CS 2302 Data Structures – Fall 2019

Lab Report #1

- Professor: Olac Fuentes
- TA: Anindita Nath

Sergio Ortiz
9-5-2019

Introduction

For this lab, we were asked to find all the anagrams from a word that the user provided. However, the lab specified that we should use recursion to scramble the word into all its n! permutations. Once we have all the permutations, we check which of these is an anagram by comparing them to a set containing most of the common English words, if any are found in the set, then we add them to an anagrams list.

Proposed Solution Design and Implementation

Part #1:

To begin implementing part one, I decided to focus on a method that would return all the permutations of a given word using recursion. This turned out to be quite easy as the book already had an implementation for us. After this, I wrote a method that would read the data off a file containing most of the common English words and added them to a set as we would need to compare to them later. From here, it was simply a matter of getting the permutations from a given words and seeing which of them could be found in the set. To make this operation cleaner, I decided to give it its own method, one which would take in a list of permutations, a set of words, and see which of the permutations could be found in the list of words, returning a list containing all the matches. After the list is generated, I pass it through a precautionary 'remove duplicates' method. This is to avoid duplicates in the list since the lab instructions specified this. Finally, the list of anagrams is returned and printed out to the user, showing him/her how much time it took to find them. To control the flow of the program, I added a while loop around the main chunk of code that would continue running as long as the user's input was not an empty string. This would allow the user to run the program multiple times and finally hit enter when they were ready to close the program.

Part #2:

Part Two asked for us to implement some modifications to optimize the runtime of our program. The first one stated that our program should only call the recursive step once if the same character could be found two times in a string. To do this, I created a set 'seen' before the recursive step of my scrambler method. Before the recursive step would run, the program would check if the current character was not in seen, and if so, would add it to seen and run the recursive functions. If the character was in seen, then the program would skip the recursive functions on it. The next optimization asked for us to call off the recursion if the partial word we had was not a prefix of any of the words in the words set. I am a bit disappointed with my implementation of this because, although it works, it takes a while to run averaging around 15–20 seconds. The first step I took was making a method that would return all the prefixes of a word in a list. Then, I created a method that would call the previous method for each word in the

words set and added all the prefixes to a different set. After it was done, it would return a set with all the prefixes of the word set. As we can see, this would take quite a while. Nonetheless, I used this set in the following way. Before my program ran the recursive call, it would check if the current scrambled word was found in the prefixes set, and if not, it would skip the recursive call. This is how I implemented the optimizations for part 2.

Experimental Results

Part #1:

First, I created the scrambler method, which would return the permutations of a given word. Given that I had closely studied the book's implementation of this method, I could not get it out of my mind as I worked on my own implementation and ended up doing basically the same thing.

```
1   def scrambler(remaining, scrambled):
2       if(len(remaining) == 0):
3           scrambled_words.append(scrambled)
4       else:
5           for letter in range(len(remaining)):
6               scramble_letters = remaining[letter]
7               remaining_letters = remaining[:letter] + remaining[letter + 1:]
8               scrambler(remaining_letters, scrambled + scramble_letters)
9
10
11  scrambled_words = []
12  scrambler("cat", "")
13  for word in scrambled_words:
14      print(word)
```

