

CS2302 Data Structures

Lab Report No. 1

Author: Bryan Ramos
Due: March 10th, 2019
Professor: Olac Fuentes
TA: Anindita Nath

Introduction

For this lab, we were asked to work with recursion and code methods that implement the concept of recursion to solve a problem. We were to read in a file of English words and calculate the anagrams for a user provided word using recursive functions and determine the length of time of calculation. The main objective of this lab was to build familiarity with Python and understanding on how recursion works, how to implement recursive functions and how to optimize recursive functions for better performance.

Proposed Solution Design & Implementation

Part 1:

For this operation, it was necessary to first create a function that accepted the name of the file containing English words as a parameter. If the file existed, the file was read in and then every word contained was added to the set. Once the entire file had been read, the set was returned for use in the other functions. If the file was not found, an error message was shown to the user so they could know the file is not in the proper location or the file name did not match the expected filename 'words_alpha.txt', for example.

Next, I created a function that holds the user interface for the program, a prompt for the user to carry on certain actions like finding anagrams or exiting the program, and a call to the recursive function. This function takes the set of words read from the file as a parameter. The user is prompted to enter a word or an empty string. If an empty string is entered, the program terminates. My function catches invalid input, that is, a user enters a word that is not in the set of words, or enters a number or symbol, displaying an error for invalid input to the user. Otherwise, the calculation of anagrams continues. Oh also, in the spirit of good user experience, if the user enters a word using uppercase letters, those letters are converted to lowercase letters so there is no conflict with the word not being found in the set of words because one version of the word is in uppercase letters and the other is lowercase letters. First, the string is split into a list of its characters, the characters are then alphabetically ordered and then joined back into a string. I figured this was one way to attempt to find the anagrams alphabetically. However, later in the function, I ensured the anagrams were printed in alphabetical order by sorting the container (set) where they were stored. To store the anagrams, I initialized an empty set that is then passed as a parameter to the recursive function. Before the recursive function is called, the start time must be recorded, then the function can be called, and when the function completes, the end time must be recorded. Besides this being a requirement, it also allowed me to observe changes in performance when optimizations were made in part 2. Then, the number of anagrams, if any, the anagrams themselves, if any, and the time it took to find the anagrams calculated by taking end time minus start time, if any, is outputted to the user.

Lastly, I wrote my recursive function which calculates the anagrams for a word. As I was doing my homework for the course, on section 2.6, I noticed a function called scramble that does something very similar to the instructions for the lab. The next day during the lecture, Professor Fuentes confirmed that. My method is an edited version of the function being explained on Zybooks section 2.6. The function takes the following parameters: the word to permute, the original word, the set of words that were read from the file, an empty set where the anagrams will be stored, and an empty string for the permutations. The recursive functions work by taking a letter from the word string, removing it from the source string and then moving it into the permutating string. Essentially, when the method is recursively called, the original word passed as a parameter will be have a character removed and that character will be added to the permutating string. The base

case is very important because it contains the conditions used to stop recursion and the conditions must be met so the anagram is added to the set of anagrams. In my function, I check for three conditions to stop recursion and add a word to the set of anagrams: all the letters from the word must have been used, the permutating string must be found in the set of words read from the file, and the permutating string must not be the original word itself. If these conditions are met, the string is added to the set of anagrams. This helped me optimize my code to prevent the original word or words not in the set of words from being calculated as valid anagrams and being returned as part of the set of anagrams, all within the recursive function.

Part 2 – First Optimization:

Part 2 consisted of making two optimizations to the program. Part 1 demonstrated the running times before making any optimizations to the recursive function. The first optimization in part 2 handles duplicate characters in a string, only making recursive calls the first time the character appears. This will avoid generating the same anagram multiple times.

For the first optimization in part 2 really the only changes that were made were in the recursive method. In order to keep track of the characters already used, I used a set to store characters that have already been used in the recursive case. When going through the recursive function there is a check, if the character that has been used already is chosen to be appended to the permutation string, that case in the for loop is terminated. Otherwise, the function continues, and the character is added to the set for already used characters.

