

Sohail Ahmad

CSC 113 | Programming Language

Professor Yuan

## **Final Code Report**

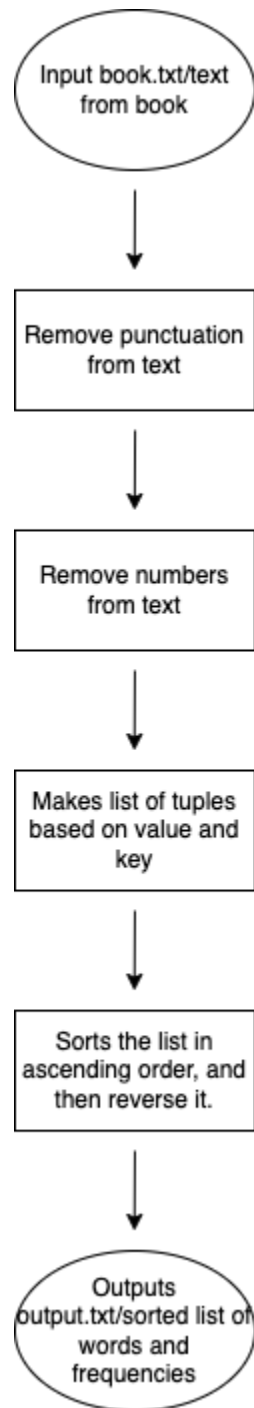
### **1. Project topic introduction:**

Throughout the semester we were introduced to new ideas and concepts used in programming languages, for our case specifically in Python 3. In order to showcase our ability with the use of Python and convey our understanding of the concepts, we were given a few options to work on a semester-long project. My group and I came to the consensus of working on the “Word Frequency Project” which entailed implementing various functions that required an understanding of concepts such as loops, functions, classes, libraries, multiprocessing, race conditions, and memory management. Our goal for this project was to use the Gutenberg project website to choose a text file to use as a sample for our dataset, and use our script to perform word frequency analysis on the text, generating an output file that would list the most frequent words and how many times they appeared. This then needed to have multiprocessing support with functions to measure the time taken for the execution of a various number of processes.

### **2. 2. how you design the program. You can use the control flow figures to describe the entire process from given input to output**

When given this project, it was difficult to get started with it straight away due to the complexity of the program. This meant that we needed to use the divide and conquer technique to accomplish this task. We simply planned out the different parts of the program in our HLD report, then we made use of the LLD report to list out the specific functions we would need. We

designed our program with simply the necessary functions, adding and building upon each iteration of progress.



The script begins by importing the **string**, **multiprocessing**, and **time** modules. The **string** module is used to access a string of all ASCII punctuation characters, and the

multiprocessing module is used to create and manage processes in the script. The time module is used to measure the execution time of the script. Next, the program declares the **filepath** for the 'book.txt' and **out\_filepath** for 'output.txt' variables, which hold the paths to the input file and the output file, respectively. The input file is a text file containing a book, and the output file will be used to store the sorted list of words and their frequencies in the input file.

The script defines four functions: **remove\_punctuations**, **remove\_numbers**, **ordered\_dict\_by\_freq**, and **process\_line**. The **remove\_punctuations** function takes a line of text as input and removes all punctuation characters from it. It does this by iterating over the characters in the **string.punctuation** string and replacing them with an empty string in the input line. The **remove\_numbers** function takes a line of text as input and removes all numeric characters from it. It does this by using a generator expression to create a new string that contains only the non-numeric characters from the input line. The **ordered\_dict\_by\_freq** function takes a dictionary as input and returns a list of tuples, each containing the value and key of the dictionary sorted in descending order by the value. It does this by iterating over the keys of the dictionary and appending the values and keys as tuples to a list, then sorting the list in ascending order by the values. Finally, it reverses the list to get the desired descending order. The **process\_line** function takes a line of text, a shared dictionary, and a lock as input and processes the line by removing punctuation and numbers, splitting it into words, and updating the shared dictionary with the counts of the words. It does this by calling the **remove\_punctuations** and **remove\_numbers** functions on the input line, then splitting the line into words and iterating over the words. For each word, it converts it to lowercase and increments its count in the shared dictionary.