This code produces the following output.

```
cat
cta
act
atc
tca
tac
```

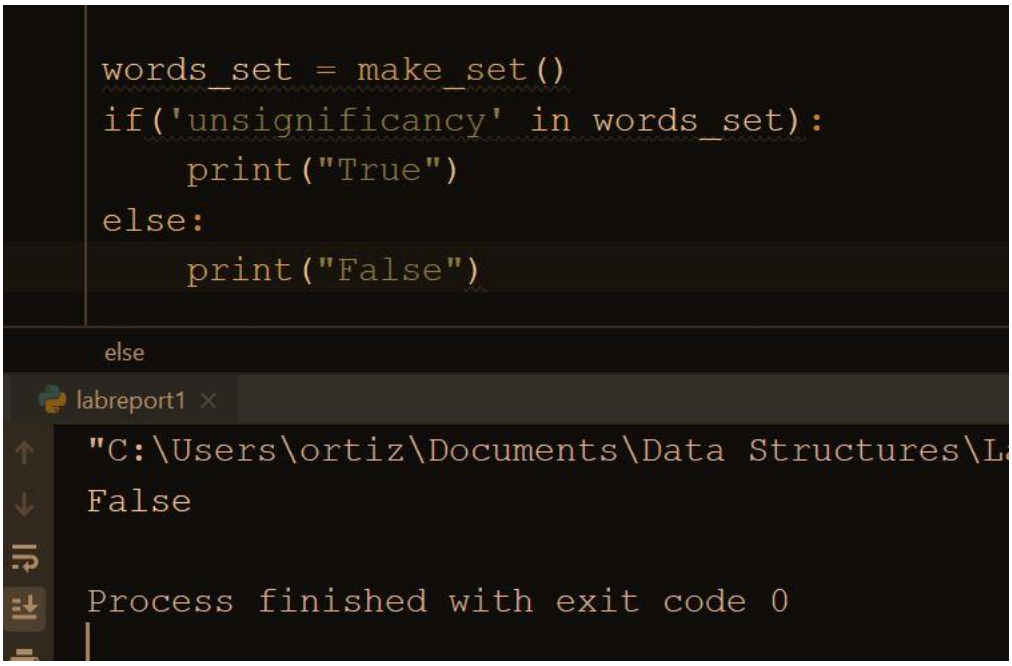
After this I focused on writing some code that would read the words off the file and added them to a set.

```
def make_set():
    english_words_file = open("words_alpha.txt", "r")
    words_set = set()

    for word in english_words_file:
        words_set.add(word)
    english_words_file.close()
    return words_set

words_set = make_set()
print(words_set)
```

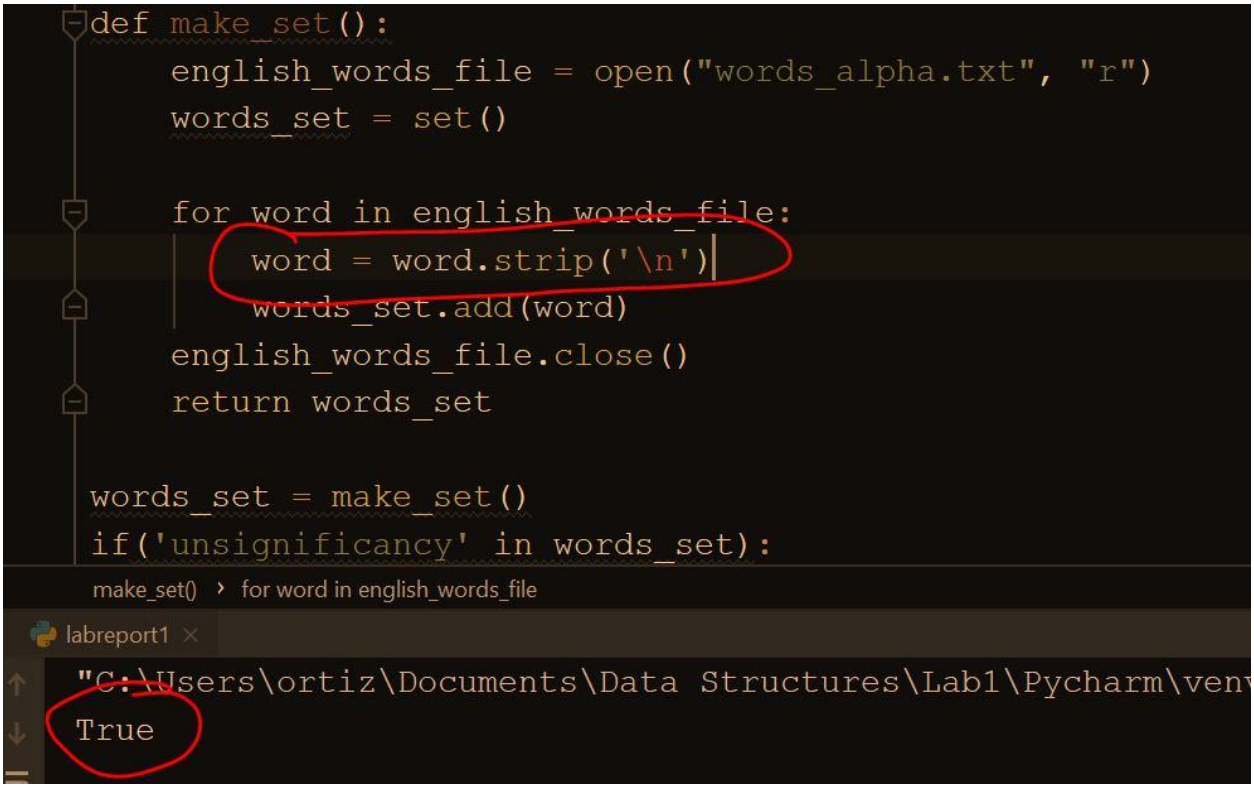
Here we can see just a tiny snippet of all the words in that set.

```
›holden\n', 'unsignificancy\n', 'nestlings\n', 'despisedness\n', 'exceptant\n',
```

