**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #1**

Due: September 6, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

Lab 1: Recursion focuses on developing a program that allows a user to input a word and then display the anagrams of the word. In order to identify these anagrams, the program must first read a downloaded file version from GitHub, located at *https://raw.githubusercontent.com/dwyl/english-words/master/words\_alpha.txt,* which provides a thorough list of words from the English language. Since the main objective of this lab is to reinforce recursive solutions, the programs from Part 1 and Part 2 compute the anagrams recursively and display the results alphabetically. However, Part 2 focuses on developing a more efficient recursive function that eliminates unnecessary recursive calls by avoiding duplicate anagram creations and evaluating prefixes. It is also crucial to note that this lab introduces a new abstract data type, sets, which will be used to store the anagrams and other data.

**Proposed Solution Design and Implementation**

**Part 1:**

For part 1, I focused on developing a program that read the words\_alpha.txt file from GitHub, populated a set with these words, prompted the user to enter a word of their choice, computed the anagrams, and displayed the anagrams. I also calculated and displayed the performance time for the recursive method, FindAnagrams. Before solving the actual problem, I first divided my program into 5 different sections. The first section included a function to read the words\_alpha.txt file. In this function, I first open the file, passed as a parameter, and read each line sequentially. As the file was read, I populated a set with the valid English words. From the documentation, I learned that each line of a file ends with a newline character. Since I only wanted the set to contain the words, I removed the newline character from each word before adding it to the set. Additionally, I realized that a user may accidentally add a blank line at the top or bottom of the txt file. In order to prevent these blank lines from being added, I ensured that each line was not just whitespace before adding it to the set. Lastly, I learned from the documentation that opening files in Python can result in a FileNotFoundError or IOError. I thus placed the file opening and read statements within a try block and caught the errors if they occurred. I returned None if an error occurred to signal that the file’s read was unsuccessful. If an error did not occur, I returned the set of English words.

After the file was successfully read, I continuously asked the user to enter a word, or empty string to finish. If the user entered a word, my program would transition to the second section which was the recursive function FindAnagrams. The input for this method includes the user’s word, the set of the English words, the set of anagrams that would be populated, the original word, and the prefix index. The prefix index was initialized to 0, denoting that a prefix had yet to be created. A local variable was assigned the letters from the prefix index to the rest of the string. These remaining words were iterated through and each one was added to the pre-existing prefix denoted by the prefix index. For example, take the word “mac”. If the prefix index was 0, then the current prefix would be the empty string “”. The remaining letters would be “m”, “a”, and “c”. As each one of these letters were iterated through, they were appended to the prefix and a new recursive call was made passing in the prefix with the other letters appended at the end. So, for “m”, the next word passed would be “mac”, for “a”, the word passed would be “amc”, and for “c”, the next word passed would be “cam”. The prefix index would also be incremented by 1. Once the prefix index equaled to the length of the string, the word was added to the set of anagrams if it was a valid word. Since I passed in a copy of the reference to the anagram set, I did not return any values and just populated the set.

Sections 3 and 4 of Part 1 are described below in Shared Design. Section 5 of the program was the actual execution. I ensured that every time the user wanted to identify the anagrams of a word, I would calculate the performance time. I thus subtracted the time before the recursive algorithm from the time after the algorithm’s completion and displayed the results.

**Part2:**

The overall functionality of the program in Part 2 is similar to that of Part 1. The words\_alpha.txt file is read, the words are added, and the anagrams based on the user’s word are printed and displayed. However, the program’s FindAnagram function is more efficient than that of Part 1 because it reduces the number of recursive calls made by using a prefix set and preventing duplicate strings. Before implementing these changes, I first designed the overall layout of the program. Part 2’s program is divided into 6 sections. The first section is the opening and reading of the file. Unlike Part 1’s program’s file read and open function, Part 2’s function includes two empty sets, one for the English words and one for their prefixes, that will be populated. Part 2 does not return a set since the copy of their references are passed as parameters. In order to catch the resulting errors that may occur, I implement a try and execute branch around the FindAnagram’s method call. Just as Part 1’s program opens the file and reads each line sequentially, Part 2’s program does so as well. However, since n! permutations are being created in Part 1, where n is the length of the word, Part 2 stresses to reduce these recursive calls by identifying if their prefix exists. Thus, as I added the words to the set of English words, I also called a method, AddPrefixes which would add the prefixes of word from the current line.

