**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #1**

Due: September 8th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this Lab we were tasked with creating a program that ask a user to input a word and then recursively finds and prints all anagrams of that word in alphabetical order. For this lab it is essential that we are familiar with the basics of recursion as it is the main method that will use.

**Proposed Solution Design and Implementation**

**Operation #1:**

First, I created a main method to start our program and then added a function that would read and store the contents of the words\_alpha file. As well as striped each line so only the remaining word would be stored in the set. Next, I would call a method that would ask the user for an input. If the user inputted a word that was valid the program would continue otherwise it would tell the user, the input was invalid or exit the loop if the user inputted nothing. Then I used a modified version of the code given to us in section 2.6 of our zyBooks homework. Within the method I added three additional parameters to compare the permutations of the original word to as well as a set to store the permutations found. The method then takes a sorted version of the user inputted word and creates all premutation of the letters given recursively by using a for loop to scramble all the letters after the initial one. Creating n! premutation where n is the length of the word. After there are no more words for the method to scramble the word is then added to a temporary set and intersected with the set created from the words\_alpha file. Then as long as the intersection returns one value the permutation is added to the set for found anagrams. After all the anagrams are collected and the method finishes the set of anagrams is stored in a new list which is then sorted in alphabetical order and printed out to the user along with the time to the sixth decimal point it took. **The Main idea of the method is still** the same as the zyBooks method scramble(r\_letters, s\_letters) but is just modified to return any matching results.

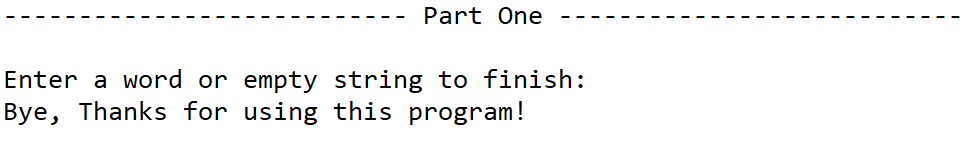
**Operation #2:**

For this operation, I used the same set created in the main method as well as copies of the two additional methods created to ask for a user inputted word and create all permutations of the word inputted but slightly modified them. First, I added a new set to store duplicate variables. While the method recursively creates all the permutations of the user inputted word it would also check if the letter being used has been used already and if so, would stop the recursive call so the duplicate permutations would not be created. Next, I created a method that would return all prefixes of an inputted word and created a for loop to create prefixes for all the words for the words\_alpha file and store them in a new set. I, then added that new set of prefixes to the parameters if the recursive method and after the recursive call would check for the duplicate letters it would then check to see if the prefix was in the new set of prefixes. If so, the program would continue until the premutation utilizes all the letters of the word and then would check if the word is in the original set. Otherwise it would also stop the recursive call from continuing as there is no need for it to.

**Experimental Results**

**Operation #1:**

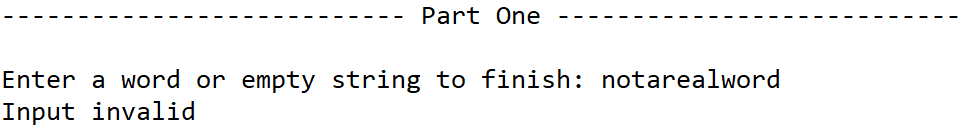
For this operation I decided to test with an empty string, a made-up word, a word of my choice and the word permutation. Allowing me to test the edge case if the user wishes to exit the program. Then the to see if the program functions as intended, and lastly to see how the program handles large words.



Case 1 (Empty String):

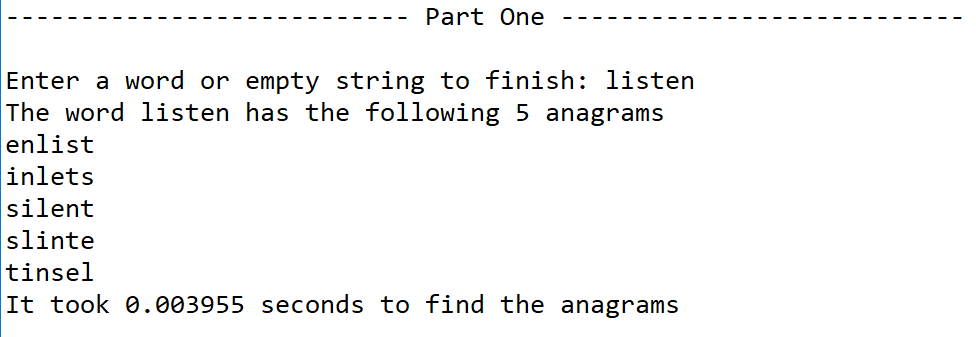
Anagrams found: None

Time: None

Case 2 (‘notarealword”):

