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

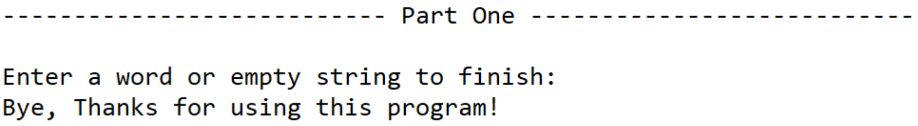
Operation #1:

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

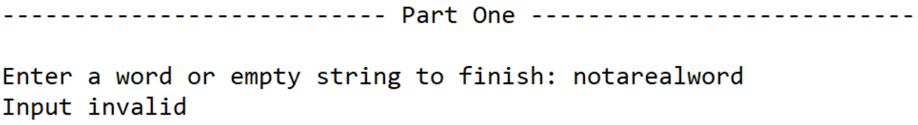


```
-------------------------- Part One --------------------------

Enter a word or empty string to finish:
Bye, Thanks for using this program!
```

Case 2 (‘notarealword”):

Anagrams found: None

Time: None

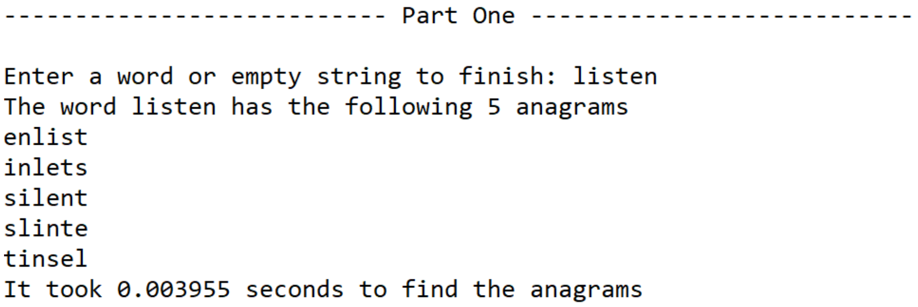


```
-------------------------- Part One --------------------------

Enter a word or empty string to finish: notarealword
Input invalid
```

Case 3 (“listen”):

Anagrams found: 5

Time: 0.003955s

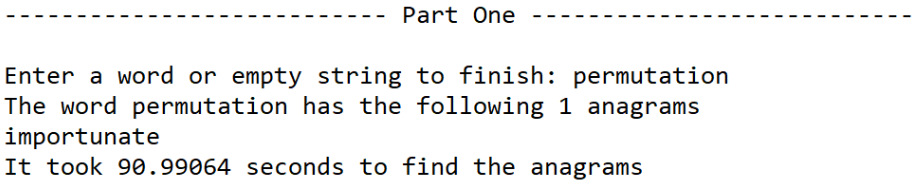


```
-------------------------- Part One --------------------------

Enter a word or empty string to finish: listen
The word listen has the following 5 anagrams
enlist
inlets
silent
slinte
tinsel
It took 0.003955 seconds to find the anagrams
```

Case 4 (“premutation”):

Anagrams found: 1

Time: 90.9906s



```
-------------------------- Part One --------------------------

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams
importunate
It took 90.99064 seconds to find the anagrams
```

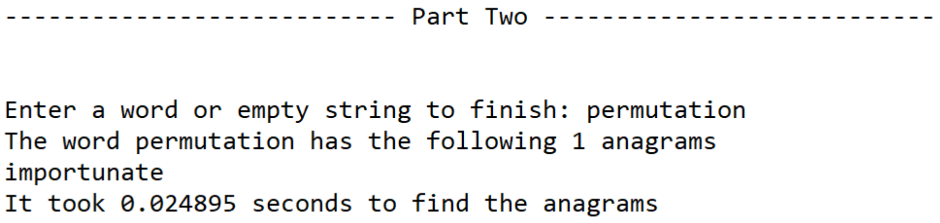
Operation #2:

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

Case 1("permutation"):