AddPrefixes accepts the current set of prefixes as well as the current word from the line being read. I have the program first check to ensure that the word is not already in the prefix set before its prefixes are added. For example, if the prefixes of the word “rate” were already added and the next line has “rat,” then all of the prefixes of “rat” will already have been added to the set. If the word did not exist in the set, I would continuously remove the last letter and add it to the set. Before adding the prefix to the set, I would again check if it already existed in the set and stop the recursive calls. Since the prefix set was passed in as a parameter, I did not return a value.

After reading the file and populating the English word and prefix sets, I moved on to the third section, FindAnagrams. FindAnagram’s included all the parameters from Part 1’s FindAnagram’s parameters, however it also included the prefix set as a new parameter. This prefix set was used to prevent further recursive calls from being made if the current appended letters were not in the prefix set. Additionally, in order to prevent duplicate anagrams from being created, I enforced new logic in my program. Before iterating through the remaining letters and making the recursive calls, I first created an empty set which would keep track of all the letters used. While the remaining letters were being iterated through, I would first check if the letter was already in the set. If it was, a recursive call would not be made and the next letter would be evaluated. For example, given the word “ana,” the Part 1 would make recursive calls with the prefix of “a,” “n,” and “a.” However, for Part 2, only a recursive call for “a” and “n” would be made. This is because when the second “a” is reached, the program checks to make sure that a recursive call with a prefix appending “a” has not been made yet.

As for section 4 and 5, they will be described below in shared design. Section 6 included the actual implementation of the program.

**Shared Design**

Both Part 1 and Part 2 included methods that would allow the user to input the alpha\_word.txt directory or name if the file could not be found. I decided to implement this method to ensure that if the words\_alpha.txt file was in a different directory from the program, the program would not crash. I thus created a method that would gracefully tell the user that the file could not be found and allow them to edit the file’s path. I also included another method to ensure that if the .txt file name was changed it would end in “.txt.”

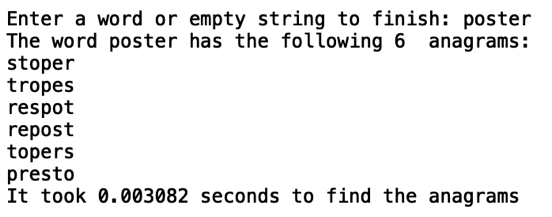
**Experimental Results**

**Part 1:**

For Part 1, I decided to insert “poster”, “university”, “permutation”, “ ”, “”, and “anna” as words. I tested “poster”, “university”, and “permutation” to ensure that the anagrams would be equal to those provided. Also, university provides an edge case for having 0 anagrams. I also added “” to ensure that the program would terminate for an empty string. “ ” is also used as an edge case to ensure that an anagram search would not occur. Lastly, I tested “anna” because it contains duplicate letters and I wanted to compare its efficiency with that of Part 2.

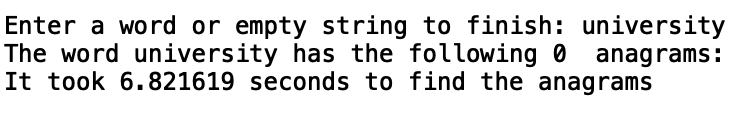
Case 1 (“poster”):

Input Word: “poster”



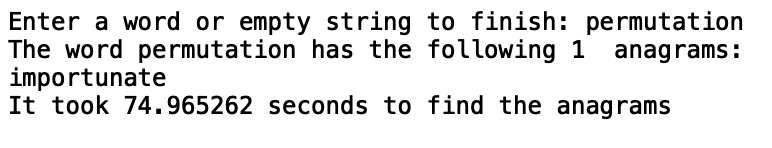
Case 2 (“university”):

Input Word: “university”



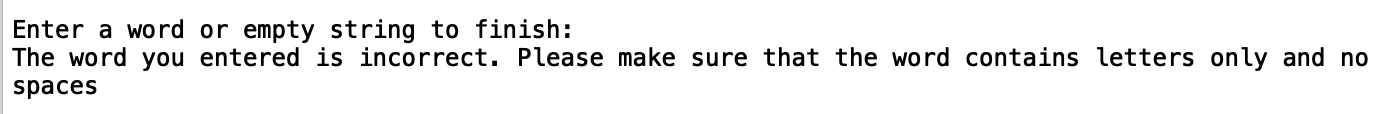
Case 3 (“permutation”):

Input Word: “permutation”



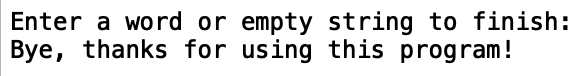
Case 4 (“ ”):

Input Word: “ ”



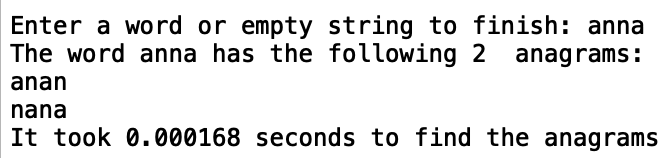
Case 5 (“”):