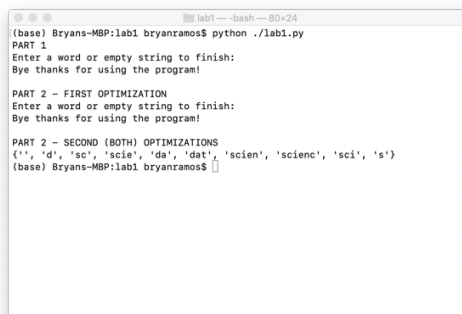
Part 2 - Second (Both) Optimizations:

Next, I implemented the second optimization as well: both optimizations. The first optimization again handles duplicate characters in a string, only making recursive calls the first time the character appears. This will avoid generating the same anagram multiple times. The second optimization stops recursion if the partial word you have is not a prefix of any word in the word set.

When I was reading the file, for part 2 with both optimizations I created a new function to read the file simply because for part 2 with both optimizations we also need a set of prefixes. Thus, I created a new read file functions that can return both the set of words and also the set of prefixes. The prefixes of each word starting from the first letter of the word to the second to last letter are added to the set of prefixes.

Example: word set is {'data', 'science'}, and the prefix set should be {'', 'd', 'da', 'dat', 's', 'sc', 'sci', 'scie', 'scien', 'scienc'}

When working on this optimization I used the example provided in the lab PDF as guidance, here is the output as I debugged and tested this function making sure it would work correctly.



```
lab1 --bash-- 80x24
(base) Bryans-MBP:lab1 bryanramos$ python ./lab1.py
PART 1
Enter a word or empty string to finish:
Bye thanks for using the program!

PART 2 - FIRST OPTIMIZATION
Enter a word or empty string to finish:
Bye thanks for using the program!

PART 2 - SECOND (BOTH) OPTIMIZATIONS
{'', 'd', 'sc', 'scie', 'da', 'dat', 'scien', 'scienc', 'sci', 's'}
(base) Bryans-MBP:lab1 bryanramos$
```

When researching things that Python allows, Python function can return multiple variables, that is powerful, and I took advantage of this when coming up with my updated read file function, instead of creating another function I returned the two sets in one function.

The function that handles my user interface now receives another parameter instead of just the set of words, the set of prefixes must also be passed so that it can be used in the call to the recursive method.

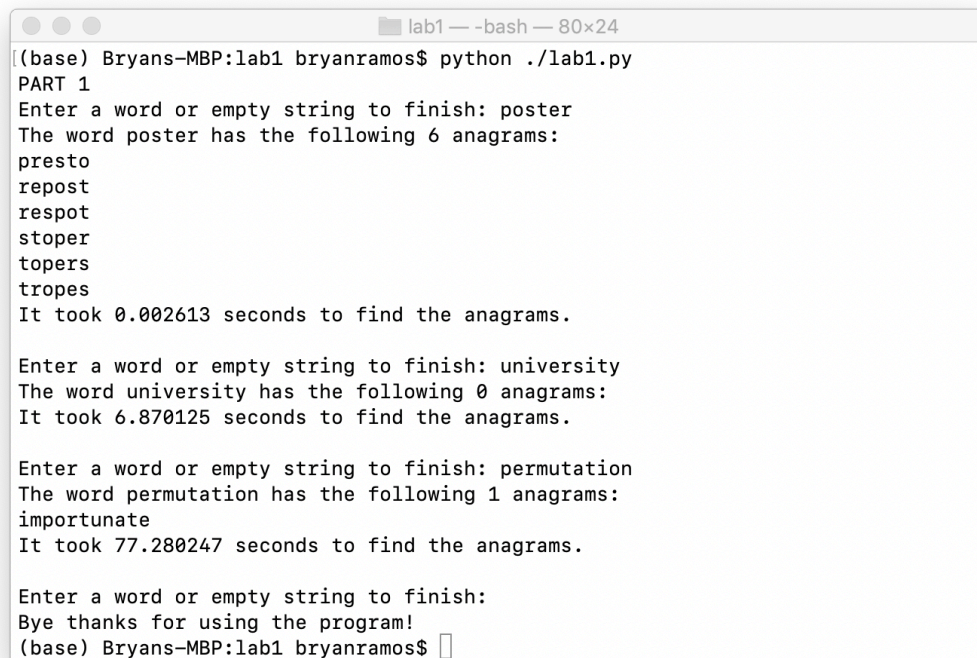
For the recursive method, I brought over the first optimization of checking already used letters. But first, the program now checks for two conditions: if not at the end of the string and the prefix is found in the set of prefixes passed, then continue recursion, otherwise, stop that iteration of the for loop. Then, we check whether the letter has already been used and if not, we add it to the set of already used letters as explained above.

