Lab 1 Report – Recursion

Jonatan M. Contreras

University of Texas at El Paso

Lab 1 Report

**Introduction**

We were asked to write a program that asks the user to input a word, and print all the anagrams the user-given word can create. The program was to find the anagrams recursively. We were then asked to implement two optimizations for the recursive function. The first optimization would ensure only recursive calls were made after the first time a character appeared in the word. The second optimization checks to see if a partial permutation, as it’s being created by the algorithm, is in a set of all prefixes of all the words from the words\_alpha.txt file.

**Proposed Solution Design and Implementation**

*Part 1*

For part 1, I began with the storing of the words in the words\_alpha.txt file. I read the file in and stored it in a set. I then made two modifications to the permutation algorithm in the ZyBooks textbook, Ch. 2.6: “Recursive exploration of all possibilities,” and used it to create the user-given word permutations. First, I changed the permutation algorithm to return a list of permutations instead of printing the permutations. Second, I included in the algorithm a check to revise that the permutation the algorithm was currently creating was not the same as the user-given word so that we don’t store it as a valid permutation.

I created a separate function that checks whether the permutations are in the set of words created from the words\_alpha.txt file. It takes a list of permutations as input and returns a list of valid anagrams from that permutation list.

The proposed solution design then asks the user to input a word. It then takes that word and creates a list of all its permutations. Finally, it checks to see which of those permutations are in the set of words given by the words\_alpha.txt file and returns a list that contains the valid anagrams.

*Part 2.1: First Optimization*

This optimization called for the algorithm to only make recursive calls the first time a character appeared in a letter for the first optimization. To do this, the algorithm first checks to see if the letter it is about to use for creating the next iteration of the permutation is already in the partial permutation it is creating. If the letter is there, then it moves onto the next letter in the word and attempts to use that instead to continue creating its permutation. This was the only change to the design of the solution. After this, it takes this permutation list and checks for anagrams via the function I designed for this task.

*Part 2.2: Second Optimization*

This optimization called for the algorithm to stop recursion if the partial permutation being created is not a prefix of any word in the word set. This optimization called for the creation of a prefix set. To do this, I created two functions.

The first function, stringCombinations, is a recursive function that takes the user-given word and an integer, which serves as an index. The function creates an empty list and appends the letter at index 0 all the way up to but not including i + 1. The algorithm then recursively calls the function again with the user-given word and i + 1 as the arguments. The base case is when the index has reached the last letter because if we appended that word, we would be appending the original, user-given word. The function then creates a set out of this list to remove any repeats and returns it.

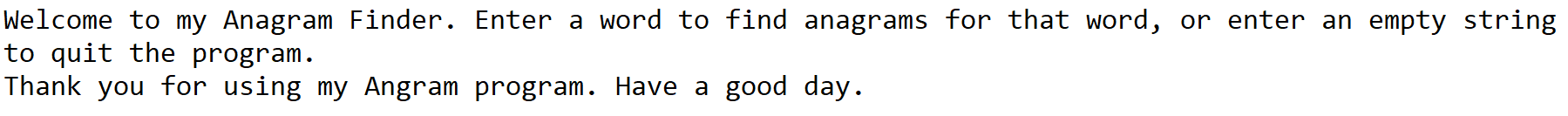
The second function, setPrefix, takes a set of words (the ones given from the words\_alpha.txt file) and returns the set of prefixes. For each word in the set passed, the function creates the prefixes via the stringCombinations function. As it does this, it updates the set of prefixes it is creating with the new prefixes. To implement the prefixes into the permutation function, I first added the prefixSet as an argument to the function and had the function check to see if the partial permutation it is creating is in the prefixSet argument. If it wasn’t, it would return an empty list.

**Experimental Results**

I tested all parts with an empty string, the three examples given in the lab report, and the word coconut.