Input Word: “”



Case 6 (“anna”):

Input Word: “anna”



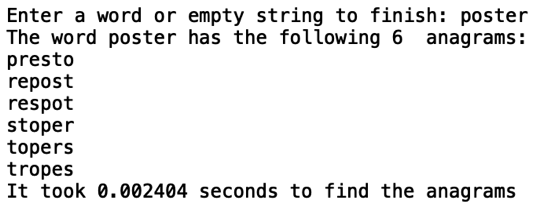
As the cases demonstrate, the program was able to function for all the words and match the same anagrams as those provided in the lab’s examples. Most startling of the cases was Case 3. Case 3 took approximately 75 seconds to perform the recursive method. The program also functioned correctly during Case 4 since an anagram cannot be composed of white spaces.

**Part 2:**

For Part 1, I decided to insert “poster”, “university”, “permutation”, “ab6”, “”, and “anna”. I tested “poster”, “university”, and “permutation” to ensure that the anagrams would be equal to those provided. I also added “” to ensure that the program would terminate for an empty string. “ab6” is also used as an edge case to ensure that an anagram search would not occurs since numbers are included. Lastly, I tested “anna” because it contains duplicate letters and I wanted to compare its efficiency with that of Part 1.

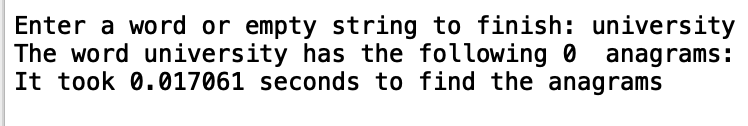
Case 1 (“poster”):

Input Word: “poster”



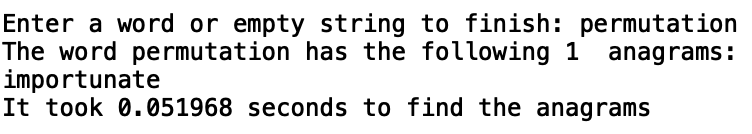
Case 2 (“university”):

Input Word: “university”



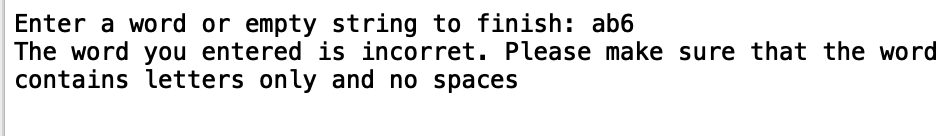
Case 3 (“permutation”):

Input Word: “permutation”



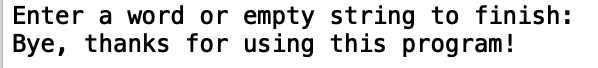
Case 4 (“ab6”):

Input Word: “ab6”



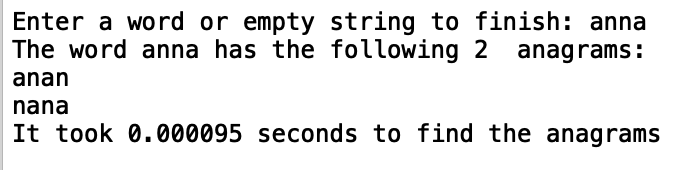
Case 5 (“”):

Input Word: “”



Case 6 (“anna”):

Input Word: “anna”



**Shared Design:**

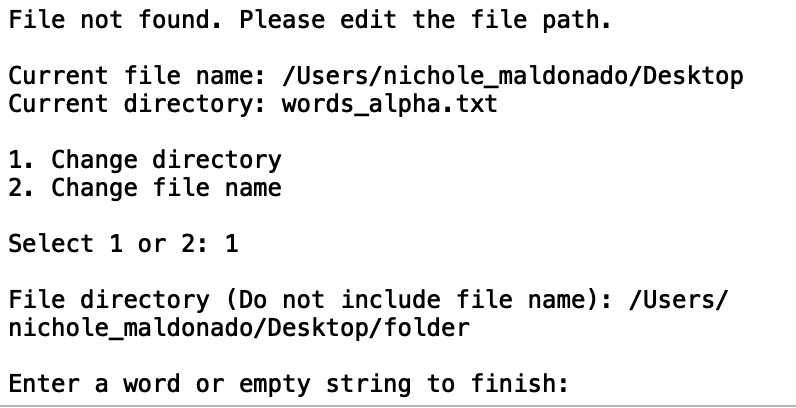
For both Part 1 and Part 2, I tested their common method FilePathSetup. FilePathSetup is called if the words\_alpha.txt file is not located in the same directory as the programs or has a different name. For Case 1 and Case 2, the programs are located in the same directory “/Users/nichole\_maldonado/Desktop”. The words\_alpha.txt file is located in “/Users/nichole\_maldonado/Desktop/folder” instead. Thus, this tests the edge case of the file and .txt file not being in the same location. It is very important for the program to handle this with ease rather than crash. Case 2 has the file located in the same directory as the programs, however its name has been changed to “banana.txt”.