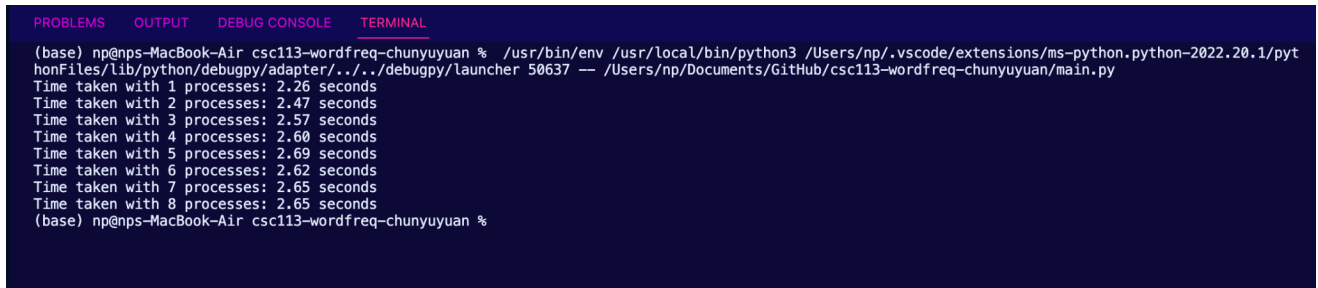
The script also defines the **measure\_time** function, which takes a number of processes as input and measures the time taken to execute the script using that number of processes. It does this by creating a manager and a shared dictionary using the manager, creating a lock to synchronize access to the shared dictionary, recording the start time, creating a pool of worker processes, and using the worker processes to process the lines in the input file in parallel. It then records the end time, sorts the shared dictionary by frequency using the **ordered\_dict\_by\_freq** function, writes the sorted dictionary to the output file, and returns the time taken to execute the script. Finally, the program calls the **measure\_time** function four times with different numbers of processes, and prints the time that is taken to execute the script using each number of processes. This allows the user to compare the execution times and see the effect of using different numbers of processes on the overall performance of the program.

Overall, this program demonstrates how to use the multiprocessing module in Python to parallelize the processing of a large input file, and how to use a lock to synchronize access to a shared dictionary. It also shows how to measure the execution time of a script and compare the performance of different numbers of processes.

**3. Results analysis: you will run your program with the different number of processes/threads 1-8 and record the execution time. Please discuss what you find from the time results. The greater number of processes/threads, the faster your program's execution speed.**

When looking at the results from our program, we observe almost no significant difference in the time taken to execute the program. It is my understanding that this is due to us using “lock” on the processes, as the dictionary is using shared memory between the processes we can run into race conditions happening if we don’t use locks on the processes. With the use of

locks we are able to prevent race conditions and get consistent results each time but seemingly can have an impact on the speed because the processes need to wait for the current process. The times generated are shown below.

A screenshot of a terminal window with a dark blue background and white text. The terminal shows the output of a Python script that measures the time taken to execute a task using different numbers of processes. The output is as follows:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
(base) np@nps-MacBook-Air csc113-wordfreq-chunyuyuan % /usr/bin/env /usr/local/bin/python3 /Users/np/.vscode/extensions/ms-python.python-2022.20.1/pyt
honFiles/lib/python/debugpy/adapter/../../debugpy/launcher 50637 -- /Users/np/Documents/GitHub/csc113-wordfreq-chunyuyuan/main.py
Time taken with 1 processes: 2.26 seconds
Time taken with 2 processes: 2.47 seconds
Time taken with 3 processes: 2.57 seconds
Time taken with 4 processes: 2.60 seconds
Time taken with 5 processes: 2.69 seconds
Time taken with 6 processes: 2.62 seconds
Time taken with 7 processes: 2.65 seconds
Time taken with 8 processes: 2.65 seconds
(base) np@nps-MacBook-Air csc113-wordfreq-chunyuyuan %
```

It is generally expected that using multiple processes can speed up the execution of a program, but this is only sometimes the case. There are several factors that can affect the performance of a program when using multiple processes, and the actual performance improvement will depend on the specific characteristics of the program and the hardware it is running on. One reason why the runtime may be slower when using multiple processes is the overhead associated with creating and managing the processes. Creating and managing processes requires extra work by the operating system and the CPU, which can add some overhead to the execution of the program. In this case, using multiple processes may not provide much of a performance improvement.

#### **4. your contribution to the project (if you are in a group) and what you learn**

I worked on various parts of the project, I worked on helping implement and troubleshoot the basic functions to get the word frequency to output into a text file, and also worked on adding multiprocessing support for the program. Throughout the semester I helped to complete the code so that we can stay on track for the deadline, helping with anything that was necessary. Besides helping implement the program I worked to help complete this report and also edit the video presentation for this project.