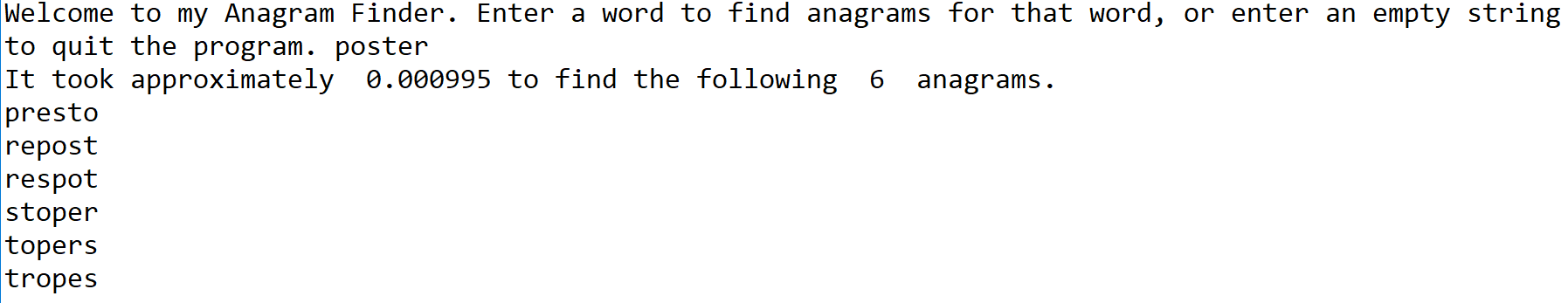
*Part 1:*

Empty String Example:



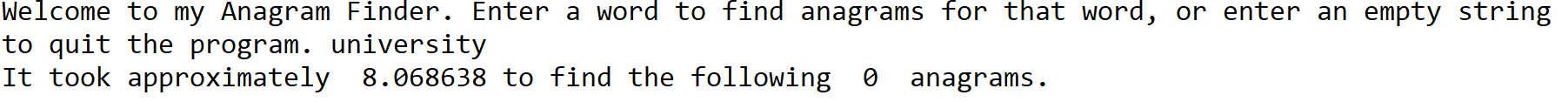
The empty string works as expected. When entered, the exit message is presented.

Poster Example:



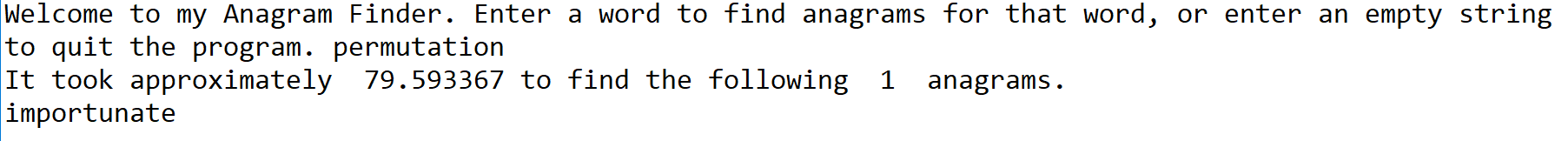
The program found the 6 anagrams of poster in a similar time frame to the lab handout.

University Example:



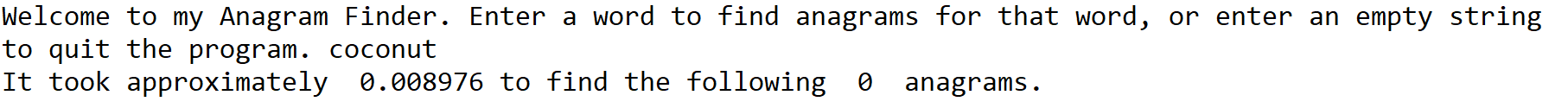
The program handled university similar to the expected results from the lab handout.

Permutation Example:



The program handled permutation similar to the expected results from the lab handout.

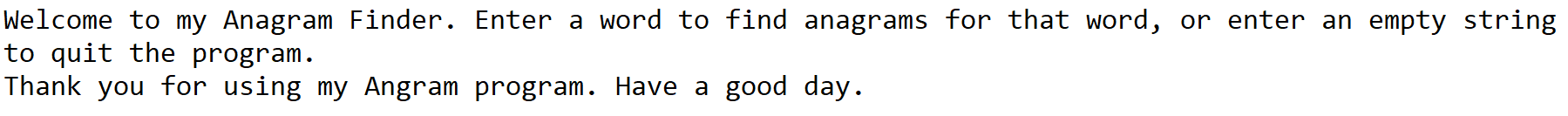
Coconut Example:



Here are the results for the coconut example.

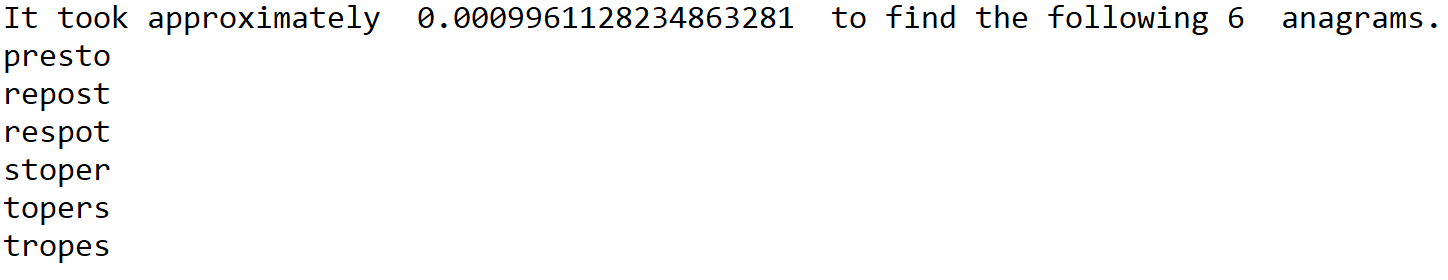
*Part 2.1:*

Empty String Example:

