Elisa Jimenez Todd  
CS 2302 – Data Structures  
Instructor: Olac Fuentes  
9/6/19

**Lab 1 – Report**

1. **Introduction**

The user will input a word for which my program must show all anagrams.

My program must be able to find all possible anagrams the word that the user inputs. It reads a text file with all words in the English dictionary. The program will scramble the word with a recursive function and compare it with the ones in the text file to find if they are anagrams. The output will be the total number of anagrams that could be formed with the original word and the anagrams.

1. **Proposed solution design and implementation**

The first step I took was figuring out how to read a text file on Python. Then, I figured out how to read it into a set. I stored it in wordSet.

After that, I worked on the user interface. How the program was going to ask for the words, how it was gonna display the results, and what messages to show.

Here I made a loop so that as long as the user inputs a word, the program keeps asking for another, but if the user input nothing, the program would end. This was done with a ‘while’ and a Boolean named ‘stay’ with value of ‘True’ as default. If the user entered an empty string, ‘stay’ changed to false and no other command was executed, the loop was exited, and the program ended.

I made 3 sections inside the loop to test each respective version of my algorithm.

Next, I worked on the function to get the anagrams. I based it on the example ZyBooks showed to scramble words. After each scrambled word, I compared it with all the words on wordSet, if the scrambled word was on wordSet, I added 1 to my counter and stored the word on a list (words\_result). To make the list and counter change, I made them global variables. This function worked well but took too long on long words. I had to make the string sorted by alphabetical order so that it would print that way too, so I did that in nameAlf.

I added the time, so I had to import time, and then get the time before the call to the function and after. The output subtracts both to get the total time.

Then, I worked on the first optimization, which I had a hard time with because it was not really working. After a while I realized that I could store the inputs on a set (letterSet), so that I could implement an if statement that if the new input had already been used, not to use it again.

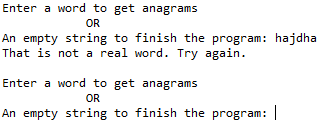
The second optimization had to take advantage of another function that stores prefixes of a list into a set. This function compared the sorted letters to the prefixes, so that if there were no combinations of such so far, the program wouldn’t continue. I had to reset words\_result before this.

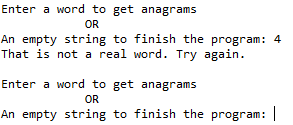
Then, I combined both. The challenge was to reset words\_result and letterset, because I had not taught about it, it was messing with my algorithm. But finally, it worked.

1. **Experimental results**

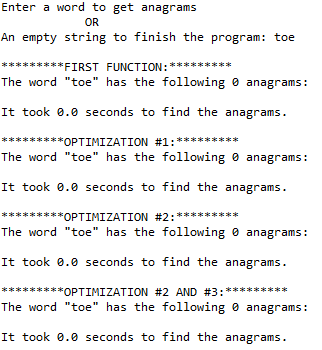
I tested all methods with different kinds of words to make sure the program worked. From easy to hard. Also testing the examples from the instructions to make sure the functions were doing their job.

I added a feature so that if the user inputs a word that does not exist, the program tells him/her to start again. Testing “hajdha” and “4”:

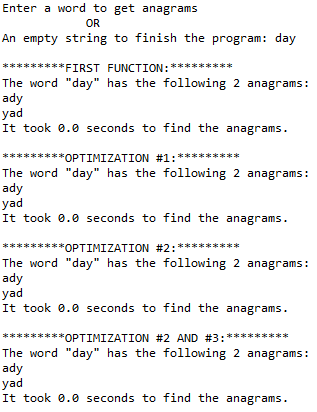




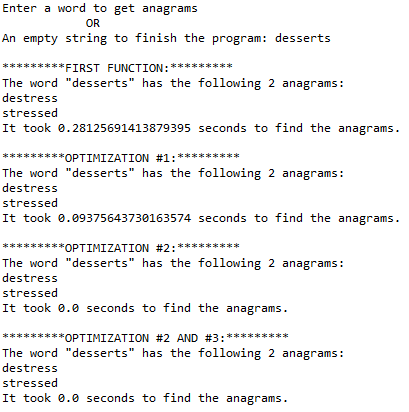
I tested with a small word that does not have anagrams. Testing “toe”:



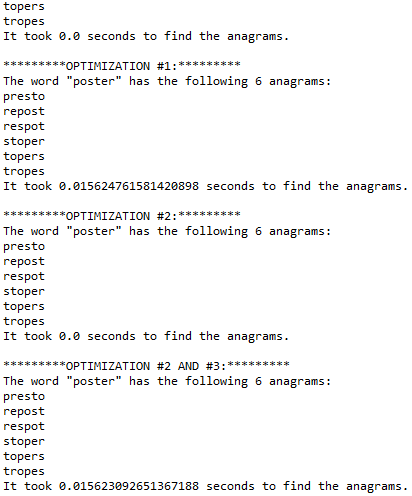
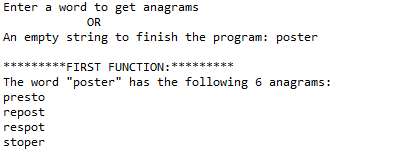
Then, a small word that does:



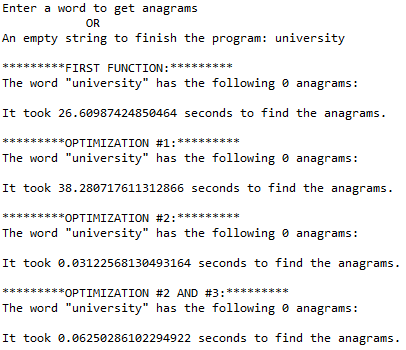
A word with many repeated letters: “desserts”



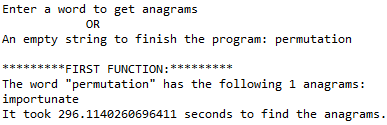
Words that were on the lab example. Testing “Poster”:



University:



Permutation:



1. **Conclusions**

The first optimization worked better on times when there were repeated letters on a word. However, when they were none, it was usually slower, probably because it had to check which added steps.

The second optimization worked wonders. The time it used to create the anagrams was much less because of the prefix set. When combined with the first, it could be both very fast or very slow.

I learned that Python has many advantages in using lists and sets. Printing and accessing an element are easier than coding all the steps. I learned how to read files, how to create sets and lists, how to compare, make for loops, print, and use time and global variables. I practiced recursion, and I can say I find it easier. I learned to reset variables that functions share (when needed). I also learned to begin labs early, so that I have enough time to think about them and ask if I do not understand. Lastly, I can say I learned patience and perseverance, because things are possible. I would do differently: be more careful about input because I did not have time to fix the use of the program with upper case. I’d change that.

1. **Appendix – Source Code**

import time

#First function

def anagram(r, s):

global wordSet

global words\_result

global name

# Base case: All letters were used

if len(r) == 0:

if s in wordSet and s != name and s not in words\_result:

words\_result.append(s)

else:

# Recursive case

# move the next letter from remaining to scrambled

for i in range(len(r)):

# The letter at index i will be scrambled

scrambled = r[i]

# Remove letter to scramble from remaining letters list

remaining = r[:i] + r[i+1:]

# Scramble letter

anagram(remaining,s + scrambled)

#Function with first optimization

def anagramOp1(r, s):

global letterSet

global wordSet

global words\_result

global name

check = str(r) + str(s)

letterSet.add(check)

# Base case: All letters were used

if len(r) == 0:

if s in wordSet and s != name:

words\_result.append(s)

else:

# Recursive case

# move the next letter from remaining to scrambled

for i in range(len(r)):

# The letter at index i will be scrambled

scrambled = r[i]

# Remove letter to scramble from remaining letters list

remaining = r[:i] + r[i+1:]

# Scramble letter

check = str(remaining) + str(s) + str(scrambled)

if (check) not in letterSet:

anagramOp1(remaining, s + scrambled)

#Function with second optimization

def anagramOp2(r, s):

global wordSet

global words\_result

global name

global prefixSet

# Base case: All letters were used

if len(r) == 0:

if s in wordSet and s != name:

words\_result.append(s)

else:

# Recursive case

# move the next letter from remaining to scrambled

if (s in prefixSet):

for i in range(len(r)):

# The letter at index i will be scrambled

scrambled = r[i]

# Remove letter to scramble from remaining letters list

remaining = r[:i] + r[i+1:]

# Scramble letter

anagramOp2(remaining, s + scrambled)

def anagramOp3(r, s):

global letterSet

global wordSet

global words\_result

global name

global prefixSet

# Base case: All letters were used

if len(r) == 0:

if s in wordSet and s != name:

words\_result.append(s)

else:

# Recursive case

# move the next letter from remaining to scrambled

if (s in prefixSet):

check = str(r) + str(s)

letterSet.add(check)

for i in range(len(r)):

# The letter at index i will be scrambled

scrambled = r[i]

# Remove letter to scramble from remaining letters list

remaining = r[:i] + r[i+1:]

# Scramble letter

check = str(remaining) + str(s) + str(scrambled)

if (check) not in letterSet:

anagramOp3(remaining, s + scrambled)

#Aux to optimization 2, makes prefix set

def getPrefixes(L):

preSet = set() #set will store prefixes

for i in range(len(L)): #words in list

for j in range(len(L[i])): #length of word

preSet.add(L[i][:j]) #adds prefixes

return preSet

#opens text file and stores it in a set

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

#creates set of prefixes from wordSet

prefixSet = getPrefixes(list(wordSet))

#creates set for scrambled letters of the word

letterSet = set()

#Boolean for menu

stay = True

#loop for menu

while stay:

#asks the user for the word

name = input('Enter a word to get anagrams\n\t\t\tOR\nAn empty string to finish the program: ')

#To exit the program

if name == (''):

stay = False

print('Bye! Thanks for using this program!')

#if input was not a real word

else:

if name not in wordSet:

print('That is not a real word. Try again.')

#To continue the program

else:

#order the string in alfabetical order so that the function prints in alfabetical orde

nameAlf = sorted(name) #stores each letter on a list

words\_result = [] # list that will store anagrams of the word

start = time.time()

anagram(nameAlf, '' ) #call to function

end = time.time()

#Show results:

print('\n\*\*\*\*\*\*\*\*\*FIRST FUNCTION:\*\*\*\*\*\*\*\*\*\nThe word "' + name + '" has the following ' + str(len(words\_result)) + ' anagrams:')

print(\*words\_result, sep = "\n")

print('It took ' + str(end-start) + ' seconds to find the anagrams.')

words\_result.clear()

#OPTIMIZATION 1

start = time.time()

anagramOp1(nameAlf, '') #call to function

end = time.time()

#Show results:

print('\n\*\*\*\*\*\*\*\*\*OPTIMIZATION #1:\*\*\*\*\*\*\*\*\*\nThe word "' + name + '" has the following ' + str(len(words\_result)) + ' anagrams:')

print(\*words\_result, sep = "\n")

print('It took ' + str(end-start) + ' seconds to find the anagrams.')

words\_result.clear() #reset

letterSet.clear() #reset

#OPTIMIZATION 2

start = time.time()

anagramOp2(nameAlf, '' ) #call to function

end = time.time()

#Show results:

print('\n\*\*\*\*\*\*\*\*\*OPTIMIZATION #2:\*\*\*\*\*\*\*\*\*\nThe word "' + name + '" has the following ' + str(len(words\_result)) + ' anagrams:')

print(\*words\_result, sep = "\n")

print('It took ' + str(end-start) + ' seconds to find the anagrams.')

words\_result.clear() #reset

#OPTIMIZATION 1 & 2

start = time.time()

anagramOp3(nameAlf, '' ) #call to function

end = time.time()

#Show results:

print('\n\*\*\*\*\*\*\*\*\*OPTIMIZATION #2 AND #3:\*\*\*\*\*\*\*\*\*\nThe word "' + name + '" has the following ' + str(len(words\_result)) + ' anagrams:')

print(\*words\_result, sep = "\n")

print('It took ' + str(end-start) + ' seconds to find the anagrams.')

words\_result.clear() #reset

letterSet.clear() #reset

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.

x- Elisa Jimenez Todd