**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #1**

Due: September 6, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

Lab 1: Recursion focuses on developing a program that allows a user to input a word and then display the anagrams of the word. In order to identify these anagrams, the program must first read a downloaded file from GitHub, located at *https://raw.githubusercontent.com/dwyl/english-words/master/words\_alpha.txt,* which provides a thorough list of words from the English language. Since the main objective of this lab is to reinforce recursive solutions, the programs from Part 1 and Part 2 compute the anagrams recursively and display the results alphabetically. However, Part 2 focuses on developing a more efficient recursive function that eliminates unnecessary recursive calls by avoiding duplicate anagram creations and evaluating prefixes. It is also crucial to note that this lab introduces a new abstract data type, sets, which will be used to store the anagrams and other data.

**Proposed Solution Design and Implementation**

**Part 1:**

For part 1, I focused on developing a program that read the words\_alpha.txt file from GitHub, populated a set with these words, prompted the user to enter a word of their choice, computed the anagrams, and displayed the anagrams. I also calculated and displayed the performance time for the recursive method, FindAnagrams. Before solving the actual problem, I first divided my program into three different sections. The first section included a function to read the words\_alpha.txt file. In this function, I first open the file, passed as a parameter, and read each line sequentially. As the file was read, I populated a set with the valid English words. From the documentation, I learned that each line of a file ends with a newline character. Since I only wanted the set to contain the words, I removed the newline character from each word. Additionally, I realized that a user may accidentally add a blank line at the top or bottom of the .txt file. In order to prevent these blank lines from being added, I ensured that each line was not solely whitespace before adding it to the set. Lastly, I learned from the documentation that opening files in Python can result in a FileNotFoundError or IOError. I thus, placed the file opening and read statements in a try block and caught the errors if they occurred. I returned None if an error occurred to signal that the file’s read was unsuccessful. I also returned None if the file path passed in did not contain a .txt file. If an error did not occur, I returned the set of English words.

After the file was read, I continuously asked the user to enter a word, or empty string to finish. If the user entered a word, my program would transition to the second section which was the recursive function FindAnagrams. The input for this method includes the user’s word, an empty string which will be used for future permutations, a set of the English words, a set of anagrams that would be populated, and the original word. The function operates by appending each letter from the remaining letters to the permutation. A recursive call would then be made, passing in the new permutation as the parameter. For example, given the word “mac”, initially the letters to append would be “mac” and the permutation would be “”. Thus, three separate recursive calls would be made. The first recursive call would have “m” as the new permutation, the second recursive call would have “a” as the new permutation, and the third recursive call would have “c” as the new permutation. The remaining letters would continue to be appended to the prefix until there were no more letters to append. Once this occurred, the permutation would be added to the set of anagrams if it was a valid, English word. Since I passed in a copy of the reference to the anagram set, I did not return any values and just populated the set.

Section three of the program was the actual execution. I ensured that every time the user wanted to identify the anagrams of a word, I would calculate the performance time. I thus, subtracted the time before the recursive algorithm from the time after the algorithm’s completion.

**Part2:**

The overall functionality of the program in Part 2 is similar to that of Part 1. The words\_alpha.txt file is read, the words are added, and the anagrams, based on the user’s word, are printed and displayed. However, the program’s FindAnagram function is more efficient than that of Part 1 because it reduces the number of recursive calls made by using a prefix set and preventing duplicate words. Before implementing these changes, I first designed the overall layout of the program. Part 2’s program is divided into three sections. The first section is the opening and reading of the file. Unlike Part 1’s file read and open function, Part 2’s function includes two empty sets, one for the English words and one for their prefixes. Part 2 does not return a set since a copy of their references are passed as parameters. In order to catch the resulting errors that may occur, I implemented a try block around the FindAnagram’s method call and caught any FileNotFoundErrors or IOErrors that may occur. Just as Part 1’s program opens the file and reads each line sequentially, Part 2’s program does so as well. However, since n! permutations are being created in Part 1, where n is the length of the word, Part 2 reduces these recursive calls by identifying if their prefix exists. Thus, as I added the words to the set of English words, I also called a method, AddPrefixes which would add the prefixes of the current line’s word.

AddPrefixes accepts the current set of prefixes and the current word being read as parameters. I have the program first ensure that the word is not in the prefix set before its prefixes are added. For example, if the prefixes of the word “rate” were already added and the next line has “rat,” then all of the prefixes of “rat” will already have been added. If the word does not exist in the set, the program continuously removes the last letter and adds the remaining letters to the set. Since the prefix set was passed in as a parameter, I did not return a value.

After reading the file and populating the English word and prefix sets, I moved on to the third section, which is the function FindAnagrams. FindAnagram’s includes all the parameters from Part 1, however it also included the prefix set as a new parameter. This prefix set was used to prevent further recursive calls from being made if the current permutation was not in the prefix set. Additionally, in order to prevent duplicate anagrams from being created, I enforced new logic in my program. Before iterating through the letters to append and letters to append were being iterated through, I first checked if the letter was already in the set. If it was, a recursive call would not be made and the next letter would be evaluated. For example, given the word “ana,” the Part 1 program would make recursive calls with the prefix of “a,” “n,” and “a.” However, for Part 2, only two recursive calls would be made for “a” and “n”. This is because the program prohibits a second recursive call that appends “a” to be made.

Section three was the execution of the program and included the performance time calculation described above.

**Shared Design**

Both Part 1 and Part 2 ask the user to insert the file path for the words\_alpha.txt file. This path is supposed to include the directory and the .txt file name. Before opening the .txt file, the program first ensures that the path includes a .txt file at the end. This is to prevent the program from attempting to open another file type, like a .docx. If the user does not enter the correct file path, the program terminates.