However, at this point I came upon the biggest stumbling block of the project for me. Notice the word ‘unsignificancy’ in the above output. This word is already in the set, so if I checked for it my program should be able to find it.



As we can see, my program cannot find the word which we know is in the set. After a lot of frustration, I was finally able to see why. For some strange reason Python adds the newline character ‘\n’ to the words which completely changes their value. Stripping each word of this newline character ended up doing the trick for me.



After having these two methods, the problem could be solved easily, merely by checking which of the permutations of the given word were in the word's set. To do this, I created a method which would take in a list with the permutations, a set with all the words, and an empty list to add the anagrams that we would find.

```
def anagram_finder(scrambled_words ,words_set, anagram_list):
    for word in scrambled_words:
        if(word in words_set):
            anagram_list.append(word)
    anagram_list.sort()
    return anagram_list
```

Not much here besides the fact that I did not know Python already had a 'sort' function and was about to create my own sort method to add style to the program. Thankfully, I came upon it by accident due to PyCharm's autofill abilities. Here is a run of this method with sample data.

```
words_set = make_set()
scrambled_words = []
scrambler("poster", "")
anagrams = []
anagrams = anagram_finder(scrambled_words,words_set,anagrams)
print(anagrams)
```

```
anagram_finder() › for word in scrambled_words › if (word in words_set)
abreport1 ×
"C:\Users\ortiz\Documents\Data Structures\Lab1\Pycharm\venv\Scripts\python.exe
['poster', 'presto', 'repost', 'respot', 'stoper', 'topers', 'tropes']
```

It works fine, but as we can see the original word is included in the list. Also, at one point I was getting duplicates, so I decided to fix both issues. For the first issue, I simply used the 'remove' function and passed in the original word. For the second issue, I created a method which would remove duplicates by adding all the elements of the list to a set and then add the elements of the set back to the list. This step might have been a bit extra, but I saw a duplicate and decided to take the necessary steps.

```
def remove_duplicates(list):
    duplicate_remover = set()
    for word in list:
        duplicate_remover.add(word)

    new_list = []
    for word in duplicate_remover:
        new_list.append(word)

    return new_list
```

```
['presto', 'repost', 'respot', 'stoper', 'topers', 'tropes']
```

At this point the program was done and it was time to make it user friendly. First, I imported the 'time' function to keep track of how much time the program took to find the anagrams. Then I added a variable 'user_word' which would store a word the user entered. I added a while loop which would keep running until the user entered an empty string and added all the calls to my method in between here. Finally, I added code that would display how many anagrams the word had, what those anagrams were, and how long it took to find them. Something quick to notice is that my program would crash when I would input unconventional words such as 'lol', so to fix this problem I checked if the user's word was in the set of the most common English words before running the main functions

```
import time
```

```
user_word = input("Enter a word or empty string to finish ")

while(user_word != ""):
    if(user_word in words_set):
        scrambled_words = []
        start = time.time()
        scrambler(user_word, "", prefixes_set)
        anagrams_list = []
        anagrams_list = anagram_finder(scrambled_words, words_set, anagrams_list)
        end = time.time()
        total_time = end - start

        anagrams_list.remove(user_word)
```

```
        print("The word "+user_word+" has the following "+str(len(anagrams_list))+" anagrams:")
        for word in anagrams_list:
            print(word)
        print("It took "+str(total_time)+" seconds to find the anagrams")
        user_word = input("Enter a word or empty string to finish ")
    else:
        user_word = input("I am sorry, we cannot recognize that word, try again ")
print("Bye, thanks for using this program!")
```

