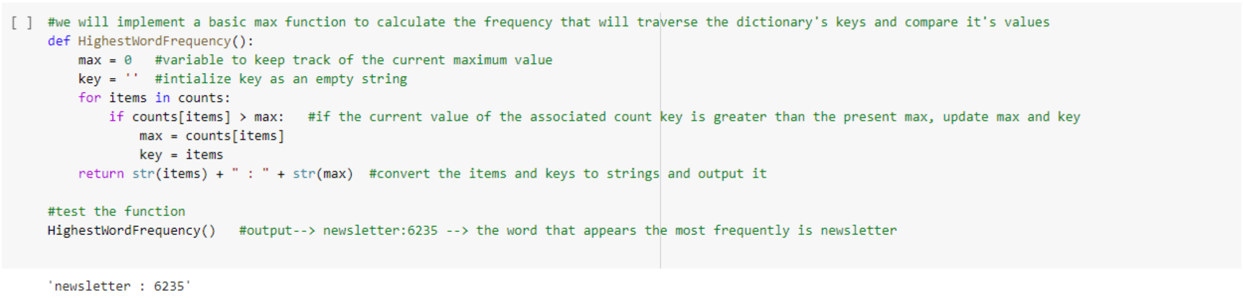


```python
#test the function
wordFrequency(counts)
```

```
{'The': 783,
 'Project': 65,
 'Gutenberg': 69,
 'eBook': 11,
 'of': 4161,
 'Memorials': 14,
 'old': 134,
 'Durham': 342,
```

Fig 5: *Code to test the wordFrequency function*

Now that we have defined the wordFrequency**()** function, we test it by passing the dictionary **counts** as a parameter. If we look at the output, we see the function returns the words along with their frequency in the text file (which, now, is stored in data). For example, the word “The” appears 783 times in the file, the word “Project” appears 65 times, and so on. This shows that the wordFrequency() function is working as intended. Note that in the figure, we only included several word frequencies for explanation purposes. The code file contains all the word frequencies as per the project requirement.

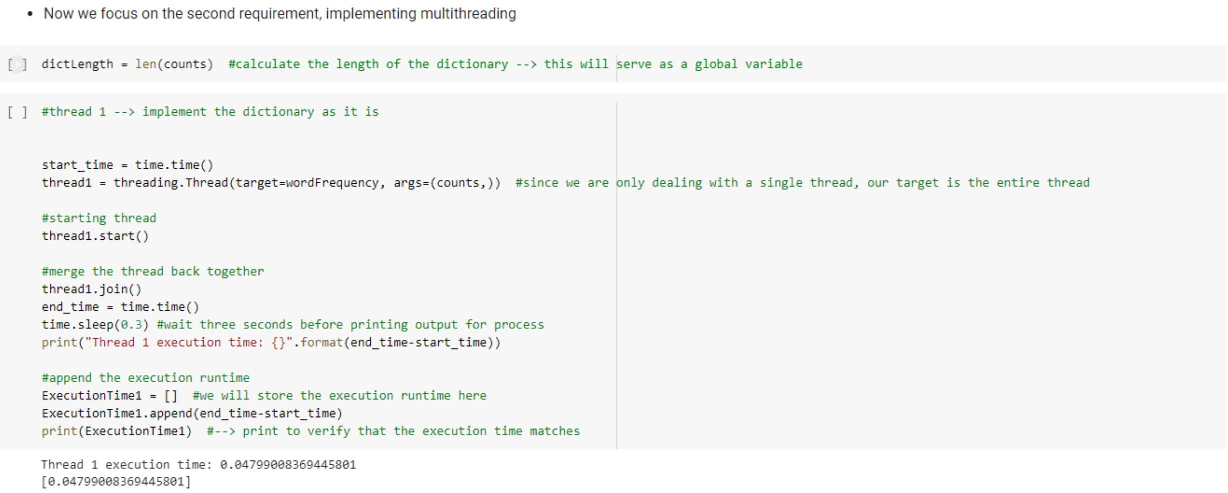


```
[ ] #we will implement a basic max function to calculate the frequency that will traverse the dictionary's keys and compare it's values
    def HighestWordFrequency():
        max = 0   #variable to keep track of the current maximum value
        key = ''  #intialize key as an empty string
        for items in counts:
            if counts[items] > max:   #if the current value of the associated count key is greater than the present max, update max and key
                max = counts[items]
                key = items
        return str(items) + " : " + str(max)  #convert the items and keys to strings and output it

    #test the function
    HighestWordFrequency()   #output--> newsletter:6235 --> the word that appears the most frequently is newsletter

    'newsletter : 6235'
```

