**LAB 1**

**Daniel Cruz**

**Introduction**

The objective of this lab was to code a program that will read a text file of English words and use this to find anagrams of a user inputted word. This lab’s goal was to get students familiar with recursive functions and their benefits. It is also designed to exemplify how to optimize recursive code within the given problem.

**Proposed Solution Design and Implementation**

Part 1.

My code is essentially broken up into three methods. The first method is used to read all the words from the text file. The method inputs a string that will be the name of the file and outputs a set with all the words in it. Since the text file separates each word with a newline character, I elected to read the text file line by line. In each line I strip the newline character and add the word into a set that will be returned by the method.

The next method is a wrapper method for the true recursive method. I chose to add this layer of abstraction to make user interfacing simpler. The user inputs a string to the method and passes a set that contains words from the text file. The method records the start time and initializes an empty set that will be passed to the recursive function to hold all the anagrams. If no anagrams were found it prints that it could not find any anagrams, otherwise it prints all the anagrams. Finally, the method prints the time elapsed searching for anagrams.

The final method is the recursive method. This method is a modified version of the algorithm proposed from Zybooks. The algorithm takes a string that is the word to permutate, an empty string for recursive permutating, a set of strings of English words, and finally a set that will store all found anagrams to be used outside of the method. The recursive section of the method works by picking a letter, placing it in the permutating string, and then removing it from the original string. It does this for every letter in the original string. The method is then recursively called using the original string stripped of the single letter and the permutating string with the stripped letter added to it. The base case is when there are no letters left in the original string to place into the permutated string. At this time, the permutated string is checked to see if the word exists in the set of English words. If the word is a real English word, then it is added to the set of anagrams. If it is not found in the set of English words, it returns without keeping the permutated string.

Part 2.

Part 2 code is very similar to the first section, so I will only discuss the new additions to the code to adhere to the new restrictions. In part 2 the code is optimized to quit finding a permutation if it is not a prefix to a known word. It also prevents permutation of the letters that repeat in a word, cutting down the amount of unnecessary recursion calls.

For reading from the file, the only major change was to also add the prefixes of each word in the file starting from the first letter to the second to last letter. Both the complete word set and set of prefixes of each word is returned.

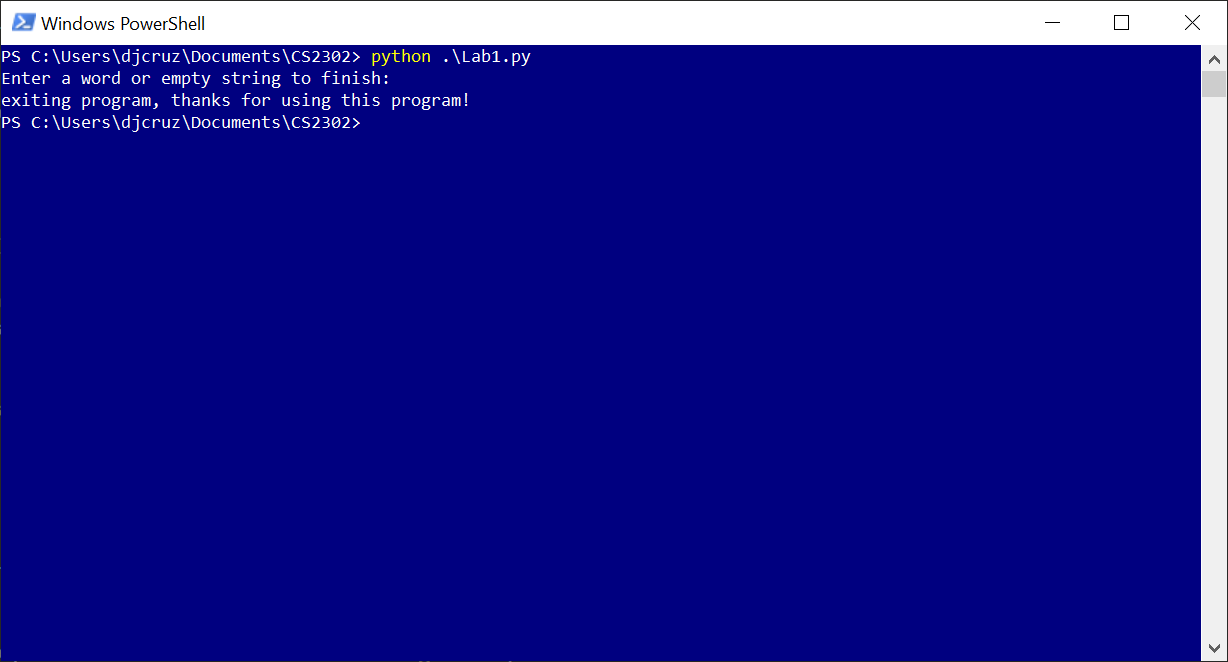
The wrapper for the recursion method was only altered to allow passing of the new prefix set to it.