Experimental Results

Part 1:

For this operation, I decided to test with words provided in the lab as sample input as well as with additional words, with an empty string and other input like numbers or symbols as well. This allowed me to check for three cases: a valid word, user input is invalid (word not in set of words or is a number or symbol), and termination by the user by providing an empty string.

Case 1 (valid input) – User enters valid input, that is, a word that is found in the set of words, and any anagrams for that word are returned. The first screen capture is the sample input from the lab PDF, the second screenshot is some testing with additional valid cases.



```
lab1 — -bash — 80x24
[(base) Bryans-MBP:lab1 bryanramos$ python ./lab1.py
PART 1
Enter a word or empty string to finish: poster
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.002613 seconds to find the anagrams.

Enter a word or empty string to finish: university
The word university has the following 0 anagrams:
It took 6.870125 seconds to find the anagrams.

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams:
importunate
It took 77.280247 seconds to find the anagrams.

Enter a word or empty string to finish:
Bye thanks for using the program!
(base) Bryans-MBP:lab1 bryanramos$
```

```
lab1 — -bash — 80x24
((base) Bryans-MBP:lab1 bryanramos$ python ./lab1.py
PART 1
Enter a word or empty string to finish: course
The word course has the following 3 anagrams:
cerous
crouse
source
It took 0.002567 seconds to find the anagrams.

Enter a word or empty string to finish: food
The word food has the following 0 anagrams:
It took 0.0001 seconds to find the anagrams.

Enter a word or empty string to finish: taco
The word taco has the following 1 anagrams:
coat
It took 0.000106 seconds to find the anagrams.

Enter a word or empty string to finish:
Bye thanks for using the program!
(base) Bryans-MBP:lab1 bryanramos$
```

Case 2 (invalid input) – User enters a string that is not a valid word (i.e. gibberish), or is a number or symbol.

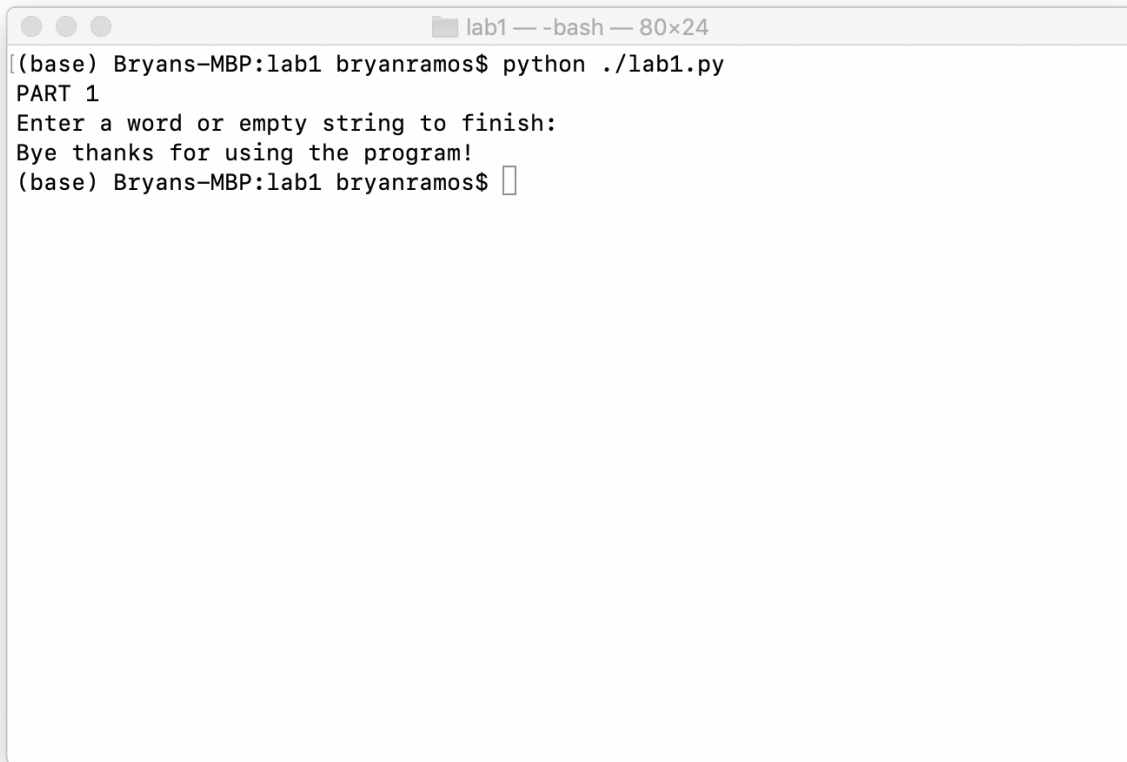
```
lab1 — -bash — 80x24
((base) Bryans-MBP:lab1 bryanramos$ python ./lab1.py
PART 1
Enter a word or empty string to finish: 123
123 is not a valid word. Please enter a valid word.

Enter a word or empty string to finish: %
% is not a valid word. Please enter a valid word.

Enter a word or empty string to finish: dfgdf
dfgdf is not a valid word. Please enter a valid word.

Enter a word or empty string to finish:
Bye thanks for using the program!
(base) Bryans-MBP:lab1 bryanramos$
```