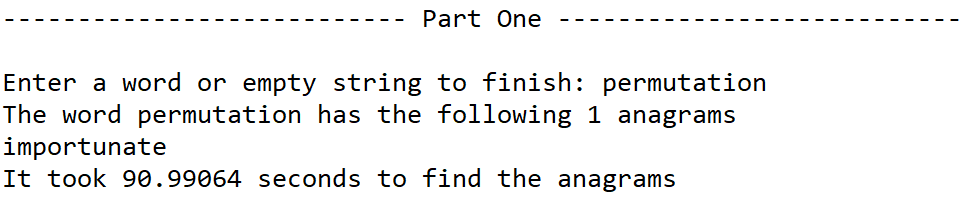
Anagrams found: None

Time: None

Case 3 (“listen”):

Anagrams found: 5

Time: 0.003955s

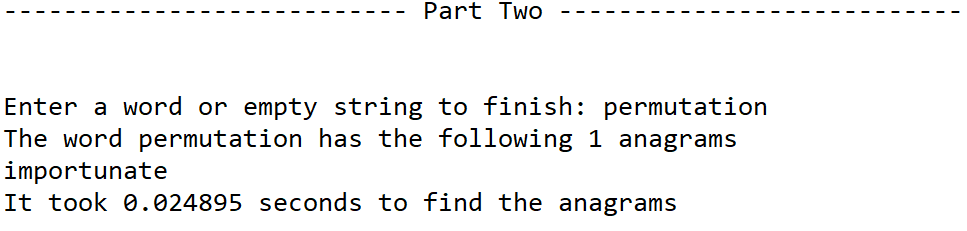
Case 4 (“premutation”):

Anagrams found: 1

Time: 90.9906s

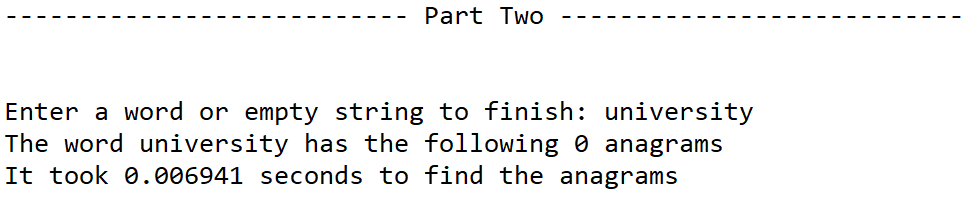
**Operation #2:**

For this operation I tested long words as the program originally struggled with them

Case 1(“permutation”):

Anagrams found: 1

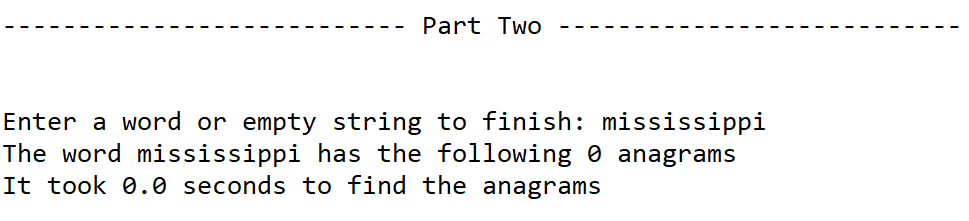
Time: 0.024895s



Case 2(“university”):

Anagrams found: 0

Time: 0.006941s

Case 3(“mississippi”):

Anagrams found: 0

Time: 0.0s

|  |  |  |
| --- | --- | --- |
| Word Length | Normal Method Time | Optimized Method Time |
| “list” | 0.0 Seconds | 0.0 Seconds |
| “towering” | 0.085766 Seconds | 0.002993 Seconds |
| “university” | 8.085670 Seconds | 0.004986 Seconds |
| “jeopardizing” | 1175.127 Seconds | 0.017951 Seconds |

**Graphs of Time Differences**

As the result of the chart shows, the first operation created handles words up to 6 letters well but begins to take longer after that as it generates all permutations including repeated ones recursively. Making the optimized method better as it not only allows the program to detect duplicate permutations and stop the recursive call but also stops the recursive calls of prefixes that are not common between the main set of prefixes.