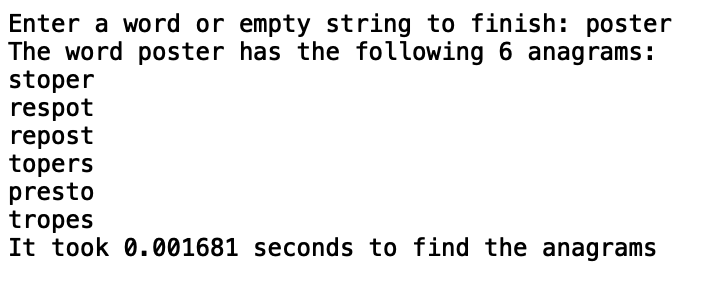
**Experimental Results**

**Part 1:**

For Part 1, I decided to insert “poster”, “university”, “permutation”, “ ”, “”, and “anna” as words. I tested “poster”, “university”, and “permutation” to ensure that the anagrams would be equal to those provided. Also, university provides an edge case for having 0 anagrams. I also added “” to ensure that the program would terminate for an empty string. “ ” is used as an edge case to ensure that an anagram search would not occur. Lastly, I tested “anna” because it contains duplicate letters and I wanted to compare its efficiency with that of Part 2.

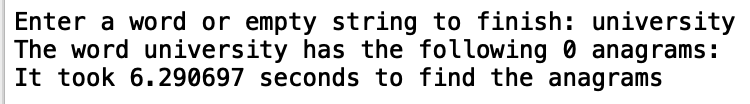
Case 1 (“poster”):

Input Word: “poster”



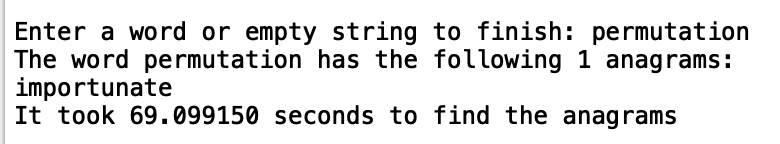
Case 2 (“university”):

Input Word: “university”



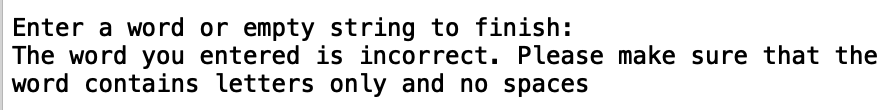
Case 3 (“permutation”):

Input Word: “permutation”



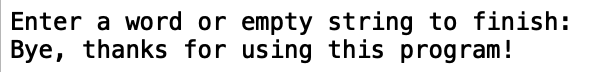
Case 4 (“ ”):

Input Word: “ ”



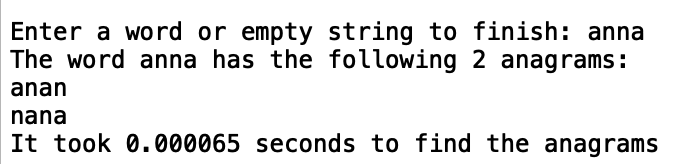
Case 5 (“”):

Input Word: “”



Case 6 (“anna”):

Input Word: “anna”



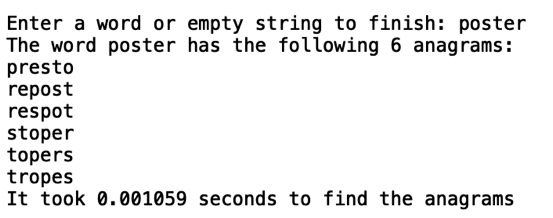
As the cases demonstrate, the program was able to function for all the words and match the same anagrams as those provided in the lab’s examples. From these cases, I was also able to identify the inefficiency of the recursive function. Case 3 demonstrates that it took the recursive function approximately 69 seconds to execute. This number is extremely startling and demonstrates the original problem that I had discovered while creating this function. While I was writing the recursive calls for “anna”, I realized that duplicate words would be computed due to the duplicate letters. I thus, thought of adding a set that would keep track of the used characters. I did not make this change to Part 1, however, to demonstrate my initial idea and how it would be enhanced in Part 2. Additionally, I was able to ensure that the program would not perform a search for anagrams of white spaces, as seen in Case 4.

**Part 2:**

For Part 1, I decided to insert “poster”, “university”, “permutation”, “ab6”, “”, and “anna”. I tested “poster”, “university”, and “permutation” to ensure that the anagrams would be equal to those provided. I also added “” to ensure that the program would terminate for an empty string. “ab6” is used as an edge case to ensure that an anagram search would not occur since a number is included. Lastly, I tested “anna” because it contains duplicate letters and I wanted to compare its efficiency with Part 1’s recursive function.

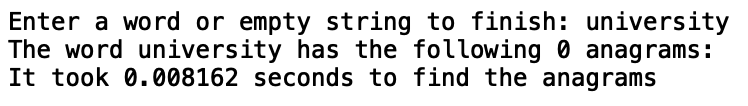
Case 1 (“poster”):

Input Word: “poster”



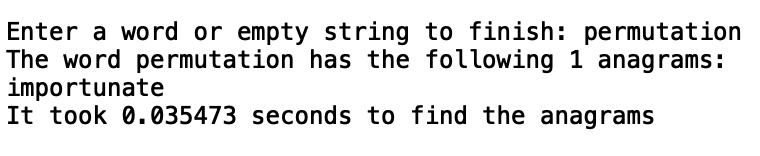
Case 2 (“university”):

Input Word: “university”



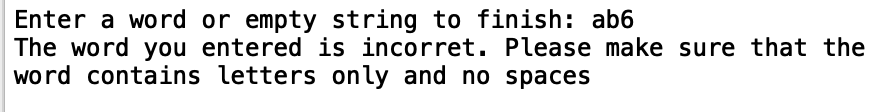
Case 3 (“permutation”):

Input Word: “permutation”



Case 4 (“ab6”):

Input Word: “ab6”