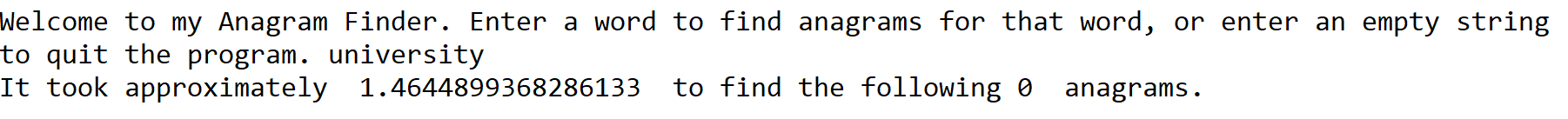


The empty string is handled as expected.

Poster Example:

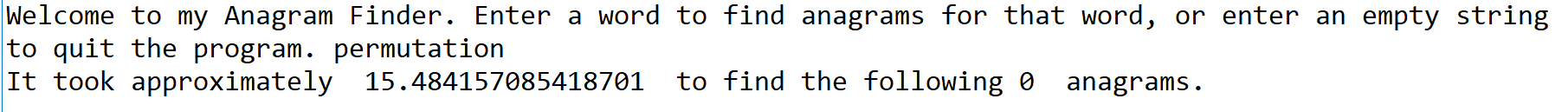
Poster renders similar results to the results from part 1.

University Example:



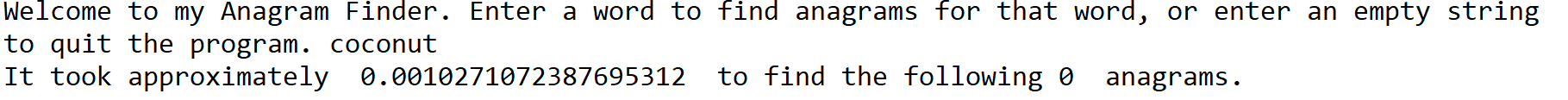
There’s a significant performance increase for the word university using only the first optimization.

Permutation Example:



There is also a significant performance increase for the word permutation using only the first optimization.

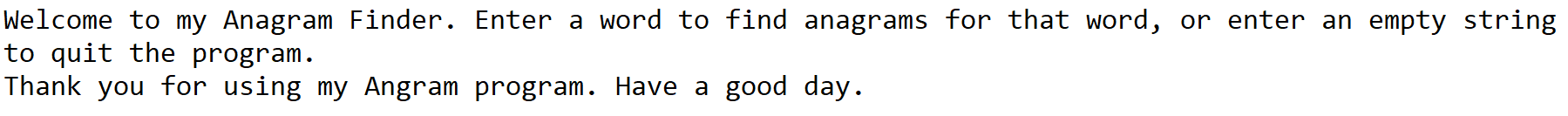
Coconut Example:



There’s a slight performance increase for coconut with this optimization.