Anagrams found: 1

Time: 0.024895s

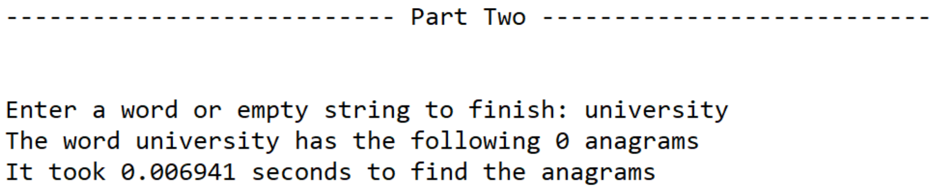


```
-------------------------- Part Two --------------------------

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams
importunate
It took 0.024895 seconds to find the anagrams
```

Case 2("university"):

Anagrams found: 0

Time: 0.006941s

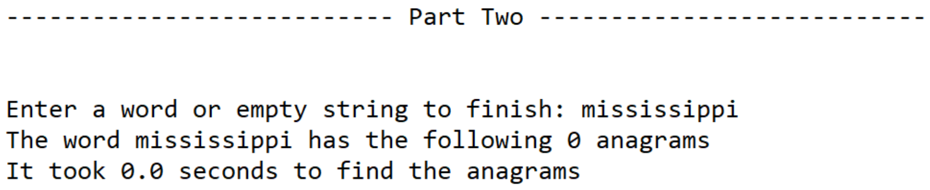


```
-------------------------- Part Two --------------------------

Enter a word or empty string to finish: university
The word university has the following 0 anagrams
It took 0.006941 seconds to find the anagrams
