Lab 1 Report

Sofia Gutierrez

Olac Fuentes

CS 2302

Introduction

In this program, the objective was to create a list of anagrams of a specific word, which was attained from user input. The program then read a list of words off a downloaded text and stored them into a set. There were two methods to implement in order to optimize the code. The first, was to create a method to dismiss any anagram duplicates. More specifically it would disregard the second character if it presented itself twice. The second method was created to store the prefixes of all the words from the word set into a second set. This would speed up the process of creating the anagrams because it would first check to see if the partial word was a prefix of any word in the word set. Both of these methods were called in the method that created the anagrams. If the word did not contain a duplicate and it was found in the set containing all the prefixes then the anagrams was created.

Design and Implementation

The first step would be to take in the user’s input word. This word was immediately tested to see if it was actually a valid string, meaning that it had to at least contain one element. The next method was created to produce the anagrams. But, before I could define the method I inserted a variable that would start the time the method to create the anagrams took to run. I printed the time outcome in the main. Within this method I passed the user input which was named r\_letters (remainding letters) and an empty string named s\_letters (scrambled letters). I inserted the first if statement which again declared the obvious base case that stated if the length of the r\_letters equals to 0 then it would print the s\_letters. The else, began with a for loop which was to iterate through r\_letters. Within the for loop I created a second variable to store the scrambled letters (scrambled\_letters) at index i, i being what is being iterated. Another variable was created named remaining\_letters which was set to r\_letters at the ith index to the first index available ([:i]) plus r\_letters at index ith + 1 ([i:]) to the last index available. This was to remove the letter from the remaining list in order to allow for any word scrambling. And finally, the method calls on itself taking in remaining\_letters and s\_letters + scramble\_letter as its parameters.

Next came a method to check whether or not the user’s word belonged in the long list of downloaded words. If it returned true, then the word would be added to a new list. Another method was created to remove any word duplicates found within the new list. After the removal of the duplicates, the words that remained were stored in a final list. Using the function sort, the permutations were then listed alphabetically. These were then printed along with the time it took to run them. Next came the optimizations. The first optimization that was implemented was to check for any duplicate letters within the user’s chosen word. This was done by a nested for loop where i (the outer loop) would iterate through the length of the user’s word and j (the inner loop) would again iterate through the length of the user’s word. A nested loop would assure that all the elements within the word were being compared. This method would return true if the word contained any replicates and false if it did not. The second optimization was then made. This optimization consisted of creating a new set to store the prefixes of the word. These two optimizations would allow for the anagrams to be found quicker.

The design choices created themselves. I knew the very first thing that was necessary in order to set the code into motion was to hard code the file that was downloaded into a set. Then from there the user was asked for a word and immediately the word was tested to see if it was even eligible, meaning if it was even in the list of words downloaded. Once the anagrams were composed, then came the task to print the words in alphabetical order and without any duplicates.

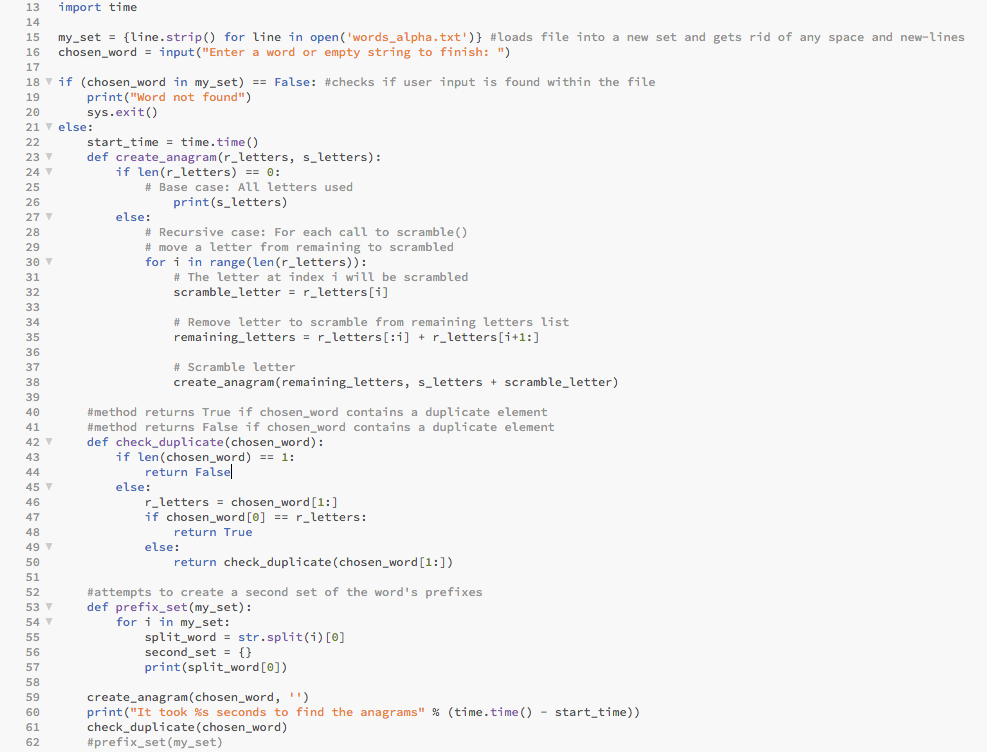
Experimental Results

The experiment that was conducted was inputting a word that was not found within the list of downloaded words. The name *Anakin* was inserted and the anagram was run. The only problem with was that name, *Anakin*, was not found within the list but instead various similar words such as *canakin*, *hyperanakinesia*, and many more where they all contained the name *Anakin*. With this in mind, the user’s input could simply not be checked to see if it was found within the list by using the set of prefixes. If this were to be done then it would return true, meaning the word is eligible to use even though it is not since the word *Anakin* alone is not part of the downloaded words. As to avoid this, as soon as the user input was set to a variable, the word was then sent to an if statement that stated if the user’s word was found within the list of downloaded words it would continue in its objective. If not, then the system would exit. With this implemented, the word *Anakin* was no longer eligible, but the word *canakin* was, and rightfully so.

Conclusions

From this lab, a more in-depth understanding of recursion was gained. Tracing was a necessity for figuring out how to implement recursion and in that there was trial and error. This repeated practice allowed me to adapt the best way to trace these recursion problems which was drawing out boxes with arrows clearly indicating the order of what was being returned. A visual representation of recursion made it so I was able to understand it more clearly.

Appendix



Academic Honesty Certification

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

Sofia Gutierrez September 9, 2019