In the recursion method, the recursive case of the method has been altered. Every permutated string being built is checked with the prefix set to see if there are possible words to make. If not, the recursive case ends without continuing building the string. The method now also contains a set of used letters. Every time a letter is chosen to be added to the permutated string, it makes sure it is not in the used letter set. If it is, then recursion ends without building the rest of the string. Otherwise, recursion continues and the letter is added to the used letter set so that it cannot be used again in case a letter is repeated in the word. This this reset in every method call so as to still be able to find all possible anagrams.

**Experimental Results**

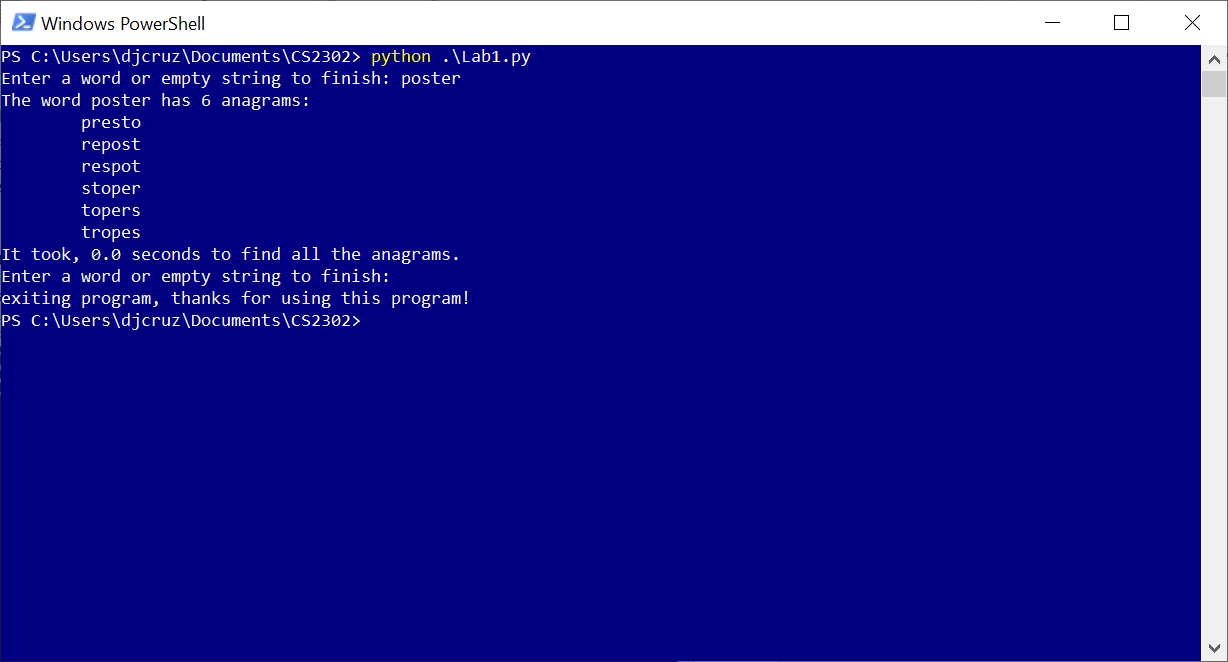
* Part 1:

The first test is an edge case, if the program is started, then exited immediately:



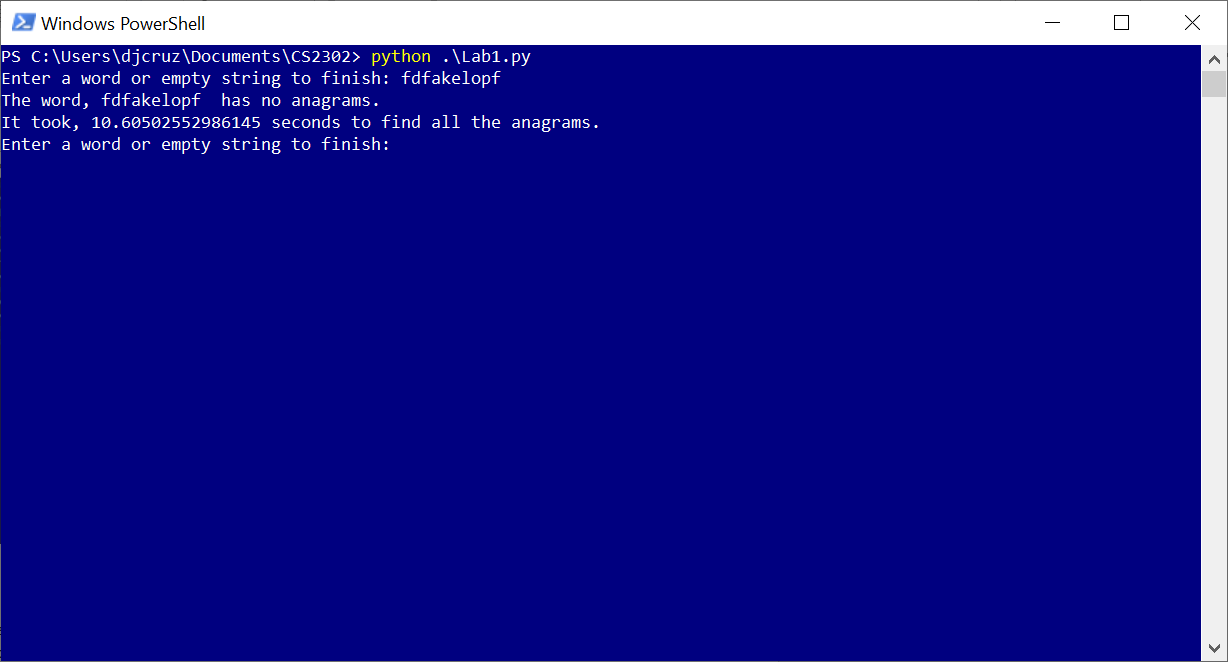
As shown in the results, the program gracefully exits.

Next test is a common use case. In this test I input the word ‘poster’ and I should expect to see results similar to those shown in the lab instructions.



This is shown to be true. All six anagrams have been found and are printed in alphabetical order. The time ranges from 0.0 to .01 seconds.

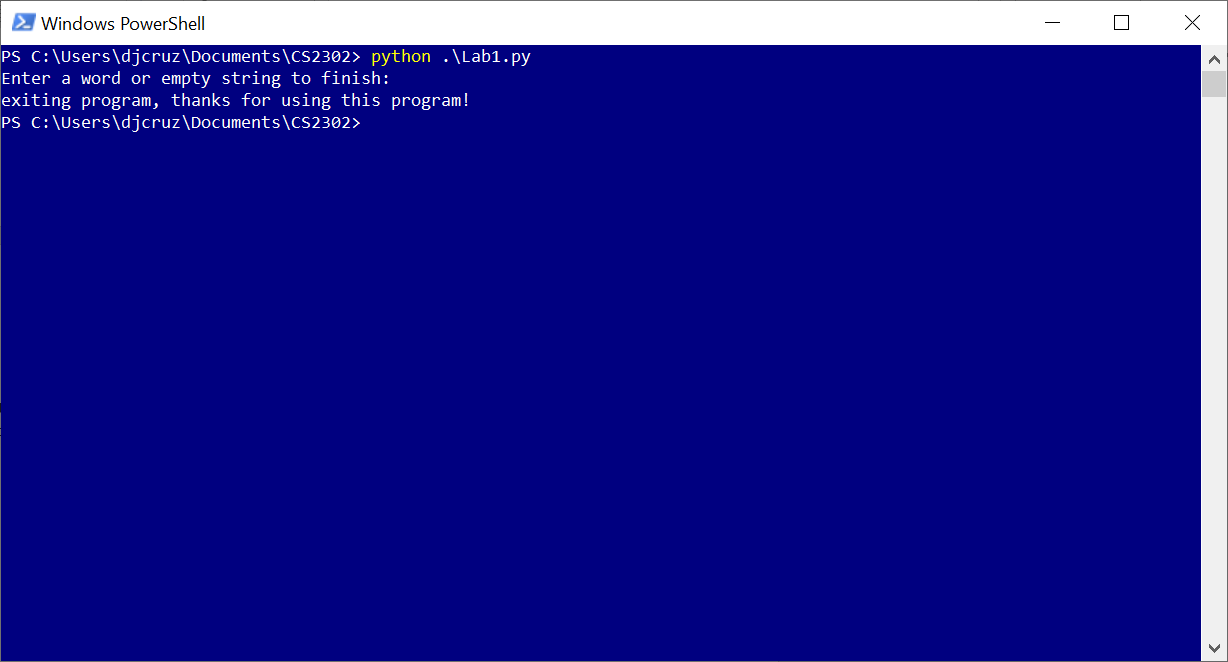
Next test is a large string of nonsense, ‘fdfakelopf. since part 1 contains no shortcuts to stop recursion, this should take significantly long than the smaller six letter word:



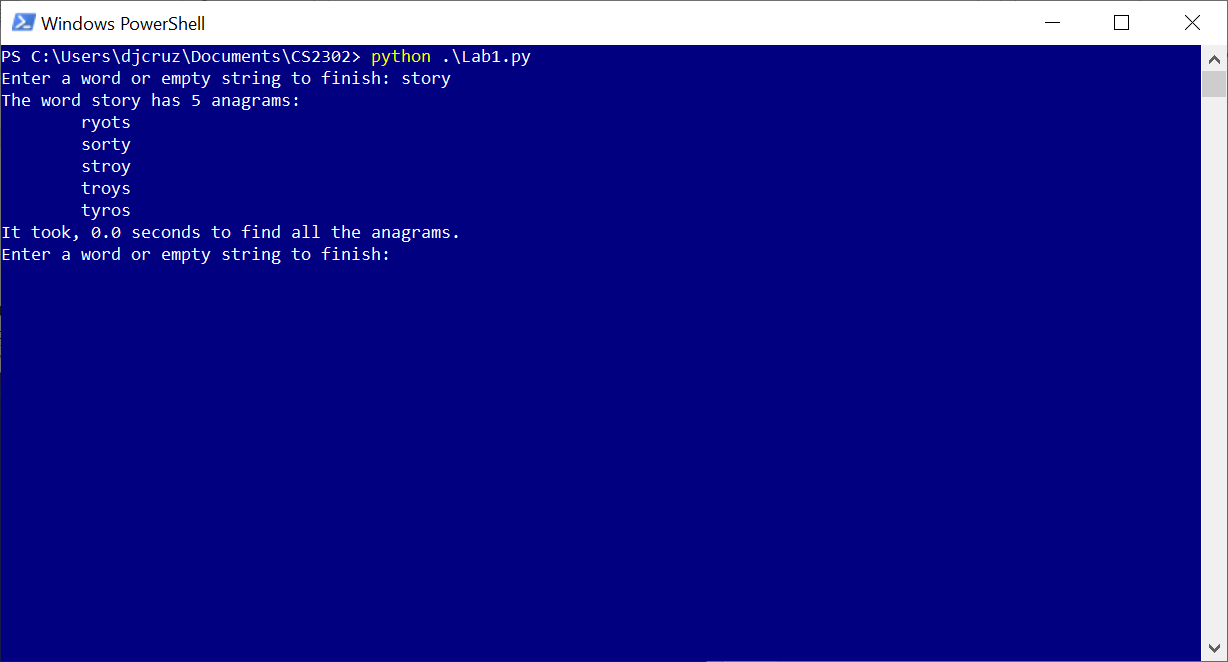
This took a considerable amount of more time and returned with no anagrams.