```

Case 3("mississippi"):

Anagrams found: 0

Time: 0.0s



```
-------------------------- Part Two --------------------------

Enter a word or empty string to finish: mississippi
The word mississippi has the following 0 anagrams
It took 0.0 seconds to find the anagrams
```

Graphs of Time Differences

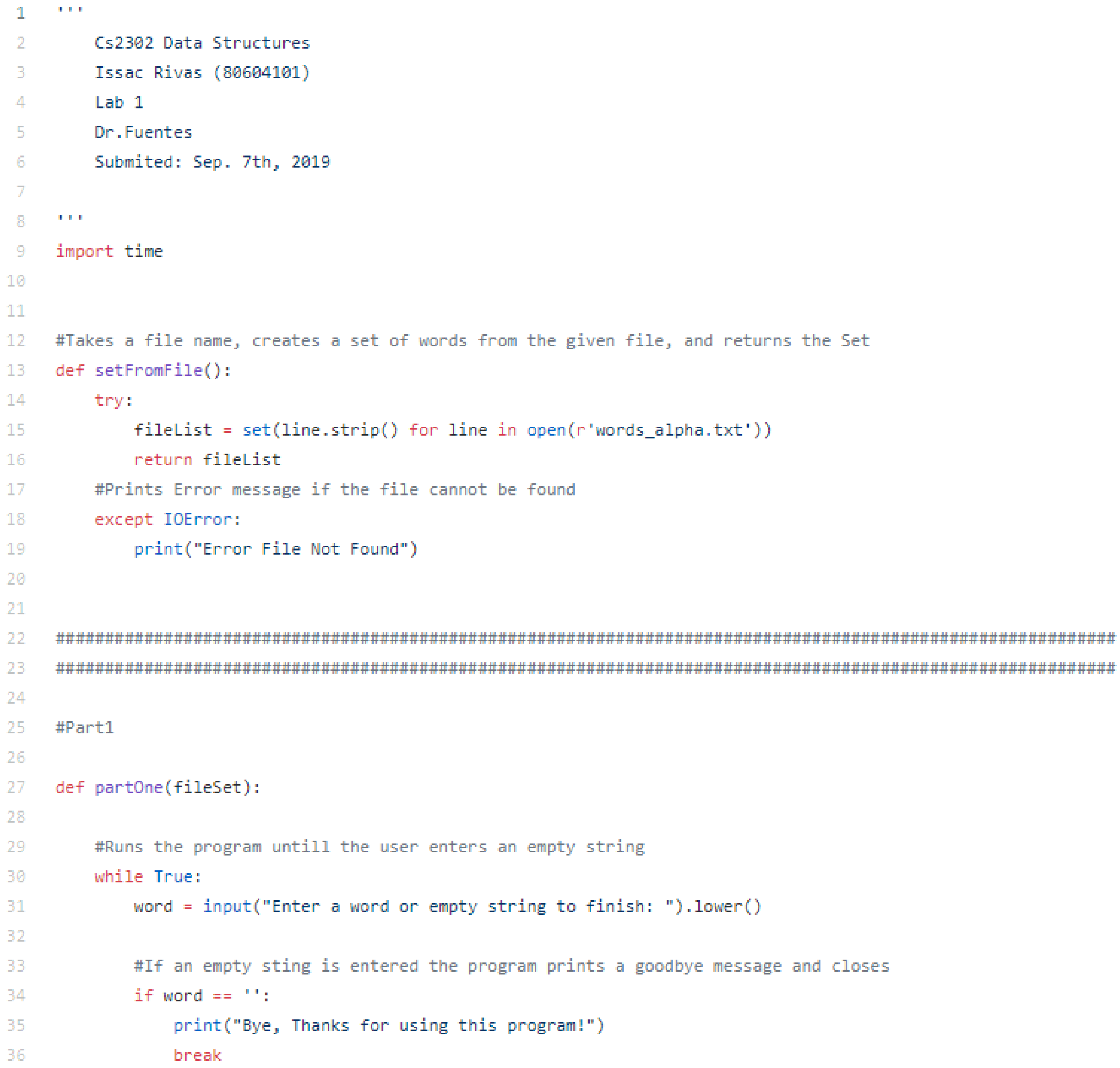
Word Length	Normal Method Time	Optimized Method Time
"list"	0.0 Seconds	0.0 Seconds
"towering"	0.085766 Seconds	0.002993 Seconds
"university"	8.085670 Seconds	0.004986 Seconds
"jeopardizing"	1175.127 Seconds	0.017951 Seconds