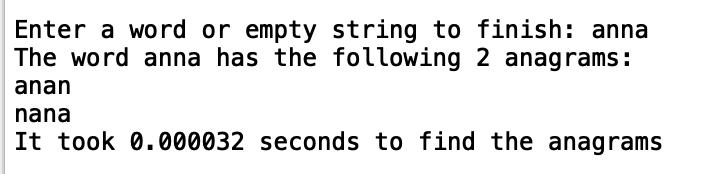
Case 5 (“”):

Input Word: “”



Case 6 (“anna”):

Input Word: “anna”



The cases above demonstrate that Part 2’s program was able to identify the same anagrams as Part 1, but in a shorter period of time. From Case 1, Case 2, and Case 3, I was able to ensure that the outputs matched the outputs from the lab’s example. Additionally, I was able to determine that the recursive function in Part 2 was more efficient than Part 1’s function. Whereas Part 1’s function took 0.000065 seconds to find the anagrams of “anna”, Part 2’s function only took 0.000032 seconds to find the same anagrams. Thus, Part 1’s function’s performance time was two times the performance time of Part 2’s function. Additionally, by checking the prefixes, I was able to decrease the performance time for “university”. In Part 2, the performance time for “university” was 0.008 seconds, whereas the performance time for Part 1 was 6 seconds.

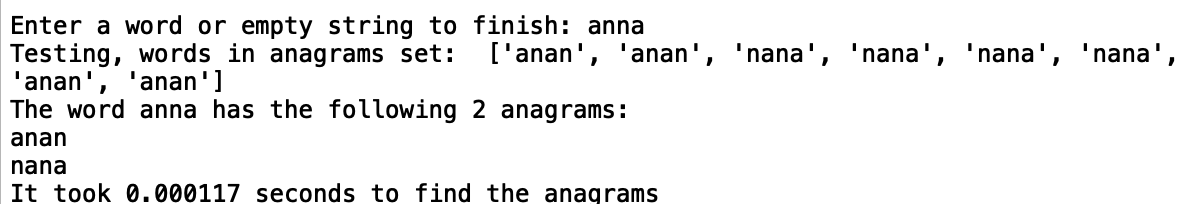
**Duplicate Anagrams:**

For Part 1 and Part 2, I wanted to demonstrate how the anagram set looked before it was printed. Part 1 removed all duplicates before displaying the anagram set. Part 2, however, prevented the duplicates from being added in the recursive function. Case 1 has “anna” for the input word and demonstrates the anagrams for Part 1 and Part 2 before they are sorted.

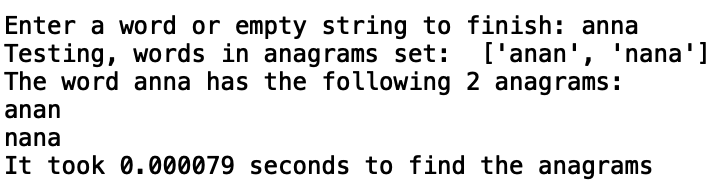
Case 1 (“anna”):

Input: “anna”

Part 1



Part 2:



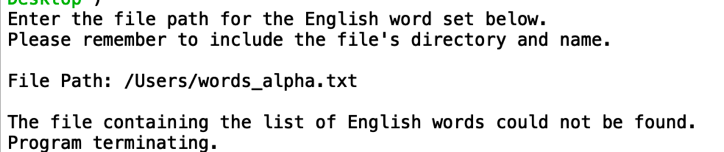
Case 1 clearly demonstrates that Part 2 does not allow recursive calls on duplicate letter combinations. Part 1, however, will continue to make n! combinations, where n is the length of the string. Thus, Part 1 is not only inefficient, but its anagram set still needs to be edited before it is displayed, unlike Part 2.

**Shared Design:**

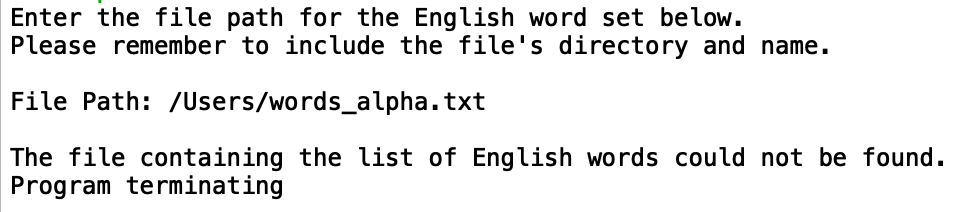
For both Part 1 and Part 2, I ensured that the programs would function correctly if the file was not found or if the user inputted the wrong file path.

Case 1 (“File Not Found”):

Part 1:

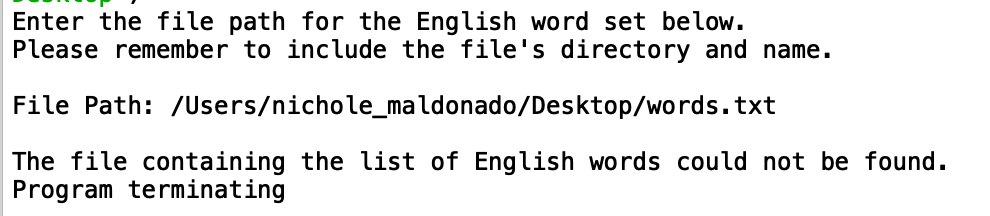


Part 2:

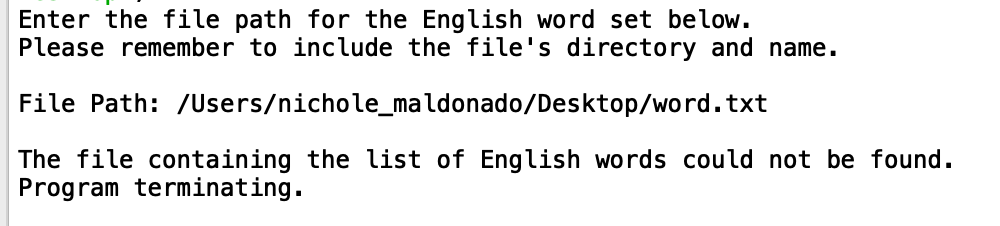


Case 2 (“Wrong Path”):

Part 1



Part 2



Case 1 and 2 demonstrate that the programs are able to operate efficiently if the file is not found or the user inputs the wrong path.