NOTE: In all cases, the default file path relies on where the user currently has the python file.

Case 1 (Wrong directory):

Correct file path: “/Users/nichole\_maldonado/Desktop/folder/words\_alpha.txt”

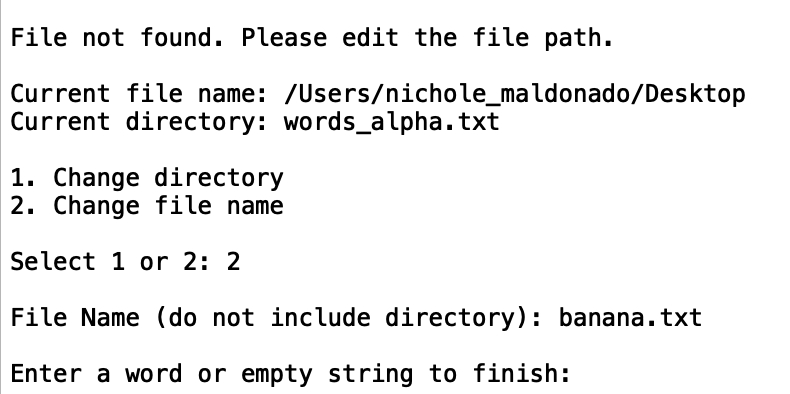
Program’s default file path: “/Users/nichole\_maldonado/Desktop/words\_alpha.txt”



Case 2 (Wrong File Name):

Correct File Name: “banana.txt”

Default File Name: “words\_alpha.txt”



Both Case 1 and Case 2 demonstrate that even if the English word file is located in a different directory or has a different name, the file can still be read once the user inputs the correct data.

**Performance Time:**

In order to compare the performance of the programs, I entered a range of words that include “apple”, “green”, “presto”, “step”, and “zoo”. Each word was entered five times sequentially. Once the word was inserted 5 times, I stopped the program and restarted the same procedures for the next word. By including words from starting with “a” to “z”, I was able to test the best and worst cases of the programs.





****

The performance graphs above demonstrate that for all instances, the program for Part 2 had a shorter performance time than the program for Part 1. By reducing the number of recursive calls through prefix checks and duplicate letter avoidance, Part 2’s performance showed its superiority.

**Recursive Calls:**

In order to verify that Part 2 is reducing the number of recursive calls, the following test compares the number of recursive calls that occurred for each word. The number of recursive calls were calculated by a global counter. This counter can be still be viewed but are commented out since they were for testing purposes only.

****

****

For all the recursive calls, Part 1 had a larger amount than Part 2. As can be seen with the number of recursive calls for coconut, the number of recursive calls for Part 2 is almost negligible when compared to Part 1.

**Conclusion**

In all, I learned from the lab that recursive calls can be very efficient with the proper logic. By simply eliminating the recursive calls for duplicate letters, I was able to reduce the performance time. For example, the word “permutation” took seconds for Part 1 and seconds for Part 2. Additionally, this lab taught me about a new data structure, sets. I had never heard of sets before and was hesitant about using them. I thought that looking for an element in a set would require at most, iterating through all the elements. However, after looking up information on sets online, I learned that the element checks were done, on average, in constant time. I also learned that since sets do not have duplicates, if you convert a list to a set, then all the duplicates will instantly be removed. Thus, sets are a data structure that I definitely plan on using moving forward. Another interesting topic that I learned about was the file locations. At first, I hardcoded the directory of the words\_alpha.txt file. However, I realized that other users who download the code may not have the code stored in the same directory. I was able to implement a feature to allow the user to access the .txt file from anywhere in their computer as long as they know its location. The results from the graph conclude the importance of careful planning in recursive functions. By using not only a set of words but also prefixes to compare the appending words, I was able to successfully limit the number of recursive calls. One of the most interesting things that I found in this lab was the large gaps in recursive calls. For the word “coconut”, Part 1’s program performed 13,196 more recursive calls than Part 2’s program. In all, this lab was very interesting and demonstrates that there is room for improvement in all programs.

**Appendix**

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.