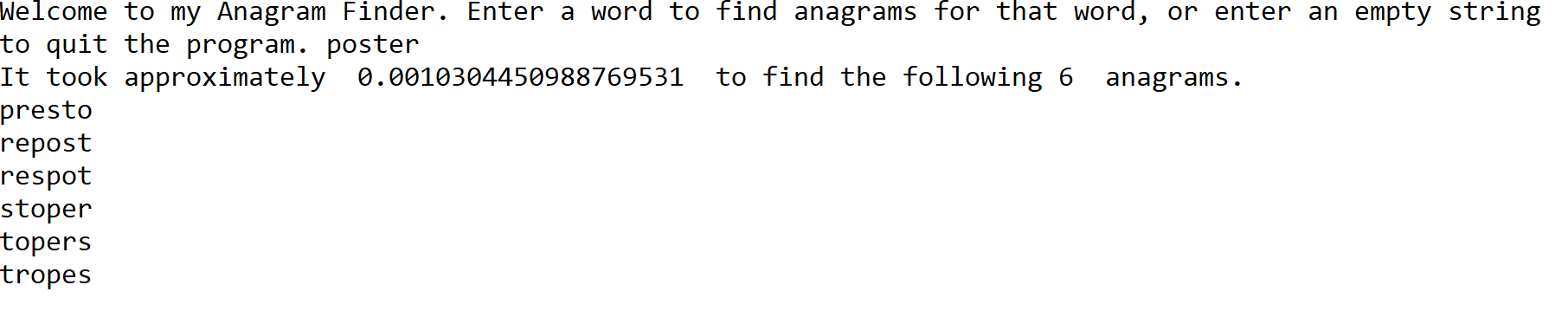
*Part 2.2:*

Empty String Example:



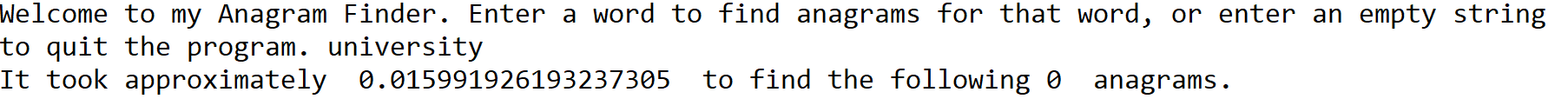
The empty string is handled as expected.

Poster Example:



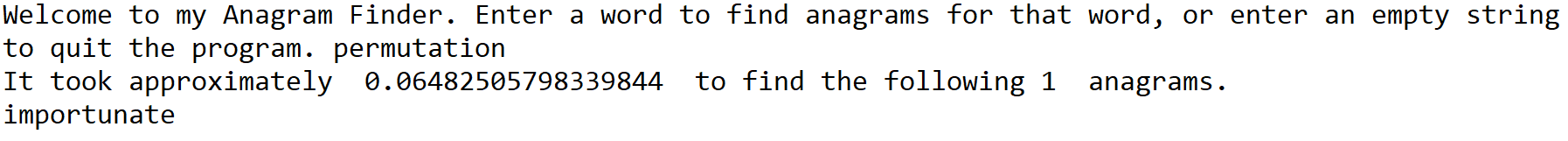
Performance doesn’t seem to be affected with the word poster.

University Example:



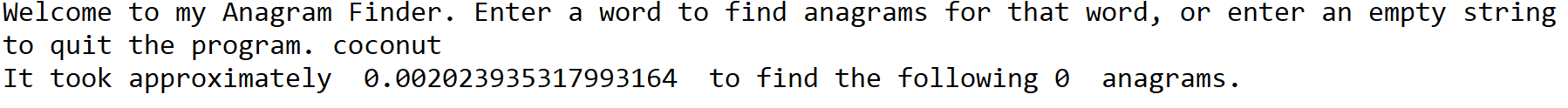
There was a significant increase in performance for the word University.

Permutation Example:



There was also a significant increase in performance for the word Permutation.

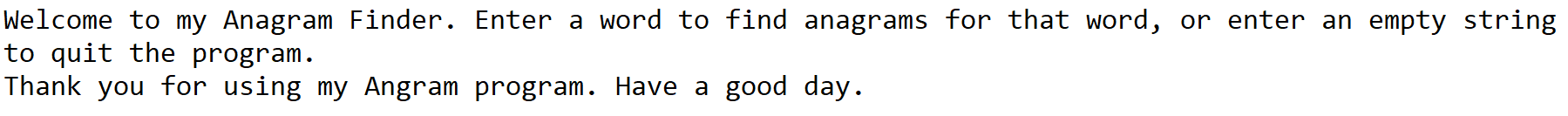
Coconut Example:



These are the results for the word coconut.

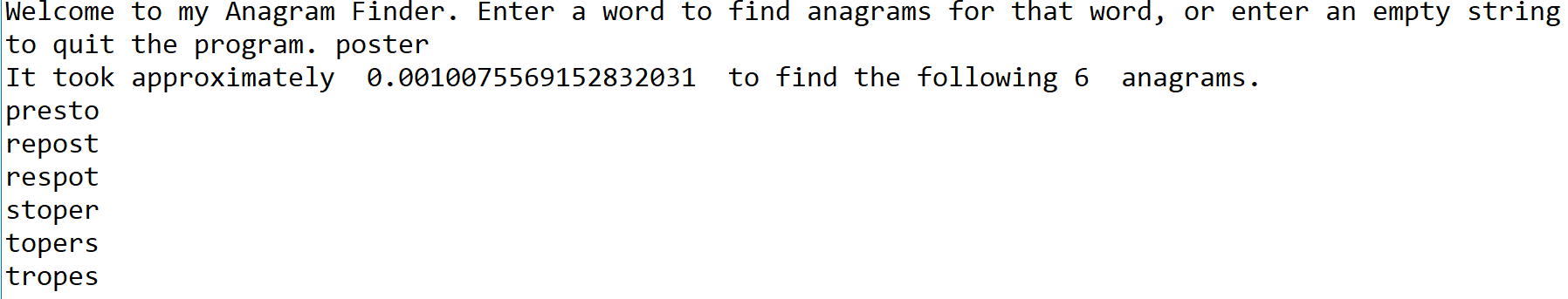
*Results with Both Optimizations Implemented:*