**Performance Time:**

In order to compare the performance of the programs, I entered a range of words that include “apple”, “green”, “presto”, “xylophone”, and “z”. Each word was entered five times sequentially. Once the word was inserted 5 times, I stopped the program and restarted the same procedures for the next word. By including words from “a” to “z”, I was able to test the best and worst cases of the programs.









The performance graphs above demonstrate that for almost all the instances, Part 2’s algorithm had a shorter performance time than Part 1’s algorithm. By reducing the number of recursive calls through prefix checks and duplicate letter avoidance, Part 2’s performance showed its superiority. In fact, this can clearly be seen for “xylophone”. When viewing the bar chart, the performance time for Part 2 is barely visible. However, for a single letter, Part 1‘s performance time was lower for Trial 3 and Trial 5. It is important to note that Part 1 was only 0.000001 seconds faster than Part 2. Thus, when compared to all the other data gaps, it is still safe to conclude that Part 2 is more efficient. Even though Part 1 may be more efficient for a single letter, as the user inputs larger words, Part 2’s program is superior.

In order to further compare the performance time between Part 1 and Part 2, I tested how each algorithm would respond to words with 1 to 6 letters. For the one letter word, I used “z”, for the two letter word I used “it”, for the three letter word I used “lab”, for the four letter word I used “anna”, for the five letter word I used “apple”, and for the six letter word I used “presto”. For each word, I inserted the it three times for each program. I also stopped and restarted the program for each word section.







As the graph demonstrates, for words with three or less letters, Part 1 is more efficient. However, as soon as the word has four or more letters, Part 2 is dramatically superior. The reason why Part 1 is more efficient than Part 2 is probably because Part 1 is doing less “lookups”. Since Part 1 is not constantly checking the set of prefixes, it is able to calculate the anagrams faster. However, once the words become larger and more complex, the duplicate calls and lack of prefix checks are a hindrance. As for Part 2, although it is slower for less than three letter words, the graph shows that these differences are insignificant. For only three out of an infinite number of lettered words is Part 1 superior. Thus, is can be concluded that Part 2 is the optimal choice.

**Recursive Calls:**

In order to verify that Part 2 is reducing the number of recursive calls, the following test compares the number of recursive calls that occurred for each word. The number of recursive calls were calculated by a global counter.

****

****

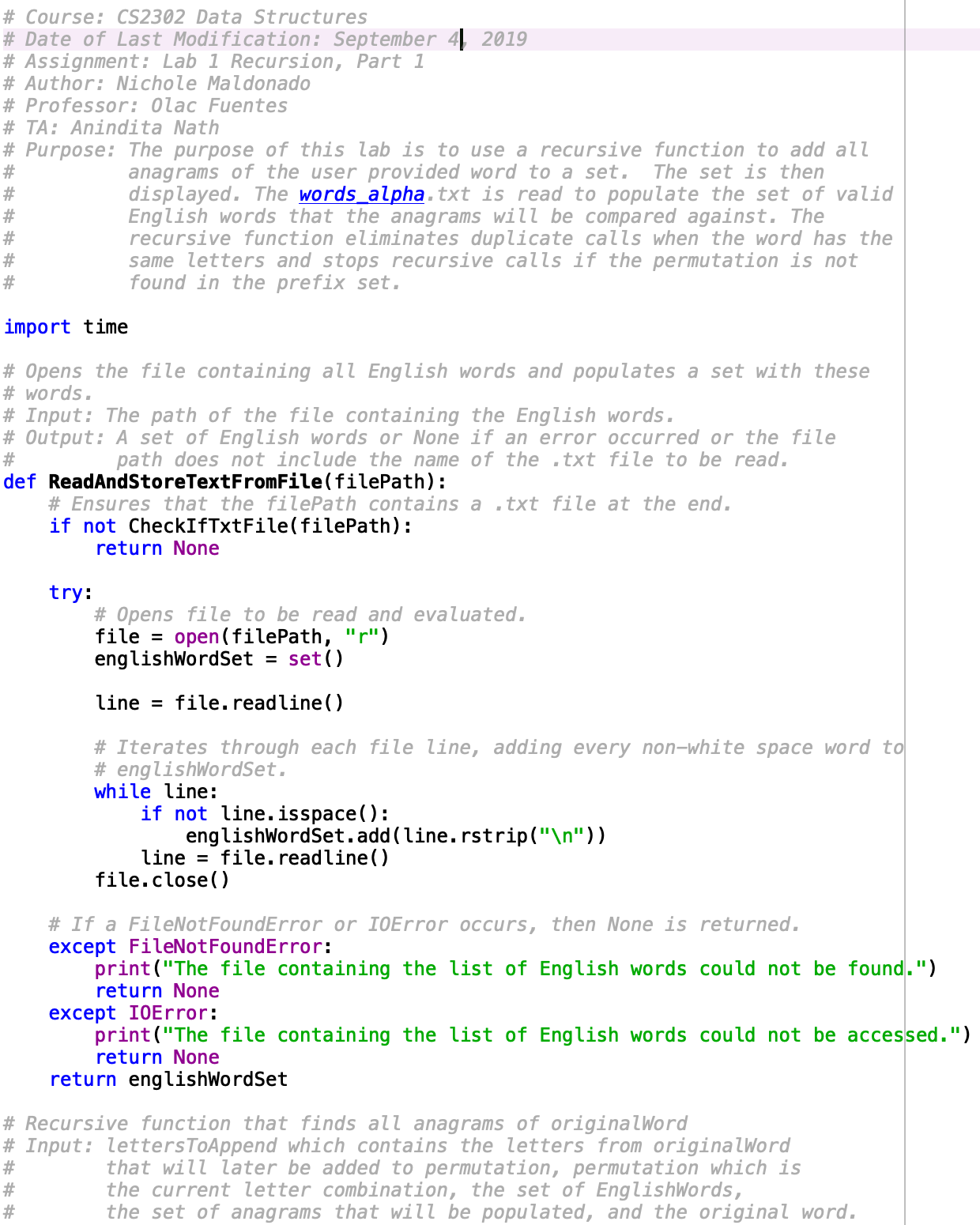
For all the recursive calls, Part 1 had a larger amount than Part 2. As can be seen with the number of recursive calls for “coconut”, the number of recursive calls for Part 2 is almost negligible when compared to Part 1.

