**CS 2302 Data Structures**

**Spring 2019**

**Lab Report #1**

Due: September 6, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab we were asked to work with recursive functions in searching for the anagrams of a word. It was important to recall the elements of recursion as well as the different data structures that facilitate the movement and comparison of data. The main objective of this lab is to use recursion to search through a set without having to use any type of loop. When optimizations were added the program through recursion and optimization ran in a much more efficient manner.

**Proposed Solution Design and Implementation**

**Part #1**

For Part 1 I attempted to modify the code that was present in zybooks which created all possible permutations of a given word. I took this source code and kept the if and else statements as well as the portion that scrambles the word by moving indexes as well as the recursive call. The base case of my modified code holds that the length of the given word will be equal to 0 when it has been scrambled completely. From this base case we enforce certain restrictions to add it to the set that contains the anagrams. We first implement a for loop that will iterate through the dictionary text checking if the scrambled word is in it. If the word is in the dictionary text then we add it to the set which contains the anagrams. The else statement in this function is where the recursive call takes place. The for loop iterates over the word, thus creating a different permutation with it being returned to see if it is added into the anagram word set. When making the recursive call the function creates a different permutation.

**Part #2**

For Part 2 we simply added a function which produces prefixes of the words in the text file and compare the scrambled words to these prefixes. From this we can eliminate making recursive calls on scrambled words that do not match the prefixes of the words in the text file. The prefix function is called directly on the set which contains the elements of the text file. This function iterates through each word and separates from a singular element to a element that is one less than the actual word. The function works this way as to not compare actual words when wanting to discard certain scrambled words by comparing them to the prefix set. The only other modification other than the creation of the prefix function is the implementation of this function in the findAnagram function. This is done primarily in the else portion of the function where we take the prefixes and compare them to the partial scrambled word using an if statement. If this partial scrambled word is in the prefix set, then we make a recursive call on the partial scrambled word. If it is not then we simply ignore it and do nothing. This modification greatly improved the run time of the program in comparison to the functions in Part 1.

**Experimental Results**

**Part #1**

Testing with word = ‘dad’

﻿

Enter a word or empty string to finish: dad

['add', 'dad']

Runtime: 0.2718145979997644

Testing with word = ‘rather’

﻿Enter a word or empty string to finish: rather

['rather']

Runtime: 26.442947832001664

Testing with word = ‘race’

﻿Enter a word or empty string to finish: race

['acer', 'acre', 'care', 'cera', 'crea', 'race']

Runtime: 1.0678602609987138

**Part #2**

Testing with word = ‘dad’

﻿Enter a word or empty string to finish: dad

['add', 'dad']

Runtime: 0.9897934260006878

Testing with word = ‘rather’

﻿Enter a word or empty string to finish: rather

['rather']

Runtime: 0.9866468779982824

Testing with word = ‘race’

﻿Enter a word or empty string to finish: race

['acer', 'acre', 'care', 'cera', 'crea', 'race']

Runtime: 0.9924852300027851

**Conclusion**

What I learned from this lab is recursion and divide and conquer method. Recursion was used to check and create the scrambled words that would be compared partially to the prefix set and then to the set that contained all possible words. Divide and conquer was used not only when attacking how to solve the problem but also when writing the code with the modifications. This took some critical thinking in breaking apart the problem to see where the modifications. In conclusion the skills learned in this lab will serve me in attacking future labs with a clear idea on how to go about solving the problem.

**Appendix**

**﻿**

**from timeit import default\_timer as timer**

**my\_set = set(line.strip() for line in open('words\_alpha.txt'))**

**word\_set = set([])**

**prefix\_set = set([])**

**permutation\_set = set([])**

**def findAnagram(word,temp):**

**if len(word) == 0:**

**if temp in my\_set and temp != word and temp not in word\_set:**

**word\_set.add(temp)**

**else:**

**for i in range(len(word)):**

**scrambles = word[i]**

**remaining = word[:i] + word [i+1:]**

**if (temp + scrambles) in prefix\_set:**

**findAnagram(remaining, temp + scrambles)**

**def findAnagramP1(word,temp):**

**if len(word) == 0:**

**for i in my\_set:**

**if temp == i:**

**word\_set.add(temp)**

**else:**

**for i in range(len(word)):**

**scrambles = word[i]**

**remaining = word[:i] + word [i+1:]**

**findAnagramP1(remaining, temp + scrambles)**

**def prefix2 ():**

**for i in my\_set:**

**for j in range(1,len(i)):**

**prefix = i[:j-1]**

**prefix\_set.add(prefix)**

**word = input('Enter a word or empty string to finish: ')**

**start = timer()**

**prefix2()**

**findAnagram(word,'')**

**print (sorted(word\_set))**

**end = timer()**

**print('Runtime:',str(end-start))**

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