Lab 1- Recursion Report

The problem at hand in Lab 1 called to create a program that prompts a user for a desired word and returns an alphabetically sorted list with all possible anagrams and the time it took to find all anagrams. The challenge was made more interesting with the addition of a constraint that was composed of having a recursive approach to our solution. We were given a text document containing over 466000 words in the English language. This was given to us in order to have a specific way to find anagrams of each permutation of the user’s chosen word. If the permutation was in the text file provided for us then it was to be considered an anagram.

After thoroughly brainstorming the different approaches that could be taken to solve this problem recursively, I decided on taking a very concise approach. I decided compressing all calculations into two functions would be possible and efficient. The first step was to scan and create a list with all the data in the text file. Using the python method splitlines() was the best choice because of the size of the text file. After having a list to which compare each permutation to find the anagrams, I had to create a method to do so.

This is where the compression decision came into play. I decided that instead of creating three different functions to create all possible permutations, find the anagrams, and doing away with repetition, all of these steps could be performed in the same function. My method ‘anagram’ is given a word, creates all possible permutations with the letters, then accesses the list created from the text file and finds which appear within the list. These words are then labeled as anagrams and are added into a set which is returned at the end of the function. The reasoning behind using a set to store the anagrams is to take care of the repetition problem that we encounter when a word has one or more of the same letters within the word.

To demonstrate the functionality of the function ‘anagram’ itself I conducted an experimental run using the word “spot” as we were given the anagrams.

A screenshot of a cell phone

Description automatically generated

As expected the result printed a set with a total of six anagrams. However, they are not sorted alphabetically yet because they are still in a set.

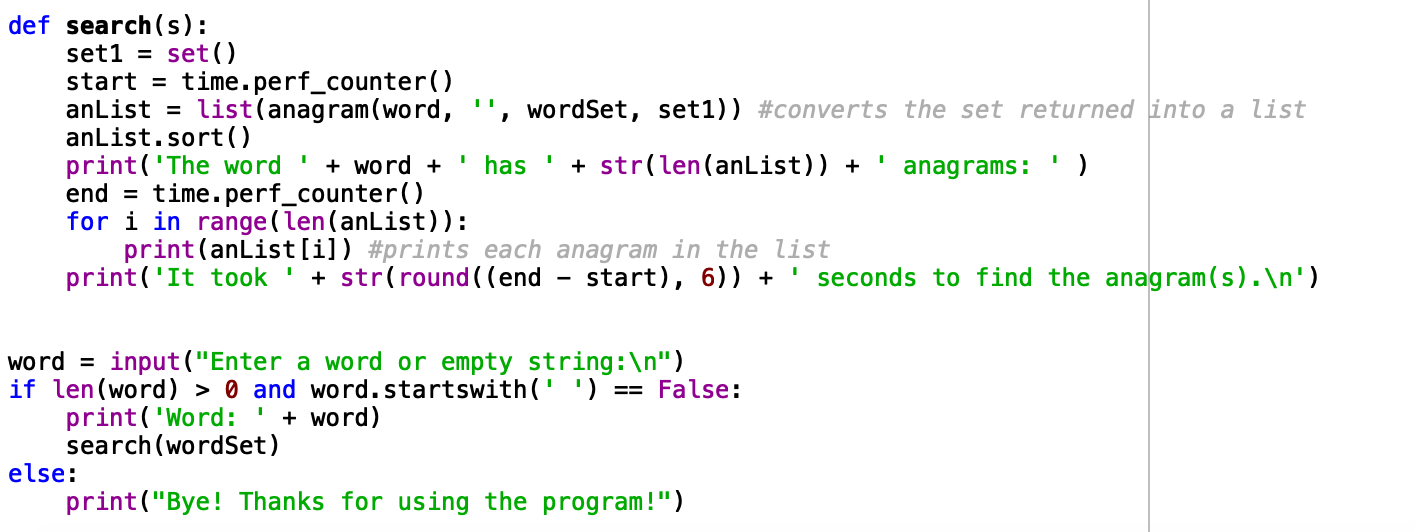
A close up of a logo

Description automatically generated

The second function, ‘search’, was left with the task of calling the first function ‘anagram’ and calculating the time it took for the program to find the anagrams of a given word. To calculate the time I first imported the time module. At first I found myself using the time.clock() method to calculate the time but after running test trials python suggested I use ﻿time.perf\_counter() instead to acquire a more accurate result. I saved the time at which the function was called in a variable and subtracted the time at which the program finished. This gave me an accurate running time which I then rounded to the nearest six decimal places. My main method only contained the user interface which prompts the user for a word of their choice

and uses it as input for the functions. To be able to catch an invalid entry such as an empty string I decided to use a simply if else statement.

An experimental run was conducted in order to demonstrate the functionality of the function ‘search’ using the same word “spot.”



First the user is prompted to input a word of their choice or an empty string in this case ‘spot.’

A close up of a logo

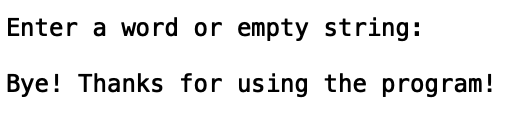
Description automatically generated

After the user completes this step the final output is displayed in the console. As expected, we receive six anagrams printed in alphabetical order along with the time it took to find the anagrams rounded to the first six decimal places.

A picture containing plant, animal

Description automatically generated

I would also like the point out the way my program handles unexpected inputs such as a simple space rather than an empty string. If a user inputs a single space it will be taken as an empty string and give the same output. Such as it is shown below.



I learned a lot from completing this recursive lab. Not only was it my first program in python but I also further expanded my knowledge in recursion. Python has so many modules that help facilitate a coder’s solution implementation and aid when attempting to make our code more efficient.

**Appendix**

﻿import time

wordSet = set()

wordSet = open("words\_alpha.txt").read().splitlines()

**Anagram Function**

﻿def anagram(w, scramble, words, s):

if len(w) == 0:

if scramble in words: #checking to see which permutations are anagrams

s.add(scramble)

else:

for i in range(len(w)):

chosen = w[i] #chooses letter to begin each permutation word with scramble

remaining = w[:i] + w[i+1:] #joins the letters in the word that were not the chosen

anagram(remaining, scramble + chosen, words, s)

return s

**Search Function**

﻿def search(s):

set1 = set()

start = time.perf\_counter()

anList = list(anagram(word, '', wordSet, set1)) #converts the set returned into a list

anList.sort()

print('The word ' + word + ' has ' + str(len(anList)) + ' anagrams: ' )

end = time.perf\_counter()

for i in range(len(anList)):

print(anList[i]) #prints each anagram in the list

print('It took ' + str(round((end - start), 6)) + ' seconds to find the anagram(s).\n')

**Main**

﻿word = input("Enter a word or empty string:\n")

if len(word) > 0 and word.startswith(' ') == False:

print('Word: ' + word)

search(wordSet)

else:

print("Bye! Thanks for using the program!")

**Academic Honesty Statement**

“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.”

* Dafne Bencomo