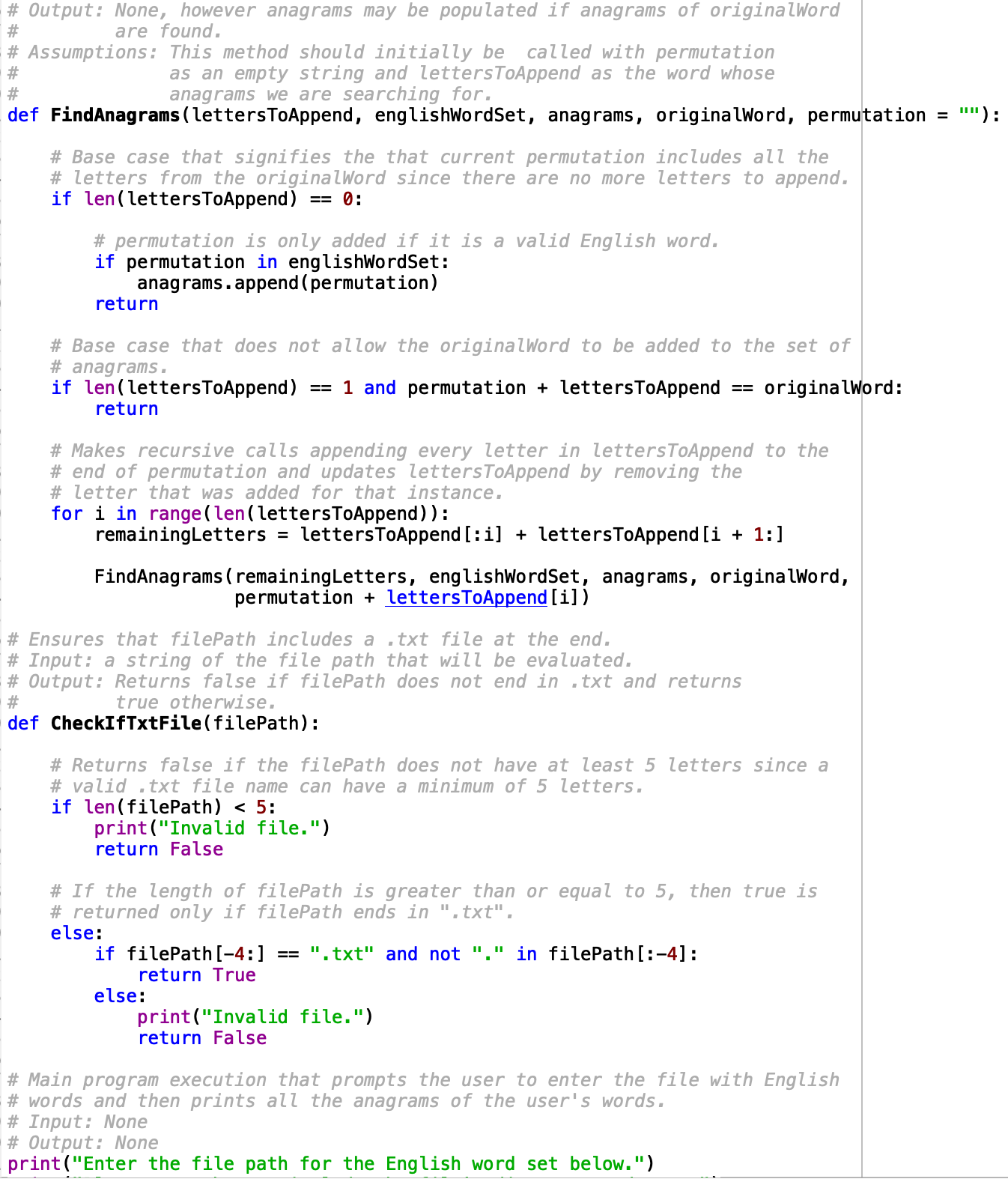
**Conclusion**

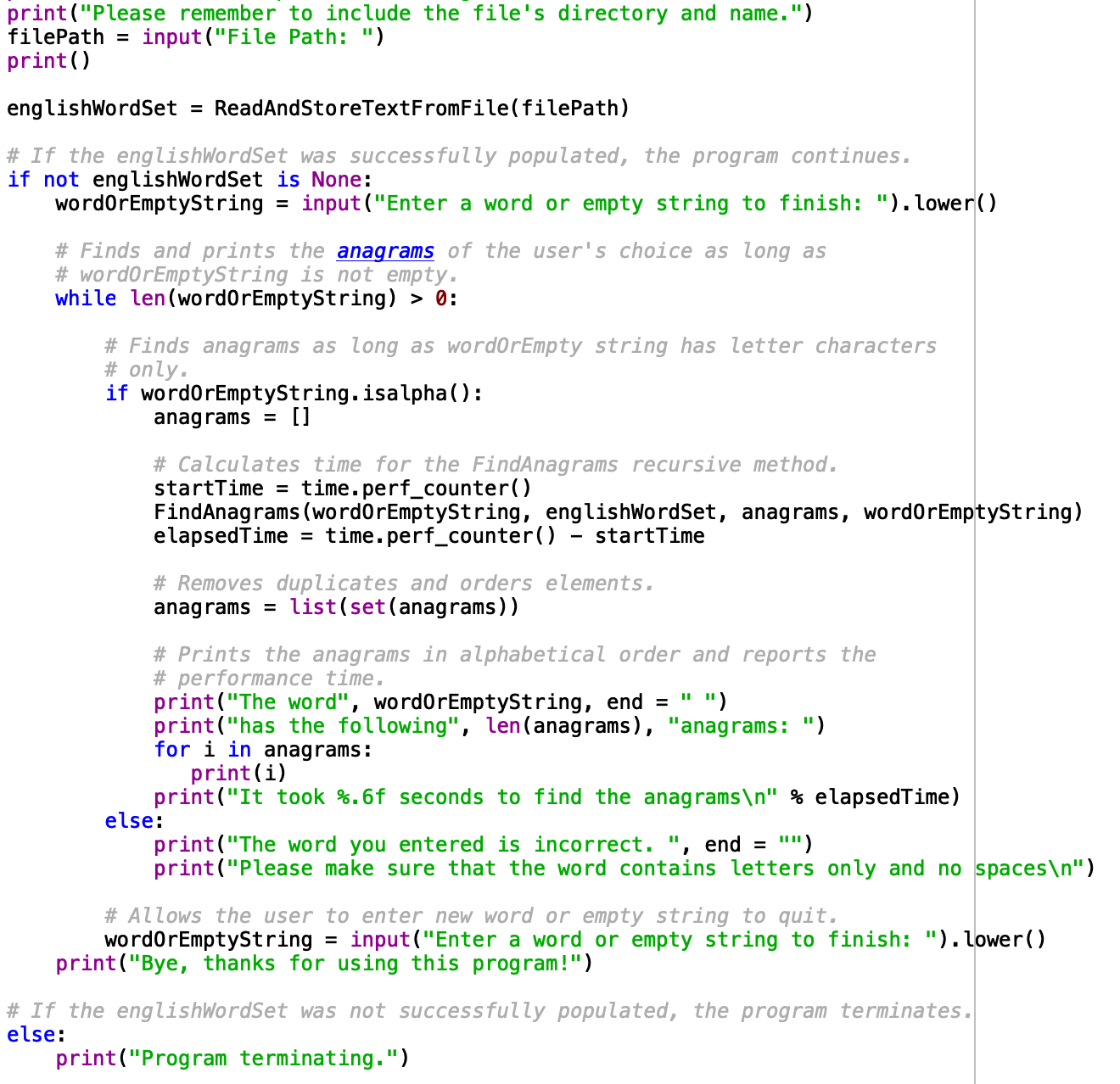
In all, I learned from the lab that recursive calls can be very efficient with the proper logic. By simply eliminating the recursive calls for duplicate letters, I was able to reduce the performance time. For example, the word “permutation” took 69 seconds for Part 1 and 0.035 seconds for Part 2. Additionally, this lab taught me about a new abstract data type - sets. I had never heard of sets before and was hesitant about using them. I thought that looking for an element in a set would require at most, iterating through all the elements. However, after looking up information on sets, I learned that on average, the element checks were done in constant time. I also learned that since sets do not have duplicates, if you convert a list to a set, then all the duplicates will instantly be removed. Thus, sets are an abstract data type that I definitely plan on using moving forward. Another interesting topic that I learned about was file locations. At first, I hardcoded the directory of the words\_alpha.txt file. However, I realized that other users who download the code may not have the code stored in the same directory. I was able to implement a feature to allow the users to access the .txt file from anywhere in their computer as long as they know its location. Additionally, the results from the graph conclude the importance of careful planning in recursive functions. By not only using a set of words, but also prefixes to compare the appending words, I was able to successfully limit the number of recursive calls. One of the most interesting things that I found in this lab was the large gaps in recursive calls. For the word “coconut”, Part 1’s program performed 13,193 more recursive calls than Part 2’s program. In all, this lab was very interesting and demonstrates that there is room for improvement in all programs.

