CSCI 1320 Computer Science I: Engineering Applications – Fall 2018
Instructor: Zagrodzki
Project: Due Sunday, Dec 9, by 6pm

Natural Language Text Analysis

There are areas within computer science that aim to understand how we use human languages, typically referred to as natural languages. Given a sample of text, can we design a computer algorithm that will suggest who wrote it? Analysing vast volumes of English language literature spanning a certain time range, what sort of vocabulary became more common and what became obsolete? Can we track the evolution of a language and how it traversed the earth's geography? With the aid of today's computational power as well as vast libraries of digital data, unprecedented advances are being made in pursuit of answers to these types of questions. In this assignment we will do a basic introduction to text analysis by reading in text data, creating a data subset for testing, cleaning, finding the most commonly occuring words, and sorting.

1. Reading in text, counting the number of words, creating test data

You are given a text file containing the full text to the book Hunger Games, Book 1. The file has been pre-processed to remove all punctuation and down-case all characters.

A. Scan the entire file and count the total number of words. Display the total number of words to terminal.
B. Write the first 100 words out to a new file. Call it "testText.txt".

2. Cleaning

In ignoreWords.txt you are given the 50 most commonly occuring words in the English language (separated by whitespaces). Write a program (you will need a new cpp file here) that creates a new text file excluding all occurrences of any of these words from a given input text file. Test your program on the test file you created in part 1 (100 words) and call it something like cleanedTextTest.txt.

If your algorithm appears to be working correctly, run it on the full Hunger Games file (use the number of words you found in part 1A and call it something like cleanedText.txt.

(Since we have not introduced techniques for creating variable length arrays, you should manually declare your arrays to a fixed length. Your array length(s) should be set to a specified length *within* the cpp file, as we cannot change the length of standard arrays during program execution time.)

3. Most commonly occuring

Read in the clean text you generated in part 2 (start a new cpp file). Create a list of all the unique words found in the entire text file (use cleanedTextTest.txt for testing). Your list should be in the form of an array of structs, where each struct has two members. One member contains the word, one member contains the number of occurrences of that word. The struct definition should look like this:

```
struct UniqueWord
{
   string word;
   int noOccurances;
};
```

Sort the array in ascending order. Display to the console the number of unique words, the 10 most frequently occurring, and 10 least frequently occurring.

Again, since we have not yet covered techniques for "growing" arrays in C++, preallocate your array to some large number. To play it safe, this could mean the number of words in the entire input file (this is assuming worst case scenario, where every word in your input file is unique).

Hint: Consider the algorithm for this before starting to write code. Start by writing pseudocode. You need to think about looking at every word in the input file and comparing it against a list of unique words.

4. Frequently Paired Words

Now extend the code you wrote in Part 3 to look for commonly paired words. You should create a new, seperate file for this task. Create a list of all word pairs, and count the number of times each pair occurs. You will need a new struct definition to store both words, and you will need to keep track of two words at once while moving through the text. You will not need to worry about word order for this task; that is, you should interpret [eat fish] and [fish eat] as separate entities.

Sort the array in ascending order. Display to the console the number of unique word pairs, the 10 most frequently occurring, and 10 least frequently occurring. Run this on both the cleaned text and the original text. How different are they?

Hints: This task is much more similar to task 3 than it may appear. Depending on your implementation and how much of the code is split into independent functions, some parts may work with very little (if any) modification. Remember to check your edge cases!

Extra Credit
Implement task 4 using a 2D array to keep track of occurrence counts instead of a 3 entity struct to improve performance. Add timers to your code to measure the improvement. Index the 2D array with both sorted and unsorted word lists to measure the change in performance. You should have access to both of these already from task 3.

Sorted WL Example	[CommonWord]	[UncommonWord]	[RareWord]	...
[CommonWord]	54	34	22	
[UncommonWord]	34	43	23	
[RareWord]	12	4	2	
...				

Check that your code is correct by displaying to the console the 10 most unique and 10 most common word pairs, and verifying that they are the same as those in part 4.

Present your results in a table:

	2D Array (Sorted)	2D Array (Unsorted)	Struct
Test File	X1 Seconds	Y1 Seconds	Z1 Seconds
Whole Text	X2 Seconds	Y2 Seconds	Z2 Seconds

You probably noticed that we do not ask for specific functions, cpp files, driver programs, or what the form of your submission should look like. This is where you should use your best judgement in applying the techniques learned in earlier assignments to make your coding as effective as possible. Not much sympathy will be given in office hours to students who come in with spaghetti code and no driver programs for testing their functions ;) Good luck!

Submission:

Zip all your cpp files and name the zip file as lastnameFirstnameProject.zip, and upload it to Moodle by the due date.