Empty String Example:



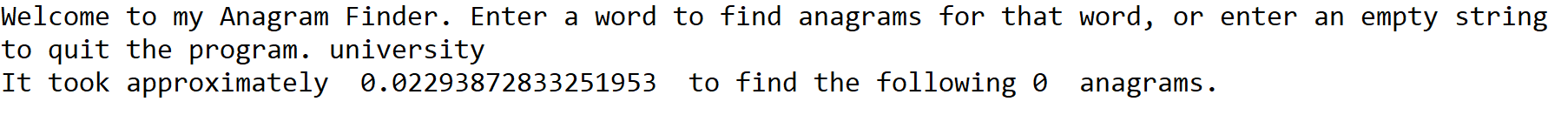
The empty string is handled as expected.

Poster Example:



Results are similar to previous ones.

University Example:



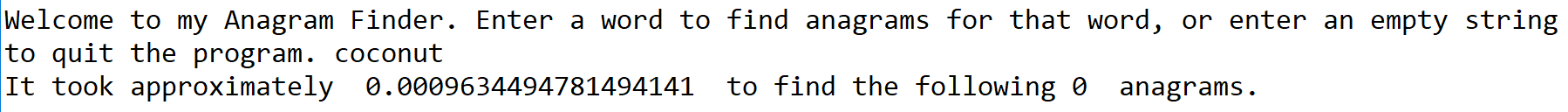
Results are similar to previous ones.

Permutation Example:



Results are similar to previous ones.

Coconut Example:



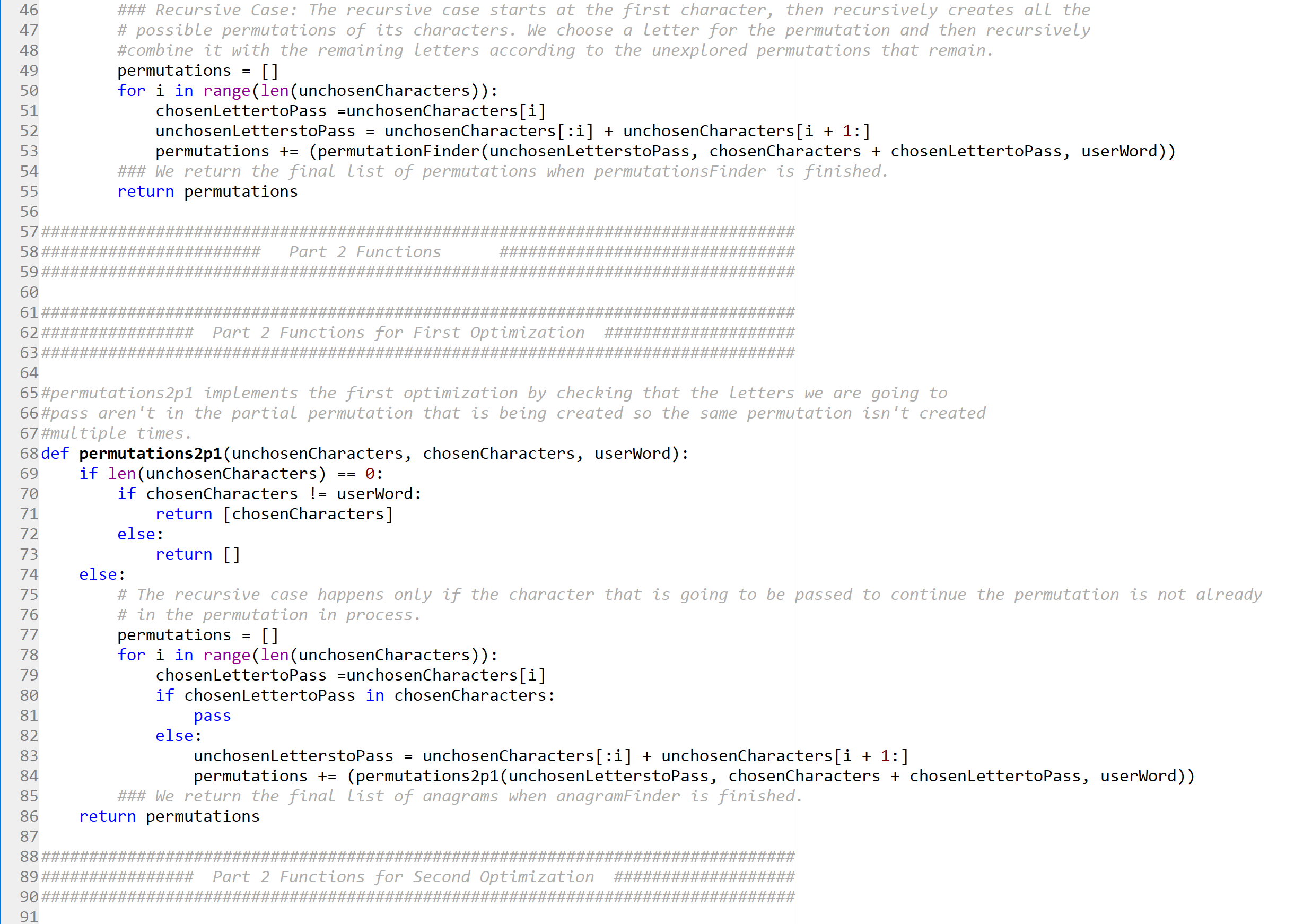
Results are similar to previous ones.