**Appendix**

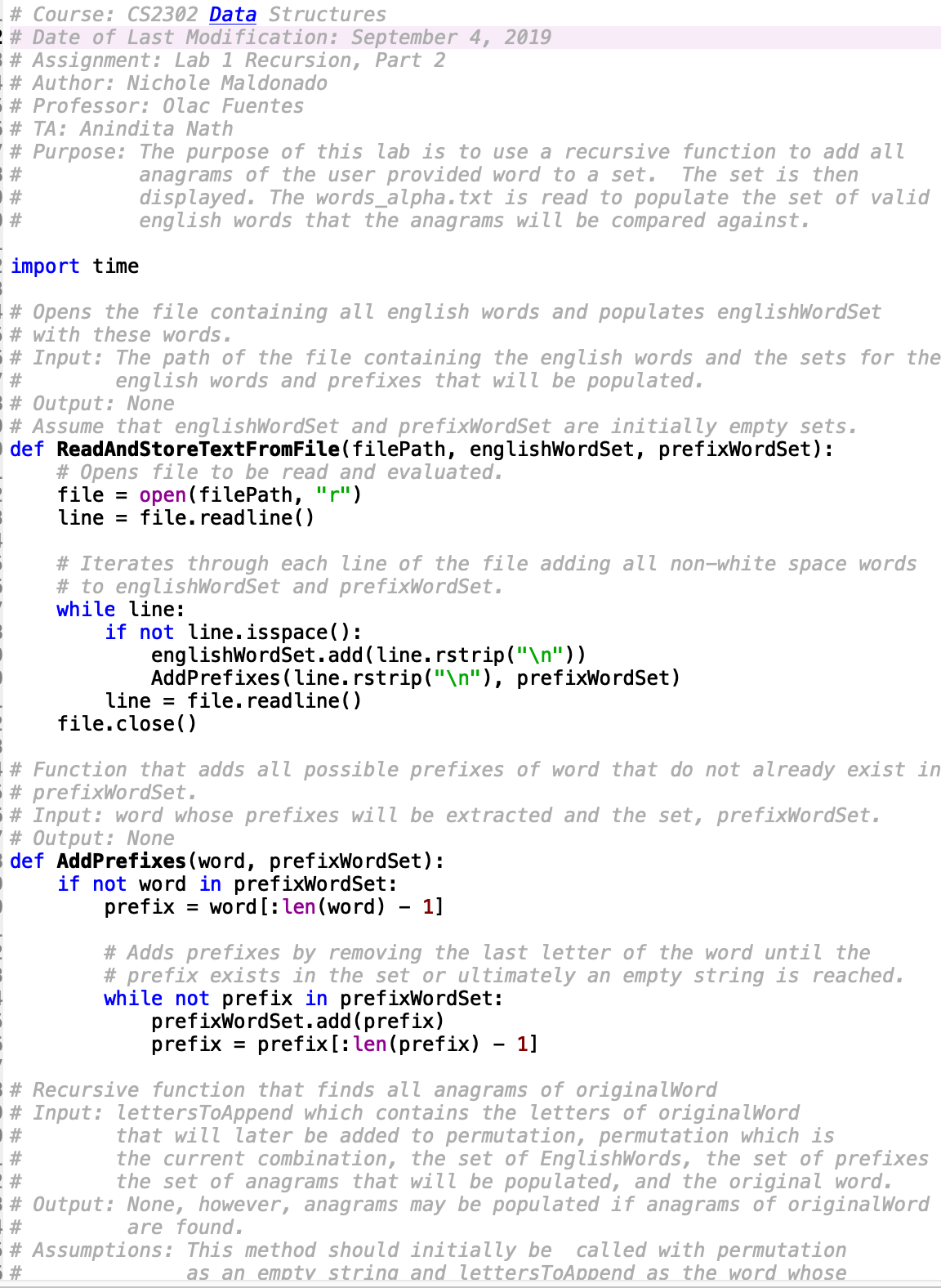
**Part 1:**

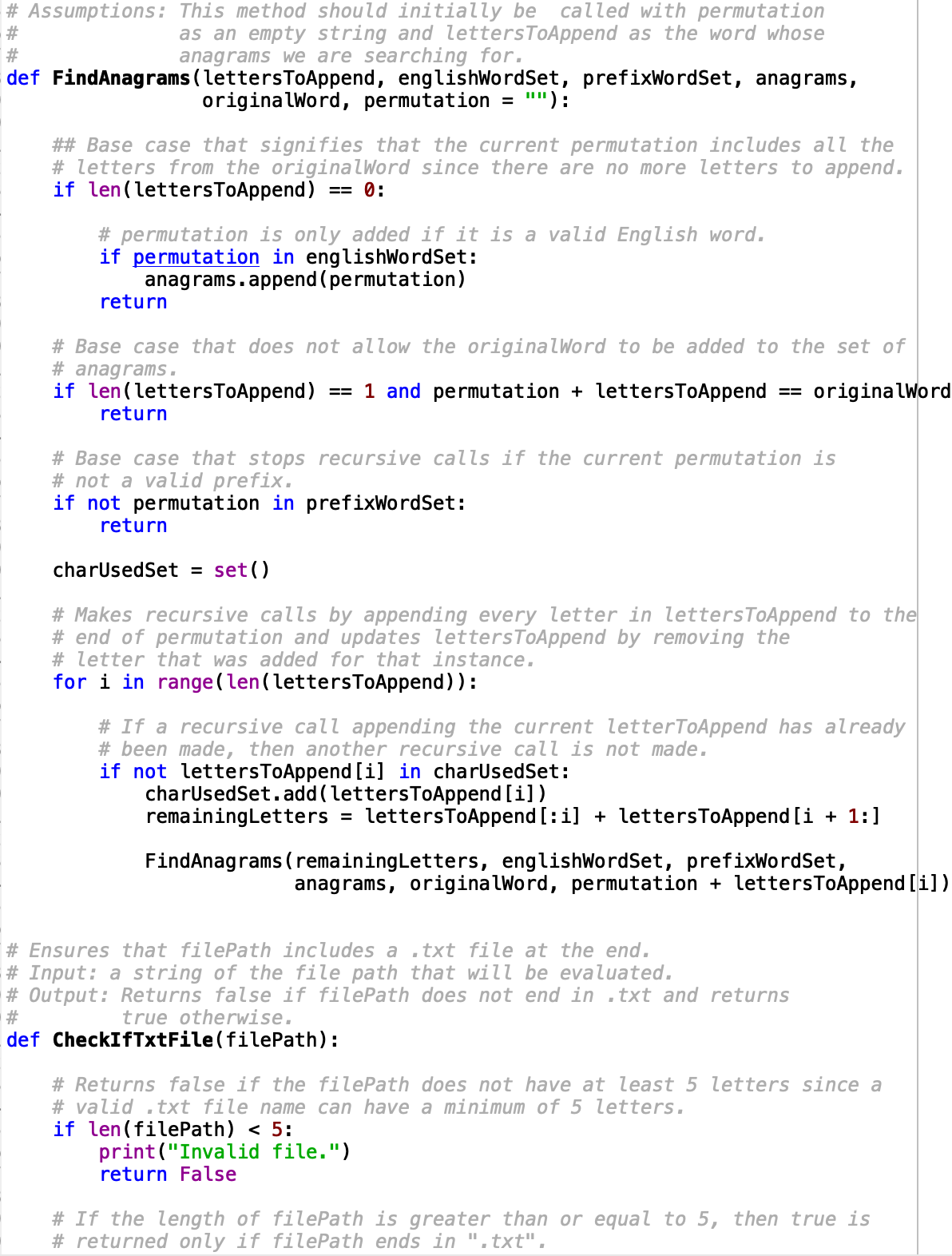
****

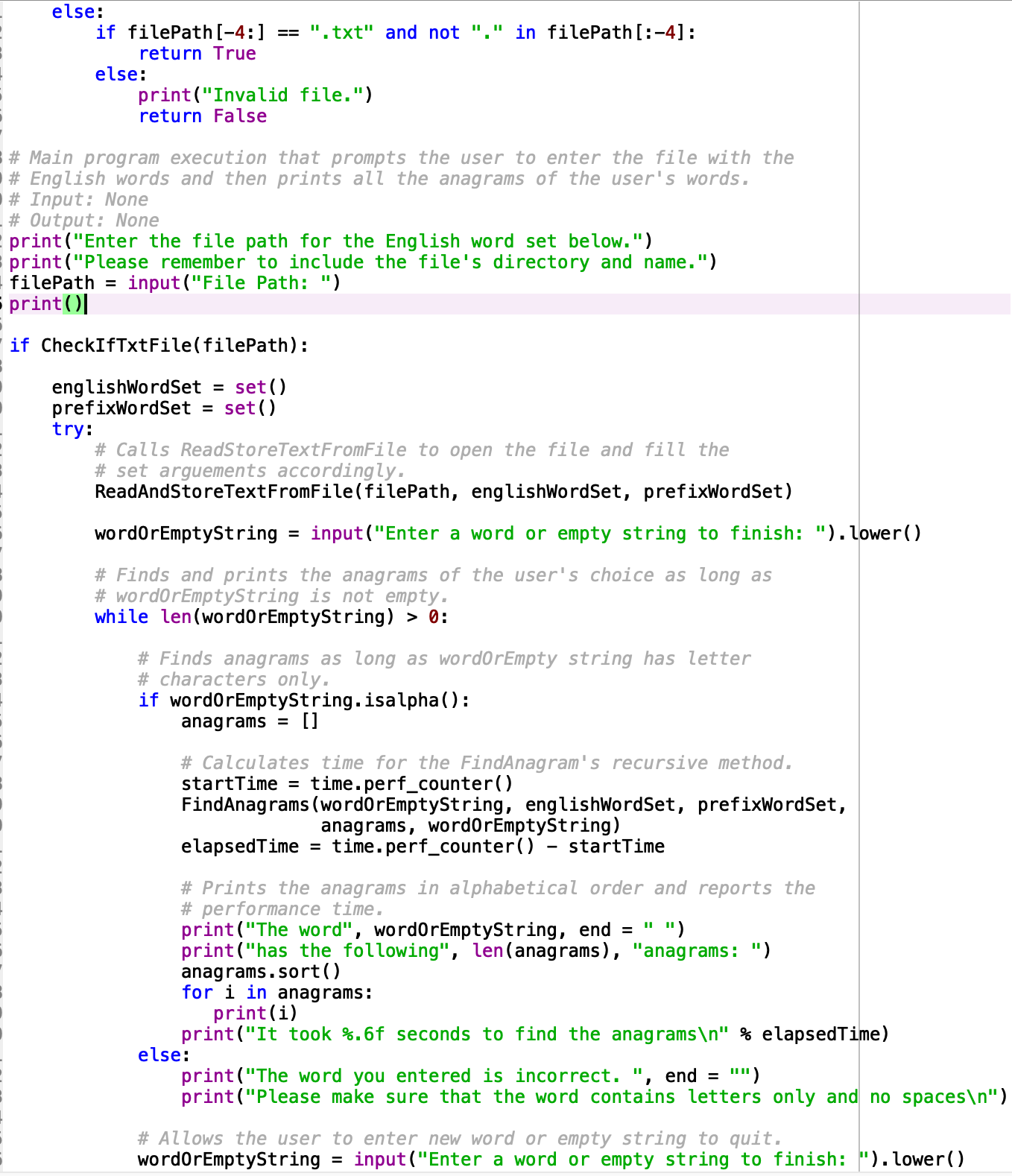