* Part 2:

Just like part 1 the first test will be a graceful start and finish:

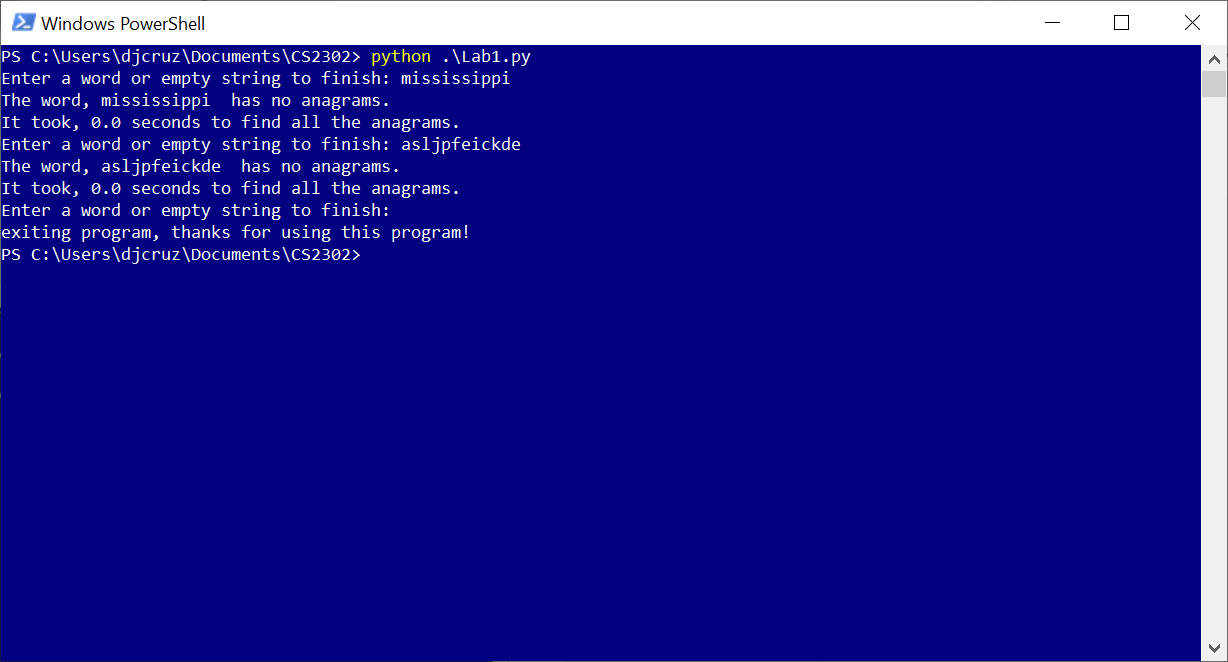


This next test will be a typical use case:



As shown, all anagrams have been found and printed.

Now that the code has been optimized for larger words, a long unique (meaning there is no anagrams) word and a large string of nonsense:

As shown, the code runs significantly faster in both cases. The more unlikely the word will have the anagrams, the less time it will take to return as such.

Finally, the percentage change between the two parts will be analyzed. A string of the continuous growing alphabet is inputted into both methods. For example, ‘a’, then ‘ab’, then ‘abc’, all the way up to ‘abcdefghijk’. This best simulates the worst runtime as there is no anagrams for the alphabet.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Method 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0.020051 | 0.110301 | 0.974926 | 9.84997 | 110.1499 |
| Method 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.014561 | 0.010056 |

If either method returned a 0, the percentage change resulted to 0. Method2 didn’t show a 0 until the second to last call. However, from here, it can be seen that there is a massive percentage change between the two values. The runtime for the final string was 110.1499 seconds for method 1 and 0.010056 seconds for method 2

**Conclusion**

This lab helped me refamiliarize myself with recursive functions. The challenges I faced with this lab stemmed from figuring out a way to collect all the anagrams without losing them in the return statements. I figured out the best way to preserve them was to have a set that went down with the recursions to collect them and preserve them from the method return. The biggest takeaway I got from this lab is how fast the runtime of a recursive function can grow based upon its input size. However, as exemplified in part two, with clever ways to cut down on recursion this runtime can be greatly reduced.

