CS 2302 Data Structures
Fall 2019

Lab Report #1

Due: September 6th, 2019
Professor: Olac Fuentes
TA: Anindita Nath

Introduction

For this lab, we were asked to recursively find all the anagrams of a given word that could exist within a given list. To do this, we would have to read a file line by line, storing the words from each into a set. Then, we would have to generate a list of possible anagrams and simultaneously see of any of those are within the given list. Lastly, once this has been established, we need to optimize this process by only allowing partial anagrams that are remotely within the list to be considered and by avoiding repetition of partial anagrams so as to save running time.

Proposed Solution Design and Implementation

Part 1:

Since anagrams are just permutations of words, I needed to implement a recursive permutation method that will generate a list of all possible permutations. To do this, I had the method first consider the base cases: First, the word has no letters left, in which case, an empty list can be returned; and second, the word has only one letter left, in which case, the only possible permutation is that letter itself, so it returns a list composed solely of that letter. From there, recursive calls are required for words of two or more letters.

If the word has two or more letters, then all possible permutations must be considered. To do this, I had to have the method take each letter in the word and determine what permutations could be made using the rest of the letters. Thus, in a for loop, the method considers the current letter taken from the word and a "word" composed of the remaining letters, and then recursively calls itself to get a list of all possible permutations of that remaining "word." Thus, it can iterate through the list its recursive call returned and append a combination of the current letter taken at the front with each permutation of the remaining letters following it to the list the method will return. Then, once all is said and done, it returns the list of all possible permutations.

This works, but also returns every possible permutation without considering whether those permutations are in the list given initially from the read file. Thus, I made a separate *anagram()* method that is essentially like the *permutation()* method, but it does not have base cases, as it itself is not recursive, and it only considers the list of permutations generated by the *permutation()* method that contains permutations of length one less than the desired word. Thus, when appending the "word-length permutations," or anagrams, to the list it is to return, it can then check to see if a given anagram is actually within the list from the read file, and only append it to the list it is returning if that is true.

Part 2:

With the methods developed, the challenge was now to optimize them so that for longer words, not every single one of their permutations was considered. To do this, I implemented one of the two optimizations advised: Before calling the *anagram_opt()* method, I first read the file a second time, this time storing each and every "prefix" of each line's word into a distinct *prefix_set* set. From there, I include that set as an additional parameter to be passed eventually to the optimized *partition_opt()* method. This is virtually identical to the original *partition()* method, save for two aspects: Like the *anagram_opt()* method, it starts with the last letter and generates permutations from the front forward; because of this, it can now also detect if a given permutation is not a prefix of any word in the set, and thus does not add them to its own returning list of permutations, which should quicken the process altogether. I did not manage to

incorporate the checking of duplicate characters, as every attempt made would invariably prevent all the anagrams from being found.

Experimental Results

Part 1:

To check both methods in both parts, I simply used the example words provided in the instructions for this lab. Fittingly, the found anagrams are all accounted for, and the runtimes of these methods are around the amount of time the example indicated was appropriate.

```
Enter a word or empty string to finish: poster
Part 1
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.003499 seconds to find the anagrams.

Enter a word or empty string to finish: university
Part 1
The word university has the following 0 anagrams:
It took 11.278396 seconds to find the anagrams.

Enter a word or empty string to finish: permutation
Part 1
The word permutation has the following 1 anagrams:
importunate
It took 141.739430 seconds to find the anagrams.
```

Part 2:

For Part 2, since I did not properly implement both optimizations, the results are a bit lackluster. All the anagrams were still found, and the amount of runtime was significantly decreased, but not nearly to the level the examples indicated.