****

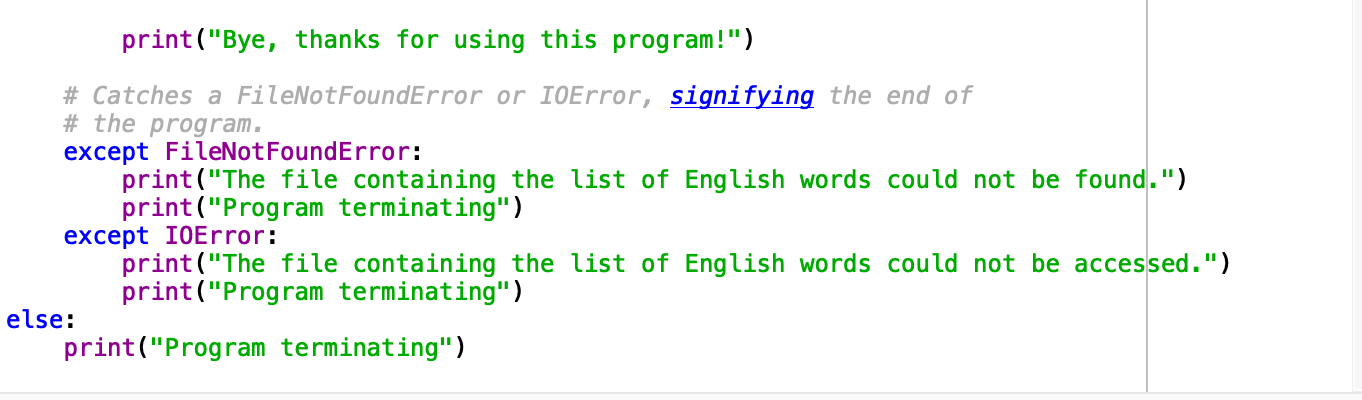
****

**Part 2:**

****

****

****

****

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



-----------------------------------