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.

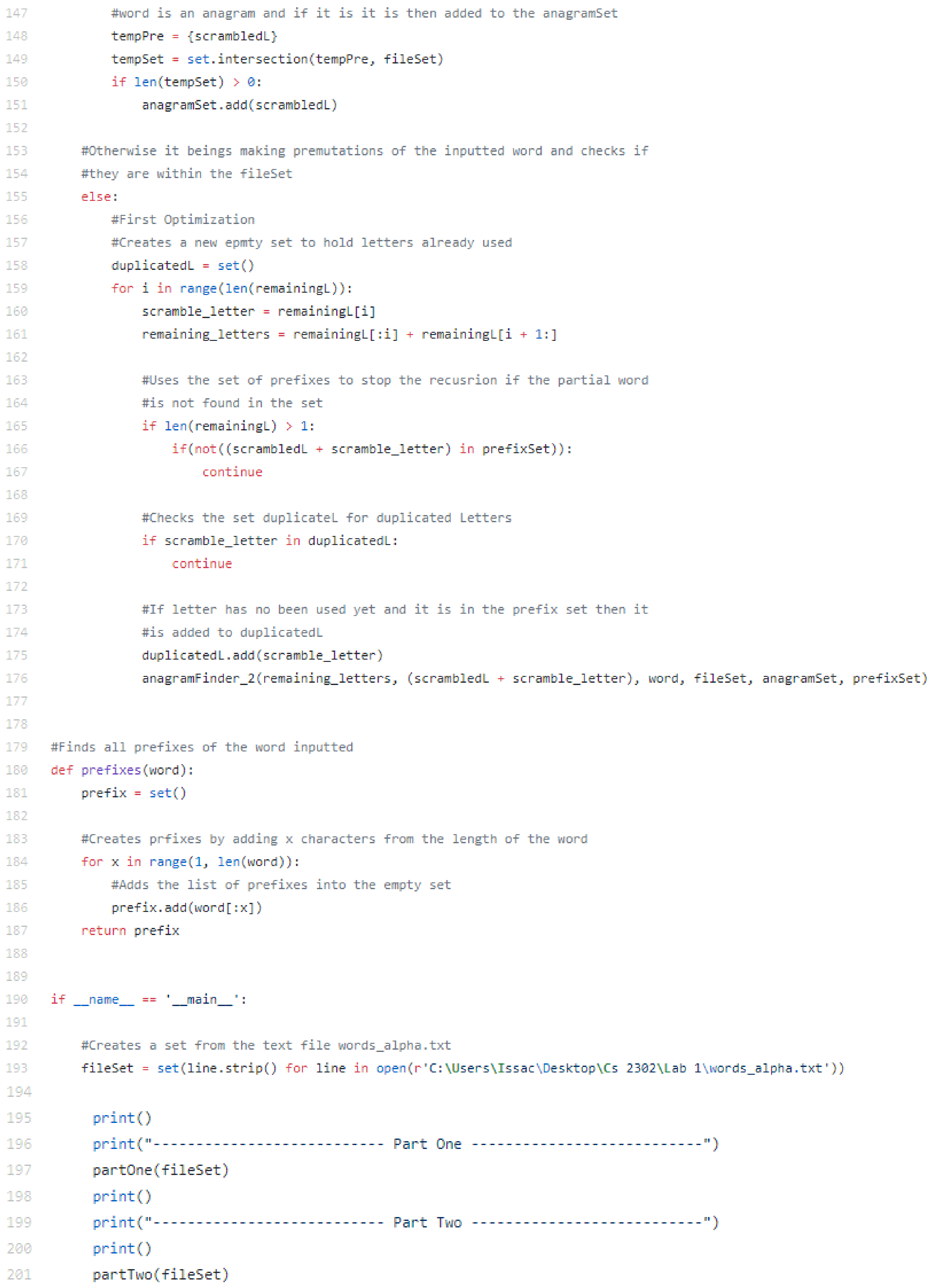
Conclusion

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



```python
'''
    Cs2302 Data Structures
    Issac Rivas (80604101)
    Lab 1
    Dr.Fuentes
    Submited: Sep. 7th, 2019

'''
import time


#Takes a file name, creates a set of words from the given file, and returns the Set
def setFromFile():
    try:
        fileList = set(line.strip() for line in open(r'words_alpha.txt'))
        return fileList
    #Prints Error message if the file cannot be found
    except IOError:
        print("Error File Not Found")


############################################################################################################
############################################################################################################

#Part1

def partOne(fileSet):

    #Runs the program untill the user enters an empty string
    while True:
        word = input("Enter a word or empty string to finish: ").lower()

        #If an empty sting is entered the program prints a goodbye message and closes
        if word == '':
            print("Bye, Thanks for using this program!")
            break

        #If a word is not inputed then it will prompt the user that that there input
        #is invalid
        elif(not(word in fileSet)):
            print("Input invalid")
        else:

            #Sorts the inputted word alphabetically
            sortedList = sorted(list(word))
            sortedWord = ''.join(sortedList)

            #Adds the word to the set of anagrams to start the set so the word
            #is already in the list when we try and remove it
            anagramSet = {word}
            start = time.time()
            anagramFinder(sortedWord, '', word, fileSet, anagramSet)
            end = time.time()
            anagramSet.remove(word)


            anagramList = sorted(list(anagramSet))
            print("The word",  word, "has the following", len(anagramSet), "anagrams")
            for anagram in anagramList:
                print(anagram)
            print("It took", ((str(end - start))[:8]), "seconds to find the anagrams")
            print()


def anagramFinder(remainingL, scrambledL, word, fileSet, anagramSet):

    #Base Case if there are not letters remaining
    if len(remainingL) == 0:

        #Adds the word to a set then compares itself with fileSet to see if the
        #word is an anagram and if it is it is then added to the anagramSet
        tempPre = {scrambledL}
        tempSet = set.intersection(tempPre, fileSet)
        if len(tempSet) > 0:
            anagramSet.add(scrambledL)

    #Otherwise it beings making premutations of the inputted word and checks if
    #they are within the fileSet
    else:
        for i in range(len(remainingL)):
            scramble_letter = remainingL[i]
            remaining_letters = remainingL[:i] + remainingL[i + 1:]
            anagramFinder(remaining_letters, (scrambledL + scramble_letter), word, fileSet, anagramSet)



#####################################################################################################
#####################################################################################################

#Part2

def partTwo(fileSet):

    #Runs the program untill the user enters an empty string
    while True:
        word = input("Enter a word or empty string to finish: ")

        #If an empty sting is entered the program prints a goodbye message and closes
        if word == '':
            print("Bye, Thanks for using this program!")
            break

        #If a word is not inputed then it will prompt the user that that there input