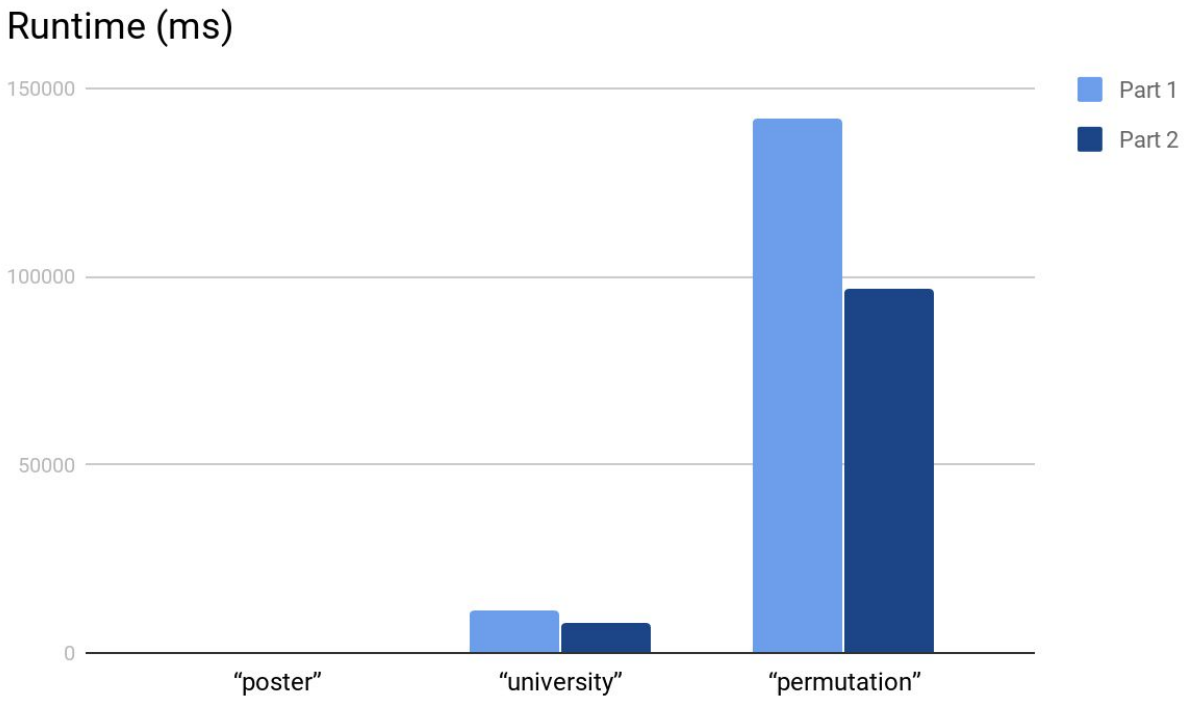
```
Part 2
The word poster has the following 6 anagrams:
presto
repost
respot
stoper
topers
tropes
It took 0.001826 seconds to find the anagrams.

Part 2
The word university has the following 0 anagrams:
It took 7.739762 seconds to find the anagrams.

Part 2
The word permutation has the following 1 anagrams:
importunate
It took 96.924631 seconds to find the anagrams.
```

Runtimes(ms)

Inputted Word	Part 1	Part 2
“poster”	3.499	1.826
“university”	11,278.396	7,739.762
“permutation”	141,739.430	96,924.631



As the results show, with longer words came longer runtimes, since more permutations had to be made the more letters there were. Also indicated is that the minor change I made does relatively optimize the procedure, though admittedly not remotely as well as would be expected professionally.

Conclusion

From this lab, I learned how to implement quicksort using recursion, stacks, and while loops as well as how to optimize quicksort. For quicksort and its optimization, while the partition method was the same swapping elements within smaller and smaller sections of the original list, the modified method was optimized because it only continued to recursively sort the smaller sections that were relevant to finding the desired element rather than every section. Regarding stacks and the while loop, I learned to implement a different partition method that, rather than track the elements of a subsection of the original list, instead tracks the indices at which these elements become relevant, thus sorting a list by its successive pivots. From all this, it can clearly be seen by my results that, of the various methods attempted, recursion (specifically necessary

recursion explicitly) is still the most efficient way to sort a list such that a desired element can be found.