Fig 6: *Function used to check the most frequent word*

The function above is used to check for the most frequent word, as in the word that appears the most throughout the text file. As we can see based on the output, the most frequent word that appears is 'newsletter', for a total of 6235 times throughout the text.

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



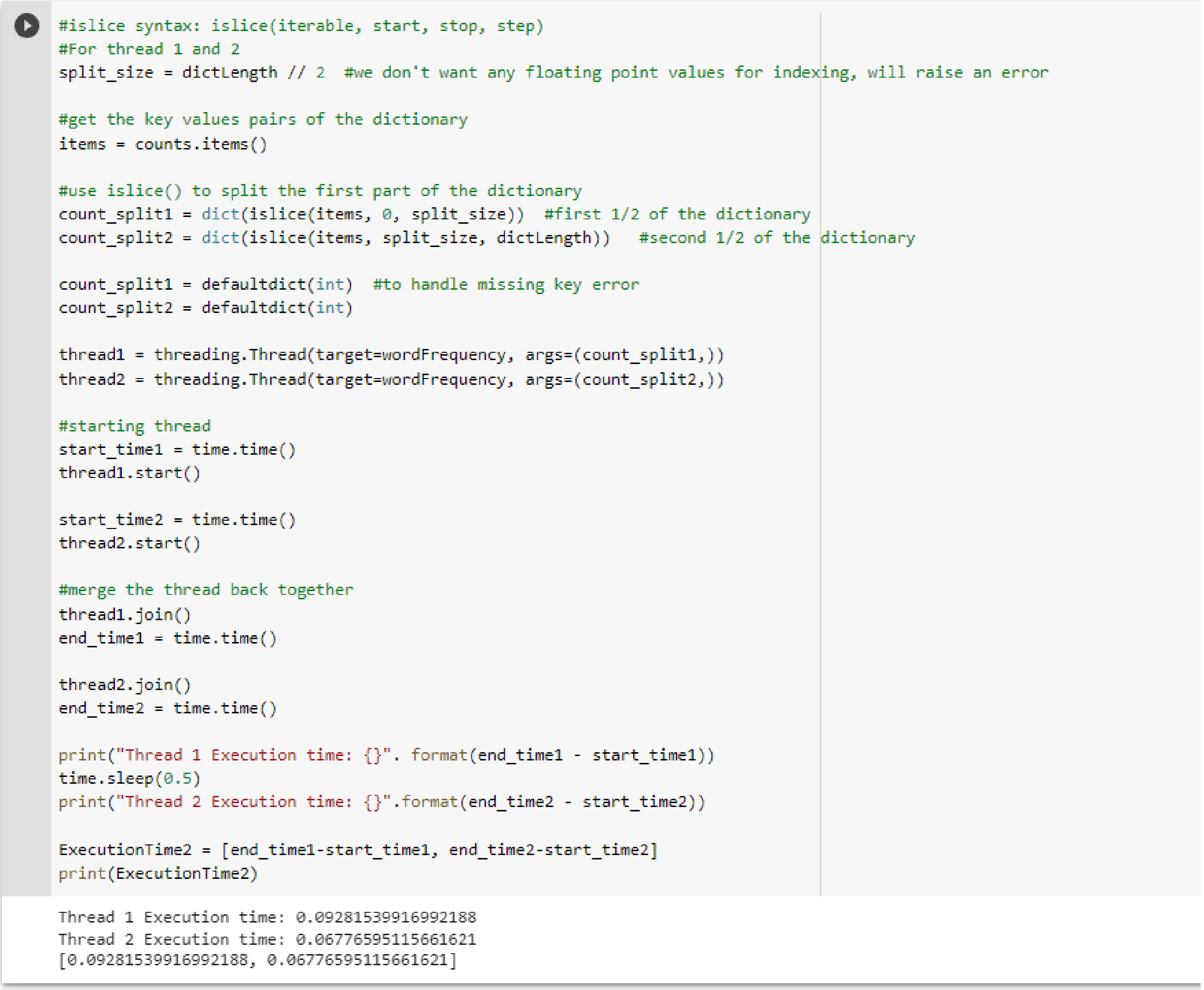
- Now we focus on the second requirement, implementing multithreading

```
[ ] dictLength = len(counts)  #calculate the length of the dictionary --> this will serve as a global variable
```

```
[ ] #thread 1 --> implement the dictionary as it is


    start_time = time.time()
    thread1 = threading.Thread(target=wordFrequency, args=(counts,))  #since we are only dealing with a single thread, our target is the entire thread

    #starting thread
    thread1.start()

    #merge the thread back together
    thread1.join()
    end_time = time.time()
    time.sleep(0.3) #wait three seconds before printing output for process
    print("Thread 1 execution time: {}".format(end_time-start_time))

    #append the execution runtime
    ExecutionTime1 = []  #we will store the execution runtime here
    ExecutionTime1.append(end_time-start_time)
    print(ExecutionTime1)  #--> print to verify that the execution time matches

    Thread 1 execution time: 0.04799008369445801
    [0.04799008369445801]
```