Here is a sample run of the program

```
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 0.002974987030029297 seconds to find the anagrams
Enter a word or empty string to finish
Bye, thanks for using this program!
```

Note that there is an extra parameter in my scrambler call ‘prefixes_set’. I will explain this in the next section where I show my implementations of part two.

Part #2:

To implement the first optimization, in which the program would only make a recursive call once if duplicate characters were found in the string, I made a set ‘seen’. Every character

that was seen would be added to this set, and the program would check if the current character was new before running the recursive step.

```
def scrambler(remaining, scrambled):
    if(len(remaining) == 0):
        scrambled_words.append(scrambled)
    else:
        seen = set()
        for letter in range(len(remaining)):
            if(letter not in seen):
                seen.add(letter)
                scramble_letters = remaining[letter]
                remaining_letters = remaining[:letter] + remaining[letter + 1:]
                scrambler(remaining_letters, scrambled + scramble_letters)
```

My implementation of the next optimization is my biggest inconformity with the program. The optimization stated that we should check if the current partial word we had was not a prefix of any word in the word set, and if so, call off the recursion. To do this, I created a method that would return all the prefixes of any string.

```
def find_prefixes(string):
    prefixes_list = []
    for i in range(len(string)):
        prefixes_list.append(string[:i])
    return prefixes_list

print(find_prefixes("science"))
```

```
labreport1 ×
"C:\Users\ortiz\Documents\Data Structures\Lab1\Pycharm\v
['', 's', 'sc', 'sci', 'scie', 'scien', 'scienc']
```

Then I created a method that would take all the words in the words set and pass them through the prefix finder method, creating a new set with all the prefixes in the word set.

```
def make_prefix_set(words_set):
    prefix_set = set()
    temp_list = []
    for word in words_set:
        temp_list = find_prefixes(word)
        for prefix in temp_list:
            prefix_set.add(prefix)
    return prefix_set
```

Before running the recursive step, the program checks if the partial word can be found in this set of prefixes.

```
def scrambler(remaining, scrambled, prefixes_set):
    if(len(remaining) == 0):
        scrambled_words.append(scrambled)
    else:
        seen = set()
        for letter in range(len(remaining)):
            if(letter not in seen):
                seen.add(letter)
                if(scrambled in prefixes_set):
                    scramble_letters = remaining[letter]
                    remaining_letters = remaining[:letter] + remaining[letter + 1:]
                    scrambler(remaining_letters, scrambled + scramble_letters, prefixes_set)
```

