**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #1**

Due: September 10th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to work with recursion and code separate recursive functions. For this lab, it is key to remember the basics of recursion and the properties of sets since both are required for all of the exercises. The main objective of this lab is to write a program that asks a user to input a word, implements recursion to find all the anagrams of that word, and prints them.

**Proposed Solution Design and Implementation**

**Operation #1:**

For this operation, I implemented a recursive function that accepts a valid word from the user and stores all of the anagrams of that word in a set. To do that, the function creates all of the possible permutations of that word by making recursive calls that move a letter from the word to a permutation. The rest of the word is passed to each recursive call, and recursion continues until a permutation is formed. To find the anagrams, I set up the base case so that it checks if a permutation is a valid anagram. To do this, I created a set of valid words in the English language by reading a file that contained those words and storing them in that set. If a permutation is found in the set of valid English words, it is considered a valid anagram, and it is stored in the set of anagrams that will be printed to the user at a later point in the program.

**Operation #2:**

For this operation, I optimized the recursive function from the previous operation so that if the word has duplicate letters, the recursive call will only be made the first time the letter appears. That way, the function avoids creating the same permutation multiple times. I implemented this optimization by adding a conditional test to the recursive case. This conditional test checks if the letter of the word at index i is not found in the rest of the word. If the test evaluates to True, then recursion will proceed as usual. If it evaluates to False, then the function will move on to the next letter of the word and resume recursion.

**Operation #3:**

For this operation, I further optimized the recursive function so that recursion stops if a partial permutation is not a prefix of any word in the set of valid English words. That way, the function avoids making recursive calls for permutations that will end up not being considered valid anagrams once they reach the base case. I implemented this optimization by creating a set of the prefixes of all the words in the set of valid English words. Then, I added another conditional test to the recursive case. This conditional test checks if the partial permutation is in the set of valid prefixes. If the test evaluates to True, then recursion will proceed as usual. If it evaluates to False, then the function will not make a recursive call for that partial permutation, Instead, it will move on to the next letter of the word and resume recursion.

Once the program runs correctly, the user is asked to select which of the three implementations they would like to use. If the user enters invalid input for their selection, then they will be asked to run the program again, and the program will end. Otherwise, they are prompted to enter a word or an empty string to end the program. If the user enters invalid input for their word (i.e. a string of characters that is not found in the set of valid English words), then they will be prompted to enter a valid word. Once the user has entered a valid word, the program generates a set of anagrams of that word based on the optimizations the user selected at the start of the program. After removing the user’s word from the set of anagrams, the program displays the number of anagrams that word has. Furthermore, it sorts the set of anagrams alphabetically and prints the anagrams in that order. After it prints the anagrams, the program displays the time it took to find the anagrams. Lastly, the program empties the set of anagrams so that it can store the anagrams of another word. This process repeats until the user enters an empty string to end the program.

**Experimental Results**

**Operation #1:**

For this operation, I will test the words ‘add’, ‘steer’, ‘earring’, and ‘intoxicate’. This allows me to determine the time it takes this function with no optimizations to return a set of anagrams after it built all N! permutations, where Nis the length of the word, for each word.

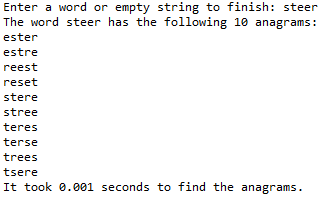
Case 1:

Input = ‘add’



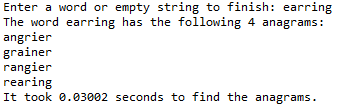
Case 2:

Input = ‘steer’



Case 3:

Input = ‘earring’



Case 4:

Input = ‘intoxicate’



**Operation #2:**

For this operation, I will test the same four words as the previous operation. Since all four words have at least one duplicate letter, these test cases allow me to determine the time it takes this function with the optimization for duplicate letters to return a set of anagrams after it built less than N! permutations for each word.

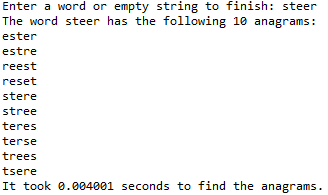
Case 1:

Input = ‘add’



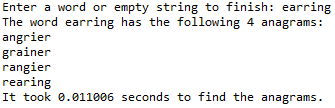
Case 2:

Input = ‘steer’



Case 3:

Input = ‘earring’



Case 4:

Input = ‘intoxicate’



**Operation #3:**

For this operation, I will test the same four words as the previous two operations. This allows me to determine the time it takes this function with both optimizations to return a set of anagrams after it built less permutations than the previous operation.

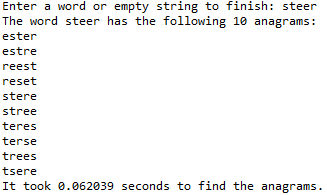
Case 1:

Input = ‘add’



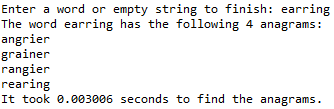
Case 2:

Input = ‘steer’



Case 3:

Input = ‘earring’



Case 4:

Input = ‘intoxicate’



**Graph of running time for each operation**

N = Length of word (3, 5, 7, 10)