Appendix

```
# Course: CS 2302 Data Structures
# Date of last modification: November 3
# Assignment: Lab 1 - Recursion
# TA: Anindita Nath
# Professor: Olac Fuentes
# Purpose: The purpose of this lab is to implement recursion
#          in order to find anagrams of a given word in a list of words.

import time

def permutation(word):
    #If word has no letters, no more permutations can be made
    if len(word) == 0:
        return []
    #If word has one letter, only one permuatation (itself) can be made
    if len(word) == 1:
        return [word]
    #If word has two or more letters, recursive calls must be made to find permutations
    l = []#List to store all possible permuatations
    #Take each letter of the current part of the word
    for i in range(len(word)):
       m = word[i]
       #Take rest of word that is not current taken letter
       restWord = word[:i] + word[i+1:]
       #Recursively generate list of all possible permutations
       for p in permutation(restWord):
           #Put taken letter in front of each permutation and append to list to return
           l.append(m + p)
    #Return list of all possible permutations after paired with every letter taken
    return l

def anagram(word, word_set):
    anagram_list = []#List to hold necessary anagrams
    #Take each letter from word and place at beginning of potential anagram
    for i in range(len(word)):
        m = word[i]
        #Take remaining letters and begin to form permutations
        remLet = word[:i] + word[i+1:]
        for p in permutation(remLet):
            #Add given permutation to anagram list if it follows current leading letter in word set
            if m+p in word_set and m + p != word:
                anagram_list.append(m + p)
    #Return sorted list for alphabetical order and no duplicates
    return sorted(set(anagram_list))

def permutation_opt(word, prefix_set):
    #If word has no letters, no more permutations can be made
    if len(word) == 0:
        return []
    #If word has one letter, only one permuatation (itself) can be made
    if len(word) == 1:
        return [word]
    #If word has two or more letters, recursive calls must be made to find permutations
```

```
    l = []#List to store all possible permuatations
    #Take each letter of the current part of the word, starting from last letter
    i = len(word) - 1
    while i > -1:
    #for i in range(len(word)):
       m = word[i]
       #Take rest of word that is not current taken letter
       restWord = word[:i] + word[i+1:]
       #Recursively generate list of all possible permutations
       for p in permutation_opt(restWord,prefix_set):
           #print('p(',restWord,'): ',p,end = ' ')
           #First check if given permutation is remotely within list
           if p in prefix_set:
               #print('Taken!',end=' ')
               #Put taken letter in front of each permutation and append to list to return
               l.append(p+m)
       i-=1
    #Return list of all possible, viable permutations after paired with every letter taken
    return l

def anagram_opt(word, word_set, prefix_set):
    anagram_list = []#List to hold necessary anagrams
    #Take each letter from word and place at beginning of potential anagram
    i = len(word) - 1#Start from last letter and work backwards
    while i > -1:
    #for i in range(len(word)):
        m = word[i]
        #Take remaining letters and begin to form permutations
        remLet = word[:i] + word[i+1:]
        for p in permutation_opt(remLet, prefix_set):
            #print('p:',p)
            #Add given permutation to anagram list if it follows current leading letter in word set
            if p+m in word_set and p+m != word:
                anagram_list.append(p+m)
        i-=1
    #Return sorted list for alphabetical order and no duplicates
    return sorted(set(anagram_list))

if __name__ == "__main__":#main method
    word_file = open("/Users/johanncampos369/Downloads/words_alpha.txt", "r")#access words_alpha.txt
    word_set = set()#create empty set

    for line in word_file:#read each word into empty set
        word_set.add(line[:-1])
    while(True):
        word = input('Enter a word or empty string to finish: ')#User Prompt
        if word == '':
            break
        #Unoptimized
        print('Part 1')
        start = time.time()
        anagram_list = anagram(word, word_set)
        end = time.time()
        print('The word',word,'has the following',len(anagram_list),'anagrams:')
        for i in range(len(anagram_list)):
            print(anagram_list[i])
        print('It took %.6f'%(end-start),'seconds to find the anagrams.')
        #Optimized
        print('Part 2')
        word_file = open("/Users/johanncampos369/Downloads/words_alpha.txt", "r")#access words_alpha.txt
        prefix_set = set()
        for line in word_file:#read each word into empty set
            for i in range(len(line) - 1):
                if line[:i] not in prefix_set:
                    prefix_set.add(line[:i])
        start = time.time()
        anagram_list = anagram_opt(word, word_set, prefix_set)
        end = time.time()
        print('The word',word,'has the following',len(anagram_list),'anagrams:')
        for i in range(len(anagram_list)):
            print(anagram_list[i])
        print('It took %.6f'%(end-start),'seconds to find the anagrams.')
    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