        #is invalid
        elif(not(word in fileSet)):
            print("Input invalid")
        else:

            #Sorts the inputted word alphabetically
            sortedList = sorted(list(word))
            sortedWord = ''.join(sortedList)

            #Adds the word to the set of anagrams to start the set so the word
            #is already in the list when we try and remove it
            anagramSet = {word}

            prefixSet = set()

            #Creats and Adds prefixes for all words in the fileSet
            #and adds them into the set prefixSet
            for setWord in fileSet:
                prefixSet.update(prefixes(setWord))

            start = time.time()
            anagramFinder_2(sortedWord, '', word, fileSet, anagramSet, prefixSet)
            end = time.time()

            #removes the original word from the anagramSet
            anagramSet.remove(word)

            #Converts anagramSet to a list, sorts them alphabetically, and
            #then prints them alphabetically
            anagramList = sorted(list(anagramSet))
            print("The word",  word, "has the following", len(anagramSet), "anagrams")
            for anagram in anagramList:
                print(anagram)
            print("It took", ((str(end - start))[:8]), "seconds to find the anagrams")
            print()


def anagramFinder_2(remainingL, scrambledL, word, fileSet, anagramSet, prefixSet):

    #Base Case if there are not letters remaining
    if len(remainingL) == 0:

        #Adds the word to a set then compares itself with fileSet to see if the
```



```python
        #word is an anagram and if it is it is then added to the anagramSet
        tempPre = {scrambledL}
        tempSet = set.intersection(tempPre, fileSet)
        if len(tempSet) > 0:
            anagramSet.add(scrambledL)

    #Otherwise it beings making premutations of the inputted word and checks if
    #they are within the fileSet
    else:
        #First Optimization
        #Creates a new epmty set to hold letters already used
        duplicatedL = set()
        for i in range(len(remainingL)):
            scramble_letter = remainingL[i]
            remaining_letters = remainingL[:i] + remainingL[i + 1:]

            #Uses the set of prefixes to stop the recusrion if the partial word
            #is not found in the set
            if len(remainingL) > 1:
                if(not((scrambledL + scramble_letter) in prefixSet)):
                    continue

            #Checks the set duplicateL for duplicated Letters
            if scramble_letter in duplicatedL:
                continue

            #If letter has no been used yet and it is in the prefix set then it
            #is added to duplicatedL
            duplicatedL.add(scramble_letter)
            anagramFinder_2(remaining_letters, (scrambledL + scramble_letter), word, fileSet, anagramSet, prefixSet)


#Finds all prefixes of the word inputted
def prefixes(word):
    prefix = set()

    #Creates prfixes by adding x characters from the length of the word
    for x in range(1, len(word)):
        #Adds the list of prefixes into the empty set
        prefix.add(word[:x])
    return prefix


if __name__ == '__main__':

    #Creates a set from the text file words_alpha.txt
    fileSet = set(line.strip() for line in open(r'C:\Users\Issac\Desktop\Cs 2302\Lab 1\words_alpha.txt'))

    print()
    print("--------------------------- Part One ---------------------------")
    partOne(fileSet)
    print()
    print("--------------------------- Part Two ---------------------------")
    print()
    partTwo(fileSet)
```

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.