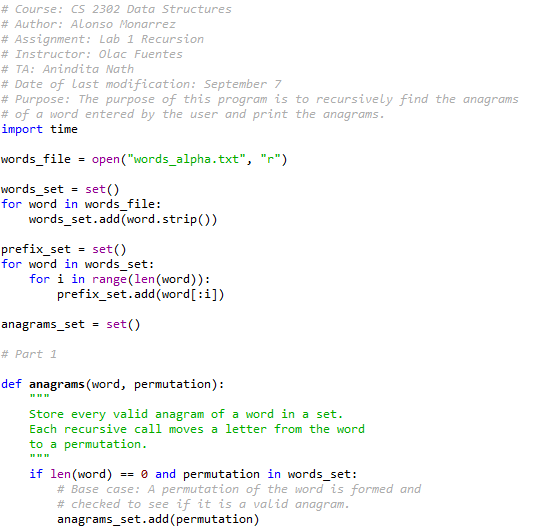
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 3 | 5 | 7 | 10 |
| Operation 1 | 0.001001 | 0.001000 | 0.030020 | 11.536648 |
| Operation 2 | 0.000000 | 0.004001 | 0.011006 | 3.564363 |
| Operation 3 | 0.000000 | 0.062039 | 0.003006 | 0.018014 |

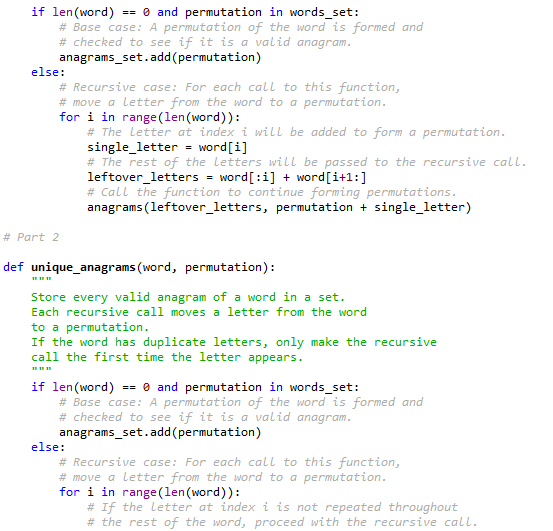
As the results show, there is a visible change in the running times among the three functions. While they all have similar running times for the first three input sizes, this all changes when the input size is at 10. This is because the first function has to make recursive calls for all N! permutations of a word, and there are no optimizations that can stop it from making recursive calls that will lead to a duplicate permutation or a partial permutation that is not a prefix of all the words in the set of valid English words. By contrast, the other two functions do not have to make as many recursive calls as the first function because their optimizations allow them to prevent any recursive call that would cause an unnecessary permutation to be built. Therefore, the more optimized the function becomes, the more efficient it is to find the anagrams of a word.

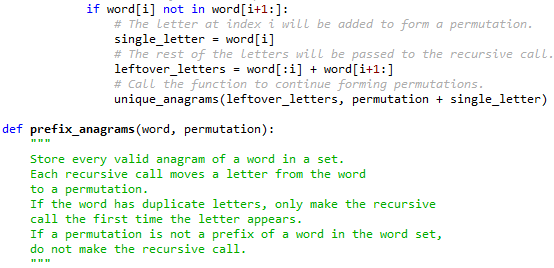
**Conclusion**

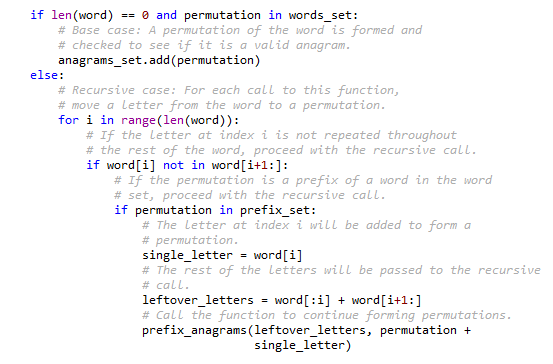
This lab helped me understand recursion better by allowing me to follow the path a recursive function takes to solve a particular problem. To accomplish this, I analyzed the first function in order to determine how it was able to create all of the permutations of a word. To help me visualize this process, I traced the function with words no longer than four letters, and I was able to interpret the order in which it made all of the recursive calls for each permutation. At this stage, I now understood the way in which recursion can solve a problem by breaking it apart into smaller sub-problems. Once I grasped the order of the recursive calls, I knew where and how I would be able to modify this function so that it did not have to make any recursive calls that would lead it to create unnecessary permutations. This resulted in the implementation of two optimized versions of the same recursive function, which had the same purpose of finding all of the anagrams of a word but required less time to do so since they did not need to make as many recursive calls as the original function. Overall, I found it helpful that this lab reinforced the main concepts behind recursion.

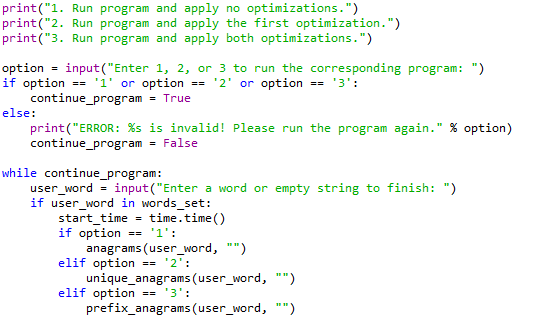
**Appendix**

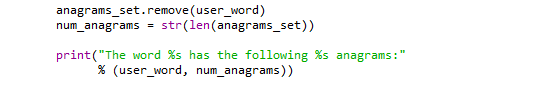


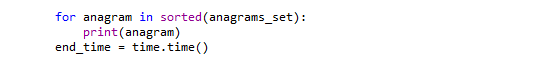


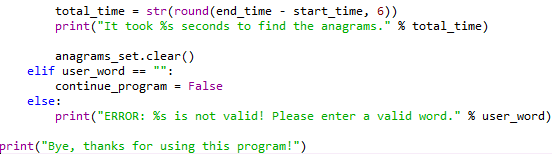












I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.