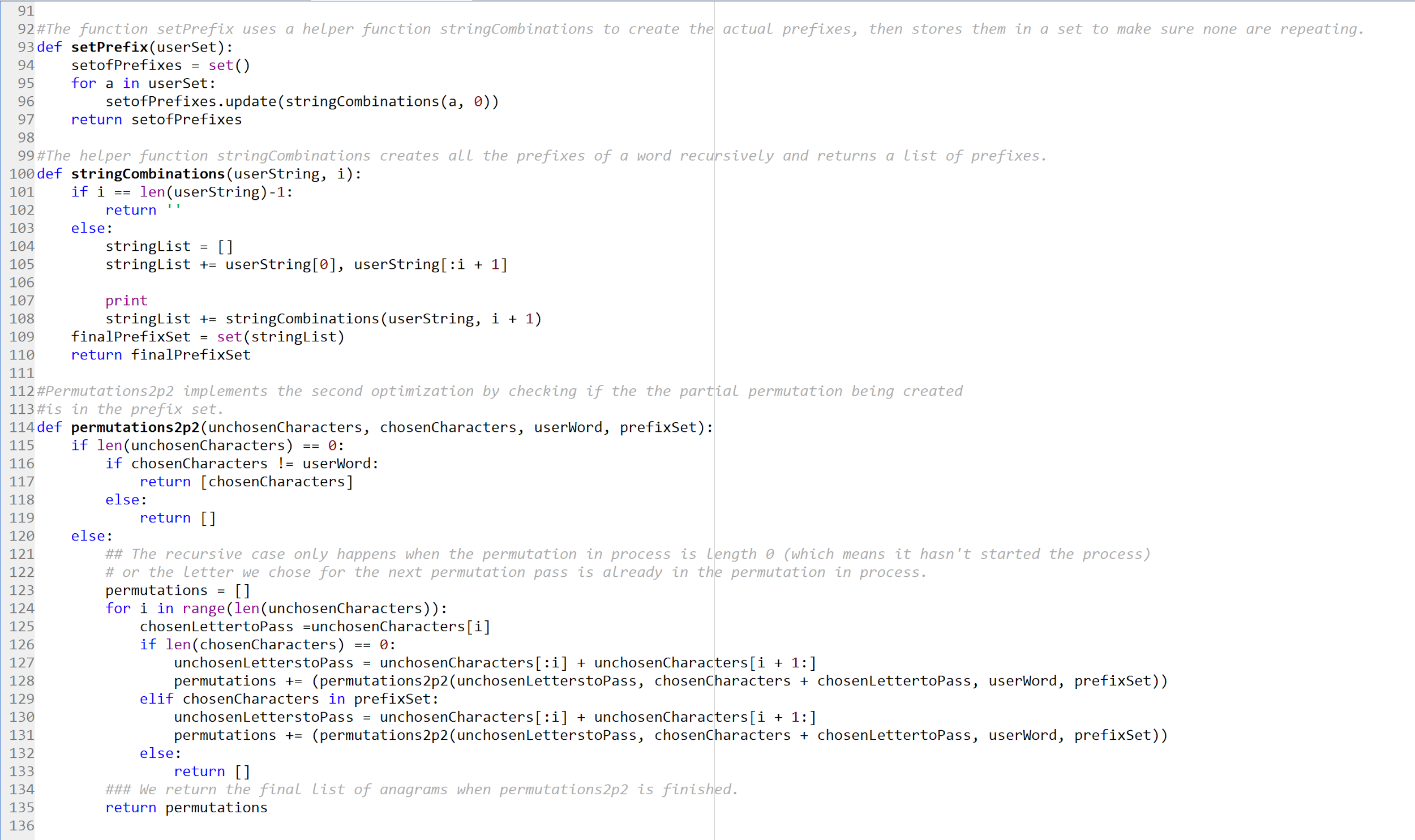
**Conclusion:**

In this lab, I learned that even though an algorithm works, it may not be efficient. It may be so inefficient that the algorithm won’t work for non-arbitrary examples. For example, with the first part of the lab, I ran the word “onomatopoeia” and it would not complete. But after implementing the optimizations, it would complete in less than a second. Thus, it is important to notice when your recursive function is making unnecessary calls and attempt to ameliorate this deleterious effort.