Fig 7: *The second part of the function requirement is to implement either multithreading or multiprocessing, for our project, we implemented multithreading.*

The multithreading function provided to us in the notes varies from the multithreading function we implemented. We made some modifications to the function because when we ran it as instructed, we ran into various errors. Therefore, we implemented our own multithreading function and visualization function. The code above shows single thread implementation, the runtime execution from start to finish is also recorded and shown.

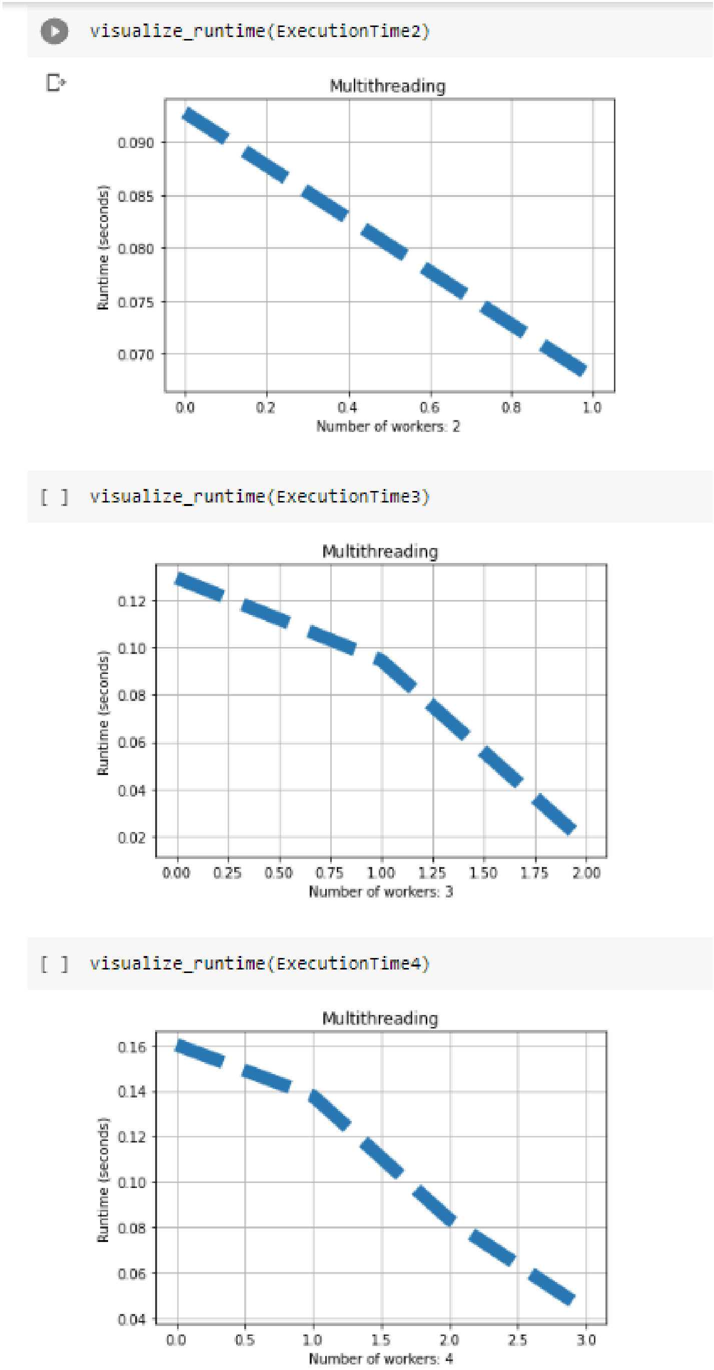


```
#islice syntax: islice(iterable, start, stop, step)
#For thread 1 and 2
split_size = dictLength // 2  #we don't want any floating point values for indexing, will raise an error

#get the key values pairs of the dictionary
items = counts.items()

#use islice() to split the first part of the dictionary
count_split1 = dict(islice(items, 0, split_size))  #first 1/2 of the dictionary
count_split2 = dict(islice(items, split_size, dictLength))   #second 1/2 of the dictionary

count_split1 = defaultdict(int)  #to handle missing key error
count_split2 = defaultdict(int)

thread1 = threading.Thread(target=wordFrequency, args=(count_split1,))
thread2 = threading.Thread(target=wordFrequency, args=(count_split2,))

#starting thread
start_time1 = time.time()
thread1.start()

start_time2 = time.time()
thread2.start()

#merge the thread back together
thread1.join()
end_time1 = time.time()

thread2.join()
end_time2 = time.time()

print("Thread 1 Execution time: {}". format(end_time1 - start_time1))
time.sleep(0.5)
print("Thread 2 Execution time: {}".format(end_time2 - start_time2))

ExecutionTime2 = [end_time1-start_time1, end_time2-start_time2]
print(ExecutionTime2)
```

```
Thread 1 Execution time: 0.09281539916992188
Thread 2 Execution time: 0.06776595115661621
[0.09281539916992188, 0.06776595115661621]
```