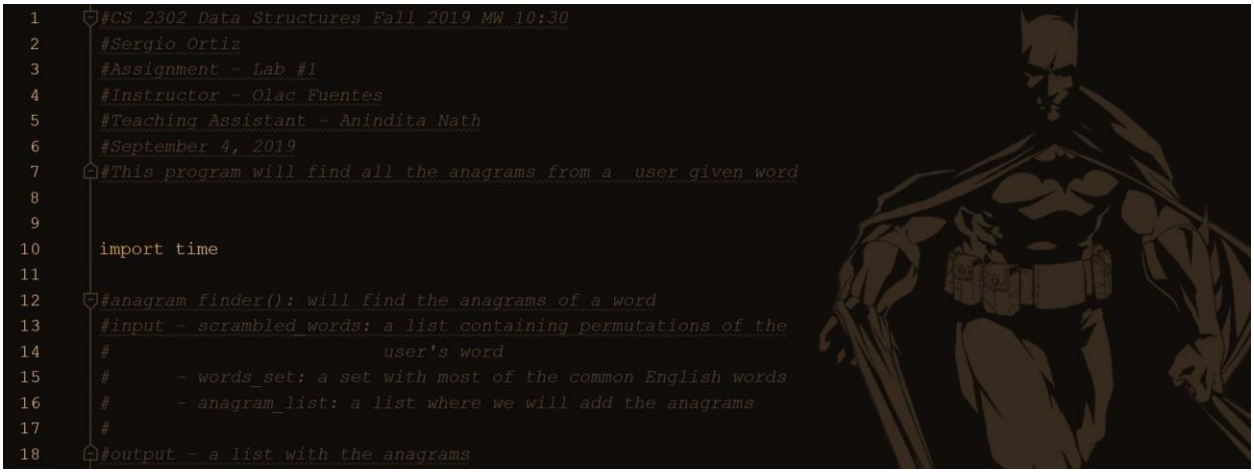
The implementation works, but I run the process of creating a set of all the prefixes before asking the user to enter a word and we can see that it takes around 20 seconds for the prompt to enter a word appears. I thought perhaps of changing my prefix finder method to return a set with the prefixes instead of a list and then simply adjoining this set with the set of all the prefixes but decided not to do it as I thought it would not do much of a difference. Nonetheless, the code optimizes the runtime of finding anagrams, but you must wait a while for it to boot up like an old Windows 7 computer.

```
Enter a word or empty string to finish university
The word university has the following 0 anagrams:
It took 0.12201857566833496 seconds to find the anagrams
Enter a word or empty string to finish
Bye, thanks for using this program!
```

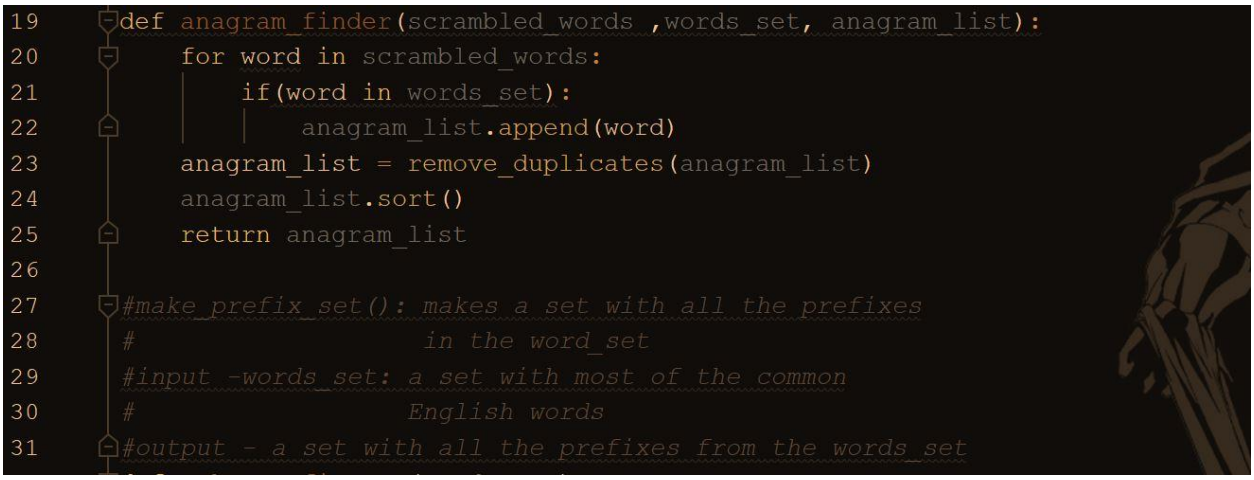
Conclusion

The biggest takeback from this project is the process of not only implementing a recursive function but also of optimizing one. Due to the recursive nature of these methods, we can have very suboptimal solutions to certain problems if we do not carefully check the behavior of the method. I found it very interesting how adding a few codes of line to check a few simple conditions could drastically improve the runtime of the program. I learned that sometimes we could get caught up in the elegance of a recursive method and not realize that we are doing extra work, perhaps adding exponential runtime to our program.