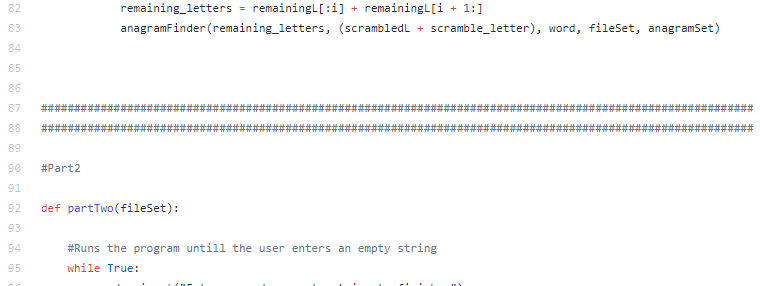
**Conclusion**

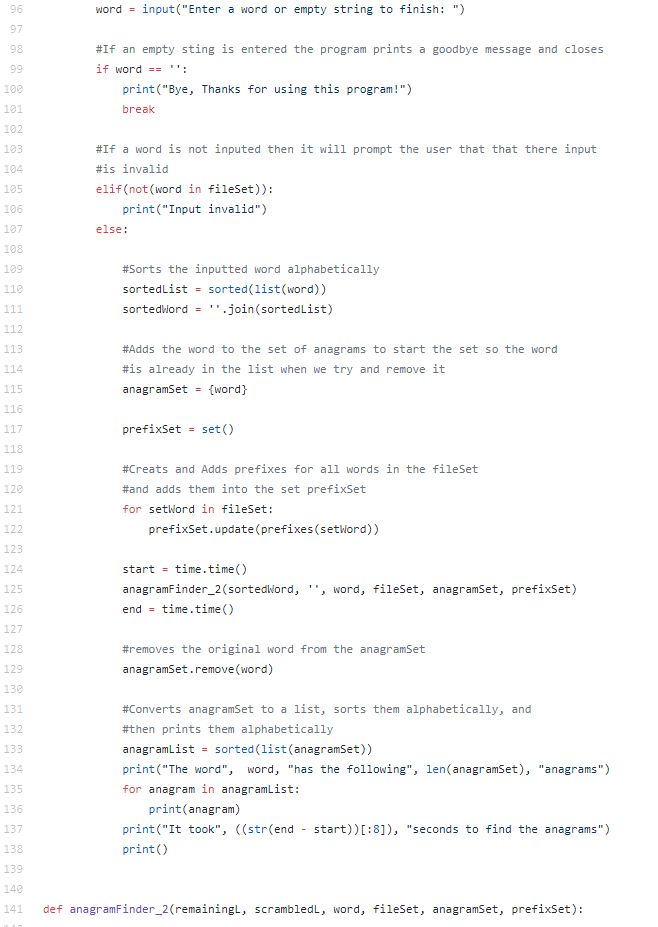
In conclusion this lab helped me better understand how to optimize recursive functions to better run the task at hand. I was able to clearly understand how making the method recursively both increased functionality and performance of the method. On top of that after optimizing the first method the function just kept improving. As if I were to follow the method reclusively it would show a major improvement in the number of recursive calls as duplicate permutations and unmatching prefixes would be stopped.

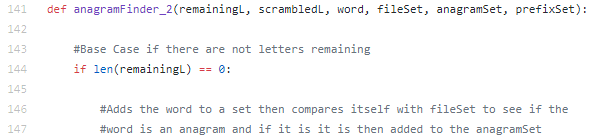
**Appendix**



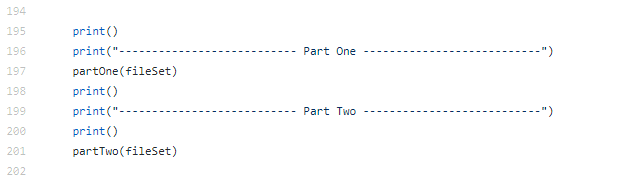












I Issac Rivas, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, preformed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any satudent in the class.