Case 3 (empty string) – User enters an empty string and the program displays a goodbye message and terminates as seen in the terminal below.

A terminal window titled 'lab1 — -bash — 80x24' showing the execution of a Python script. The prompt is '(base) Bryans-MBP:lab1 bryanramos\$'. The user enters 'python ./lab1.py'. The output is 'PART 1', 'Enter a word or empty string to finish:', and 'Bye thanks for using the program!'. The prompt returns to '(base) Bryans-MBP:lab1 bryanramos\$' with a cursor.

```
lab1 — -bash — 80x24
(base) Bryans-MBP:lab1 bryanramos$ python ./lab1.py
PART 1
Enter a word or empty string to finish:
Bye thanks for using the program!
(base) Bryans-MBP:lab1 bryanramos$
```

Based on the results for part 1, if the user inputs longer strings, calculation time will take much longer.

Part 2 - First Optimization:

There was an impact on calculation time from part 1 with no optimization to part 2 with one optimization. The calculation times for words like 'university' and 'permutation' was considerably lower.


```

lab1 — lab1.py — 79x21
PART 2 - FIRST OPTIMIZATION
Enter a word or empty string to finish: poster
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.003135 seconds to find the anagrams.

Enter a word or empty string to finish: university
The word university has the following 0 anagrams:
It took 4.248484 seconds to find the anagrams.

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams:
importunate
It took 45.934852 seconds to find the anagrams.

Enter a word or empty string to finish: 

```

Here is a side-by-side comparison of the sample input from the lab for part 1 program and part 2 with 1 optimization program.

```

lab1 — -bash — 80x24
(base) Bryans-MBP:lab1 bryanramos$ python ./lab1.py
PART 1
Enter a word or empty string to finish: poster
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.002613 seconds to find the anagrams.

Enter a word or empty string to finish: university
The word university has the following 0 anagrams:
It took 6.870125 seconds to find the anagrams.

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams:
importunate
It took 77.280247 seconds to find the anagrams.

Enter a word or empty string to finish:
Bye thanks for using the program!
(base) Bryans-MBP:lab1 bryanramos$ 

```

```

lab1 — lab1.py — 79x21
PART 2 - FIRST OPTIMIZATION
Enter a word or empty string to finish: poster
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.003135 seconds to find the anagrams.

Enter a word or empty string to finish: university
The word university has the following 0 anagrams:
It took 4.248484 seconds to find the anagrams.

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams:
importunate
It took 45.934852 seconds to find the anagrams.

Enter a word or empty string to finish: 