Fig 8: *Shows the implementation for 2 threads*

The difference between single thread and multi thread is that for the dictionary which we used to store the word frequency, the word being the keys and the values showing the number of times they appear, for multi threads, we needed to split the dictionary into various components. The number of splits made in the dictionary was directly proportional to the number of threads being implemented. So, for 2 threads, the dictionary was split in half and the splitted dictionary was pushed into the threads, with the same target function: wordFrequency. For 3 threads, the dictionary was split by 1/3rd, for 4 threads, the dictionary was split by 1/4th. This was achievable using the “islice” function found in the “itertools” library. The primary pattern common among the threads is that, with each passing thread, the runtime decreases, meaning the process speeds up as the number of threads increases. Although the image above only shows the implementation of 2 threads, the remaining threads were implemented in the same manner, the only thing different was how the dictionary was split, but the execution of the threads remains the same, as the number of threads increased, the runtime decreased. The execution time was pushed into an array, using which we will visualize the runtime.



```
visualize_runtime(ExecutionTime2)
```

```
[ ] visualize_runtime(ExecutionTime3)
```

```
[ ] visualize_runtime(ExecutionTime4)
```

Fig 9: *Graph showing the runtime execution speed for threads 2,3 and 4*

As observable based on the graph shown, as the number of threads increases, the runtime in seconds decreases. This trend can be seen amongst all the multi threaders, with each passing thread execution, the runtime lowers.

Contribution to the project:
For our project, I imported the text from the txt file into our code, implemented the wordFrequency function, and implemented threads 3 and 4. In my video, I explained how we import texts from the txt file and store them in a string variable. Then I explained the implementation of the wordFrequency function where we count the frequency of each word that appears in the text file (now in the string variable), and how they are stored in a dictionary. Finally, I explained the implementation and execution time of threads 3 and 4. I also contributed to the project report by explaining wordFrequency, and how it outputs a dictionary containing words and their frequency.

By working on this project, I learned how to deal with large amounts of data (text in this case), and use a dictionary to define functions in an easier, convenient way. I also learned how to implement Multithreading to speed up the execution time of a program.