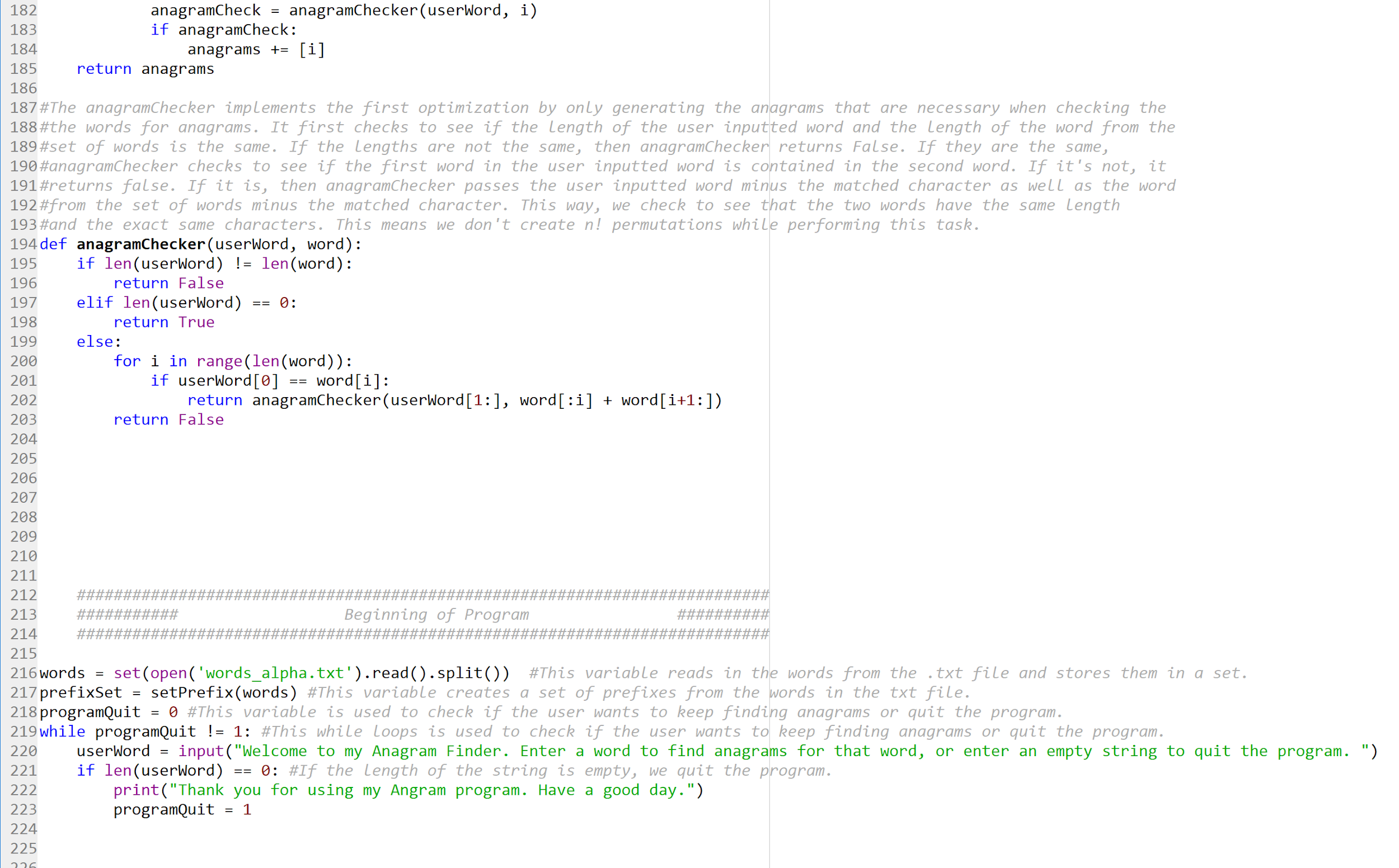
**Appendix**

****

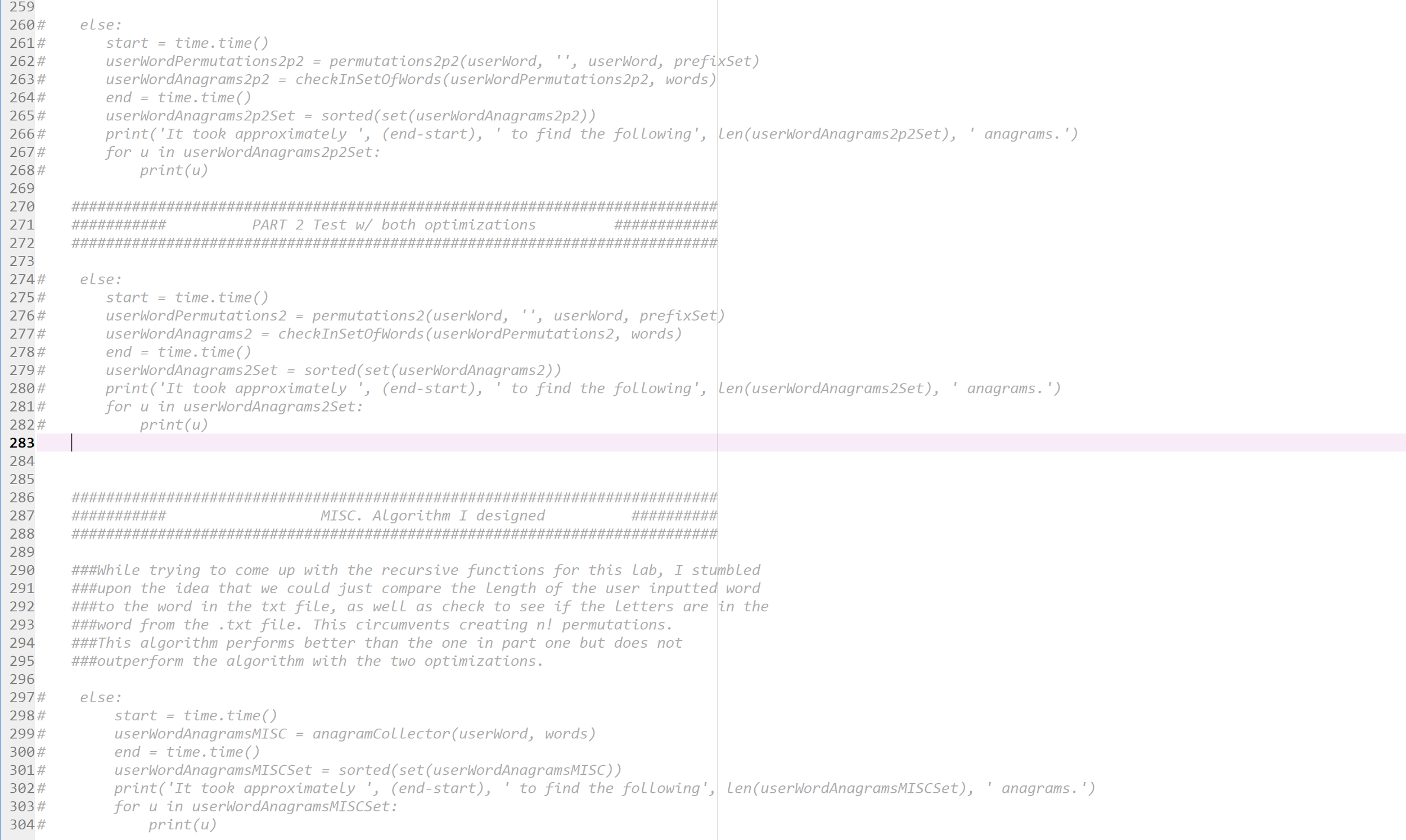
****

****

****

****

****

****

**Academic Honesty Certification**

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.