```

Part 2 - Second (Both) Optimizations:

```

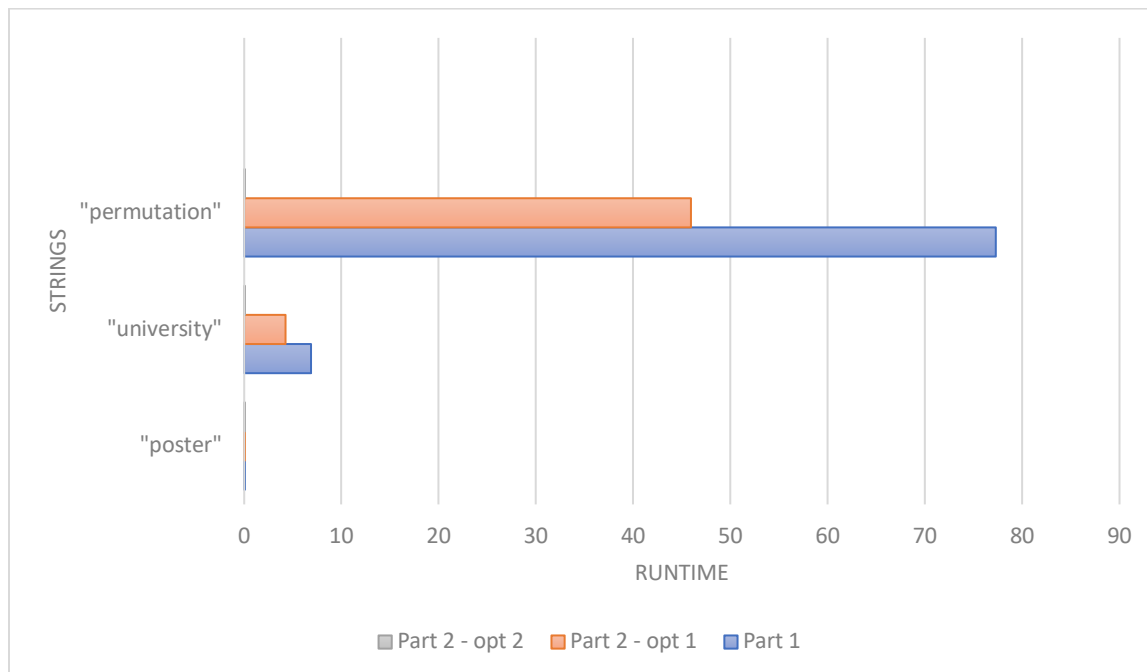
lab1 — lab1.py — 79x21
PART 2 - SECOND (BOTH) OPTIMIZATIONS
Enter a word or empty string to finish: poster
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.00108 seconds to find the anagrams.

Enter a word or empty string to finish: university
The word university has the following 0 anagrams:
It took 0.008083 seconds to find the anagrams.

Enter a word or empty string to finish: permutation
The word permutation has the following 1 anagrams:
importunate
It took 0.030363 seconds to find the anagrams.

Enter a word or empty string to finish: 

```



As shown by the times outputted to the user and the graph above, the optimizations for part 2 bring down calculation for large strings like 'university' or 'permutation' significantly. The calculating time for the word 'permutation' dropped by **99.96%** when the two optimizations together were made.

Conclusion

Personally, this lab really helped me in familiarizing myself with Python. In the past, I've written a lot of JavaScript and Java code and this summer I wrote a lot of C# at my internship. I knew this course was going to be taught in Python and these past few weeks as I've been learning Python every once in a while I still think I'm writing Java or C#, I'll try to use braces or declare the variable type or use else if instead of elif, so it's been taking a little getting used to.

With this lab, I had a great way to challenge myself to memorize Python's syntax rules and I'm really liking some of the features of Python like returning multiple variables in a function and the fact that the syntax is a lot more lenient.

Recursive functions can be very costly when it comes to runtime if not optimized correctly. However, when I made the optimizations to my recursive function, I was really surprised how much the runtime changed, especially after the prefix set optimization, calculating anagrams for big words like permutation had a 99.96% decrease in time. That's amazing!