Appendix



```
#CS 2302 Data Structures Fall 2019 MW 10:30
#Sergio Ortiz
#Assignment - Lab #1
#Instructor - Olac Fuentes
#Teaching Assistant - Anindita Nath
#September 4, 2019
#This program will find all the anagrams from a  user given word


import time

#anagram finder(): will find the anagrams of a word
#input - scrambled_words: a list containing permutations of the
#                         user's word
#      - words_set: a set with most of the common English words
#      - anagram_list: a list where we will add the anagrams
#
#output - a list with the anagrams
```



```
def anagram_finder(scrambled_words ,words_set, anagram_list):
    for word in scrambled_words:
        if(word in words_set):
            anagram_list.append(word)
    anagram_list = remove_duplicates(anagram_list)
    anagram_list.sort()
    return anagram_list


#make_prefix_set(): makes a set with all the prefixes
#                   in the word_set
#input -words_set: a set with most of the common
#                  English words
#output - a set with all the prefixes from the words_set
```

```
32  def make_prefix_set(words_set):
33      prefix_set = set()
34      temp_list = []
35      for word in words_set:
36          temp_list = find_prefixes(word)
37          for prefix in temp_list:
38              prefix_set.add(prefix)
39      return prefix_set
40
41  #make_set(): will make a set with most of the common
42   #           english words by reading them off a file
43   #           and adding them to a set
44   #input - none
45  #output - a set with English words
```

```
46  def make_set():
47      english_words_file = open("words_alpha.txt", "r")
48      words_set = set()
49
50      for word in english_words_file:
51          word = word.strip('\n')
52          words_set.add(word)
53      english_words_file.close()
54      return words_set
55
56  #find_prefixes(): will find all the prefixes of a word
57   #input -string: a word
58  #output - a list with all the prefixes of the given word
59  def find_prefixes(string):
60      prefixes_list = []
61      for i in range(len(string)):
62          prefixes_list.append(string[:i])
63      return prefixes_list
64
```

```
65  #remove_duplicates(): will remove duplicates from a list
66   #                    by adding all the elements to a set
67   #                    then returning them back to the list
68   #input - list: a list with words, might contain duplicates
69  #output - list: a list without any duplicate words
70  def remove_duplicates(list):
71      duplicate_remover = set()
72      for word in list:
73          duplicate_remover.add(word)
74
75      new_list = []
76      for word in duplicate_remover:
77          new_list.append(word)
78
79      return new_list
```

```
#scrambler(): will take a word and return its permutations
#input -remaining: a list with the remaining words (not scrambled)
#      -scrambled: a list with the scrambled words
#output -none, but the scrambled words are appended to a list
#        scrambled_words
def scrambler(remaining, scrambled, prefixes_set):
    if(len(remaining) == 0):
        scrambled_words.append(scrambled)
    else:
        seen = set()
        for letter in range(len(remaining)):
            if(letter not in seen):
                seen.add(letter)
                if(scrambled in prefixes_set):
                    scramble_letters = remaining[letter]
                    remaining_letters = remaining[:letter] + remaining[letter + 1:]
                    scrambler(remaining_letters, scrambled + scramble_letters, prefixes_set)
```

```
words_set = make_set()
prefixes_set = make_prefix_set(words_set)
user_word = input("Enter a word or empty string to finish ")

while(user_word != ""):
    if(user_word in words_set):
        scrambled_words = []
        start = time.time()
        scrambler(user_word, "", prefixes_set)
        anagrams_list = []
        anagrams_list = anagram_finder(scrambled_words, words_set, anagrams_list)
        end = time.time()
        total_time = end - start

        anagrams_list.remove(user_word)
```

```

        print("The word "+user_word+" has the following "+str(len(anagrams_list))+" anagrams:")
        for word in anagrams_list:
            print(word)
        print("It took "+str(total_time)+" seconds to find the anagrams")
        user_word = input("Enter a word or empty string to finish ")
    else:
        user_word = input("I am sorry, we cannot recognize that word, try again ")
print("Bye, thanks for using this program!")

```

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.