**Appendix**

from time import time

def getWords1(fileName):

try:

fp = open(fileName)

wordSet = set()

for word in fp:

#remove newline character from word

word = word.replace('\n', '')

wordSet.add(word)

return wordSet

except IOError as e:

print(e)

return set() #return empty set on IO error

def getWords2(fileName):

try:

fp = open(fileName)

wordSet = set()

strSet = set()

for word in fp:

word = word.replace('\n', '')

#for each word, add a running prefix to a different set.

for i in range(len(word)):

strSet.add(word[:i])

wordSet.add(word)

return wordSet, strSet

except IOError as e:

print(e)

return set()

#These recusive methods are based off and adapted from zybooks section 2.6

def scramble1(r\_letters, wordSet, returnSet, s\_letters = ""):

#base case

if len(r\_letters) == 0:

if(s\_letters in wordSet):

returnSet.add(s\_letters)

#recursive case

else:

#for each letter in the string, add it to a different string, remove it from the previous string,

#then recusively repeat the process

for i in range(len(r\_letters)):

scramble\_letter = r\_letters[i]

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

scramble1(remaining\_letters, wordSet, returnSet, s\_letters+scramble\_letter)

def scramble2(r\_letters, wordSet, prefixSet, returnSet, s\_letters = ""):

#base case

if len(r\_letters) == 0:

if(s\_letters in wordSet):

returnSet.add(s\_letters)

#recusive case

else:

#for each letter in the string, add it to a different string, remove it from the previous string,

#then recusively repeat the process

usedLetters = set()

for i in range(len(r\_letters)):

scramble\_letter = r\_letters[i]

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

#If this is not the end of the string and the prefix is found in the set of word prefixes, then recusively continue.

#otherwise finish this iteration of the for loop here without continueing recursion

if((len(remaining\_letters) != 0) and not ((s\_letters+scramble\_letter) in prefixSet)):

continue

#if the letter chosen to be appended has already been used, stop this iteration for the for loop without recursion

#else add letter to the set and continue forth.

if(scramble\_letter in usedLetters):

continue

usedLetters.add(scramble\_letter)

scramble2(remaining\_letters, wordSet, prefixSet, returnSet, s\_letters+scramble\_letter)

def printAnagram1(word, wordSet):

#will be used to calculate runtime.

startTime = time()

#an empty set to store anagrams found from the recursive function

anagramSet = set()

scramble1(word, wordSet, anagramSet)

endTime = time()

if(len(anagramSet) == 0):

print("The word, " + word + " has no anagrams.")

else:

#convert to list for sorting

anagramList = list(anagramSet)

#if the inputted word is found in the anagram list, remove it.

if word in anagramList:

anagramList.remove(word)

#sorts anagram list alphabetically

anagramList.sort()

print("The word " + word + " has " + str(len(anagramList)) + " anagrams:")

for anagram in anagramList:

print("\t" + anagram)

print("It took, " + str(endTime - startTime) + " seconds to find all the anagrams.")

def printAnagram2(word, wordSet, prefixSet):

#will be used to calculate runtime.

startTime = time()

wordList = list(word)

#an empty set to store anagrams found from the recursive function

anagramSet = set()

scramble2(word, wordSet, prefixSet, anagramSet)

endTime = time()

if(len(anagramSet) == 0):

print("The word, " + word + " has no anagrams.")

else:

#convert to list for sorting

anagramList = list(anagramSet)

#if the inputted word is found in the anagram list, remove it.

if word in anagramList:

anagramList.remove(word)

#sorts anagram list alphabetically

anagramList.sort()

print("The word " + word + " has " + str(len(anagramList)) + " anagrams:")

for anagram in anagramList:

print("\t" + anagram)

print("It took, " + str(endTime - startTime) + " seconds to find all the anagrams.")

def main1():

wordSet = getWords1("words\_alpha.txt")

userInput = None

while userInput != "":

print("Enter a word or empty string to finish: ", end = "")

userInput = input()

if(userInput != ""):

printAnagram1(userInput, wordSet)

print("exiting program, thanks for using this program!")

def main2():

wordSet, prefixSet = getWords2("words\_alpha.txt")

userInput = None

while userInput != "":

print("Enter a word or empty string to finish: ", end = "")

userInput = input()

if(userInput != ""):

printAnagram2(userInput, wordSet, prefixSet)

print("exiting program, thanks for using this program!")

if \_\_name\_\_ == '\_\_main\_\_':

#main1()

main2()

I [Daniel Cruz] 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