Appendix

```
'''
    Course: CS2302 Data Structures Fall 2019
    Author: Bryan Ramos [88760110]
    Assignment: Lab 1 Recursion
    Instructor: Dr. Olac Fuentes
    TA: Anindita Nath
    Last Modified: Sep 6 2019
    Purpose: Find the anagrams for a word provided by the user using
    recursion.
'''

import time as t

# ----- PART 1 -----

# Read in file
# parameter filename: The name of the file to read.
# returns: set of words from file
def readFile1(filename):
    # attempt to open the file
    try:
        file = open(filename, "r")
        # each word is saved as a value in the set
        setOfWords = set(file.read().split())
        return setOfWords
    # if file not found, catch file not found exception and output an
    error
    except IOError:
        print("Error: File", filename, "not found!")
```

```

def findAnagrams1(r_letters, originalWord, setOfWords, setOfAnagrams,
s_letters = ""):
    # all the letters have been used in our base case
    if len(r_letters) == 0 and s_letters in setOfWords and originalWord
!= s_letters:
        setOfAnagrams.add(s_letters)
    # recursive case
    else:
        for i in range(len(r_letters)):
            # move a letter from r_letters to scrambled letters, remove
it from remaining letters
            scrambleLetter = r_letters[i]
            remainingLetters = r_letters[:i] + r_letters[i + 1:]
            scrambled = s_letters + scrambleLetter

            findAnagrams1(remainingLetters, originalWord, setOfWords,
setOfAnagrams, scrambled)

# Display prompt for user
# parameter setOfWords: Set of the words from the file.
def prompt1(setOfWords):
    print("PART 1")
    # the prompt will instruct the user to do one of the following:
    # (1) enter a word, the anagrams for that word will be found, then
the prompt is reshown
    # (2) enter an empty string and the program will terminate
    while (True):
        word = input("Enter a word or empty string to finish: ").lower()

        # if no word is provided: terminate the program
        if not word:
            print("Bye thanks for using the program!")
            break
        # else if: input provided is not a valid word or a number
        elif word not in setOfWords:
            print(word, "is not a valid word. Please enter a valid
word.")
        # else: find the anagrams for that word
        else:
            # split a word into a list of its characters, organize them
alphabetically
            splitWord = sorted(list(word))
            # take the ordered list of characters and join them into a
string
            splitWord = "".join(splitWord)

            # init empty set to store the returned set of anagrams
            setOfAnagrams = set()

            start = t.time()
            findAnagrams1(splitWord, word, setOfWords, setOfAnagrams, "")
            end = t.time()

```

```

        print("The word", word, "has the following",
len(setOfAnagrams), "anagrams:")
        # sort anagrams in alphabetical order
        setOfAnagrams = sorted(list(setOfAnagrams))
        # print each anagram
        for anagram in setOfAnagrams:
            print(anagram)
        print("It took", round(end-start, 6), "seconds to find the
anagrams.\n")

# ----- PART 2 -----

# ----- FIRST OPTIMIZATION -----

# If the string has duplicate characters, only make recursive calls the
first time the character\
# appears. This will avoid generating the same anagram multiple times.
def findAnagrams2_opt1(r_letters, originalWord, setOfWords,
setOfAnagrams, s_letters = ""):
    # all the letters have been used in our base case
    if len(r_letters) == 0 and s_letters in setOfWords and originalWord
!= s_letters:
        setOfAnagrams.add(s_letters)
    # recursive case
    else:
        lettersAlreadyUsed = set()
        for i in range(len(r_letters)):
            # move a letter from r_letters to scrambled letters, remove
it from remaining letters
            scrambleLetter = r_letters[i]
            remainingLetters = r_letters[:i] + r_letters[i + 1:]

            if scrambleLetter in lettersAlreadyUsed:
                continue
            lettersAlreadyUsed.add(scrambleLetter)
            scrambled = s_letters + scrambleLetter

            findAnagrams2_opt1(remainingLetters, originalWord,
setOfWords, setOfAnagrams, scrambled)

# Display prompt for user
# parameter setOfWords: Set of the words from the file.
def prompt2_forOpt1(setOfWords):
    print("\nPART 2 - FIRST OPTIMIZATION")

    # the prompt will instruct the user to do one of the following:
    # (1) enter a word, the anagrams for that word will be found, then
the prompt is reshown
    # (2) enter an empty string and the program will terminate
    while (True):
        word = input("Enter a word or empty string to finish: ").lower()

        # if no word is provided: terminate the program
        if not word:

```

```

        print("Bye thanks for using the program!")
        break
    # else if: input provided is not a valid word or a number
    elif word not in setOfWords:
        print(word, "is not a valid word. Please enter a valid
word.")
    # else: find the anagrams for that word
    else:
        # split a word into a list of its characters, organize them
        alphabetically
        splitWord = sorted(list(word))
        # take the ordered list of characters and join them into a
        string
        splitWord = "".join(splitWord)

        # init empty set to store the returned set of anagrams
        setOfAnagrams = set()

        start = t.time()
        findAnagrams2_opt1(splitWord, word, setOfWords,
setOfAnagrams, "")
        end = t.time()

        print("The word", word, "has the following",
len(setOfAnagrams), "anagrams:")
        # sort anagrams in alphabetical order
        setOfAnagrams = sorted(list(setOfAnagrams))
        # print each anagram
        for anagram in setOfAnagrams:
            print(anagram)
        print("It took", round(end-start, 6), "seconds to find the
anagrams.\n")

# ----- SECOND (BOTH) OPTIMIZATIONS -----

# Read in file
# parameter filename: The name of the file to read.
# returns: set of words from file and the set of prefixes
def readFile2(filename):
    # attempt to open the file
    try:
        file = open(filename, "r")
        # set will contain the prefixes of all the words in the word set
        setOfPrefixes = set()
        # each word is saved as a value in the set
        setOfWords = set(file.read().split())
        for word in setOfWords:
            # i.e. word set is {'data', 'science'}, your prefix set
            should
            # be {"", 'd', 'da', 'dat', 's', 'sc', 'sci', 'scie', 'scien',
            # 'scienc'}
            for i in range(len(word)):
                setOfPrefixes.add(word[:i])
        return setOfWords, setOfPrefixes

```

```

    # if file not found, catch file not found exception and output an
    error
    except IOError:
        print("Error: File", filename, "not found!")

# If the string has duplicate characters, only make recursive calls the
first time the character\
# appears. This will avoid generating the same anagram multiple times.
# Stop recursion if the partial word you have is not a prefix of any word
in the word set.
def findAnagrams2_opt2(r_letters, originalWord, setOfWords,
setOfPrefixes, setOfAnagrams, s_letters = ""):
    # all the letters have been used in our base case
    if len(r_letters) == 0 and s_letters in setOfWords and originalWord
!= s_letters:
        setOfAnagrams.add(s_letters)
    # recursive case
    else:
        lettersAlreadyUsed = set()
        for i in range(len(r_letters)):
            # move a letter from r_letters to scrambled letters, remove
it from remaining letters
            scrambleLetter = r_letters[i]
            remainingLetters = r_letters[:i] + r_letters[i + 1:]

            if (len(remainingLetters) != 0) and not ((s_letters +
scrambleLetter) in setOfPrefixes):
                continue
            if scrambleLetter in lettersAlreadyUsed:
                continue

            lettersAlreadyUsed.add(scrambleLetter)
            scrambled = s_letters + scrambleLetter

            findAnagrams2_opt2(remainingLetters, originalWord,
setOfWords, setOfPrefixes, setOfAnagrams, scrambled)

# Display prompt for user
# parameter setOfWords: Set of the words from the file.
# parameter setOfPrefixes: Set of prefixes
def prompt2_forOpt2(setOfWords, setOfPrefixes):
    print("\nPART 2 - SECOND (BOTH) OPTIMIZATIONS")

    # the prompt will instruct the user to do one of the following:
    # (1) enter a word, the anagrams for that word will be found, then
the prompt is reshown
    # (2) enter an empty string and the program will terminate
    while (True):
        word = input("Enter a word or empty string to finish: ").lower()

        # if no word is provided: terminate the program
        if not word:
            print("Bye thanks for using the program!")

```



```

        break
        # else if: input provided is not a valid word or a number
    elif word not in setOfWords:
        print(word, "is not a valid word. Please enter a valid
word.")
    # else: find the anagrams for that word
    else:
        # split a word into a list of its characters, organize them
        alphabetically
        splitWord = sorted(list(word))
        # take the ordered list of characters and join them into a
        string
        splitWord = "".join(splitWord)

        # init empty set to store the returned set of anagrams
        setOfAnagrams = set()

        start = t.time()
        findAnagrams2_opt2(splitWord, word, setOfWords,
setOfPrefixes, setOfAnagrams, "")
        end = t.time()

        print("The word", word, "has the following",
len(setOfAnagrams), "anagrams:")
        # sort anagrams in alphabetical order
        setOfAnagrams = sorted(list(setOfAnagrams))
        # print each anagram
        for anagram in setOfAnagrams:
            print(anagram)
        print("It took", round(end-start, 6), "seconds to find the
anagrams.\n")

if __name__ == '__main__':
    # Part 1 of the lab assignment
    setOfWords = readFile1("words_alpha.txt")
    prompt1(setOfWords)

    # Part 2 of the lab assignment with first optimization
    prompt2_forOpt1(setOfWords)

    # Part 2 of the lab assignment with second (both) optimizations
    setOfWords, setOfPrefixes = readFile2("words_alpha.txt")
    prompt2_forOpt2(setOfWords, setOfPrefixes)

```

I, Bryan Ramos, 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.