CS 1501 Assignment 1: Anagram Solver

By: Andrew Beers

Email: acb75@pitt.edu

Anagram Solution

Part A of Assignment 1 required to write a recursive backtracking algorithm to solve all possible anagram solutions of a given input string. The approach used for this problem stemmed off of the TA’s suggestions during recitation. The algorithm begins by initially calling a function with a blank StringBuilder and a StringBuilder containing the input string. Also the current number of words is added as a third parameter, which serves later on when finding multiple word solutions. The algorithm begins by adding a single letter from the input string to the solution string. The solution string is sent to the searchPrefix() method to find out if it is a word, prefix, both, or neither.

If the solution string is a prefix, a recursive call is made to start the process again, this time with the prefix as the solution string, the input string without the prefix letters as the new input string, and same number of current words parameter. If the solution is a word, there are two cases to consider. If the solution contains all of the letters, the algorithm is over and the solution is added to the solutions array. If the current word does not contain all of the letters, a space is appended to the current string, and a recursive call is made this time with the current string, the letters remaining from the input string, and with the number of words increased. This approach requires a different action if the number of words is greater than 1. If the number of words is greater than one, searchPrefix() should only operate after the space on the recursive call. With this approach, multiple word solutions can be achieved in a manageable way. If the solution is not a prefix we take the letter back and try the next letter in line.

Issues

I had many issues programming this assignment including understanding how to remove letters from the input string on the recursive call but retain the original string in the current call, how to deal with subtracting letters off the solution string I didn’t want anymore, and coming up with a solution for multi word anagrams.

My first issue of removing letters, resulted in me trying every other letter in the solution string as a beginning letter and skipping the in between letters. I found out that calling deleteCharAt on a StringBuilder, modifies the StringBuilder itself, even if you assign it to another variable. I realized this by using print statements, all of my debugging efforts involved many print statements, perhaps the main downfall of using a text editor to develop code. To solve this issue I used another stringBuilder to hold the previous value of my remaining input string across recursive calls.

The second issue of removing letters from the solution string was solved almost simultaneously to issue one, I was getting undesirable results when looping through letters, and changing around when I deleted the character, based on the searchprefix return value, and my approach to issue one, this was solved by deleting a character only if the solution string was not a prefix or a word.

The last issue of multiple word solutions took me a long time to come up with my initial solution of running the program the same amount of times as the number of letters in my solution. Even though this seemed to find most of the solutions, it didn’t have any memory of which words went with which, which in turn had me just printing out all the smaller words in order, since I couldn’t think of a way to piece the smaller words together in a way that I didn’t duplicate or forget any letters of the original input string in the combination of solution strings. I used many print statements and ended up calling my recursive backtracking algorithm with then number of words I was currently looking for. If I found a word, still had letters left, and the number of words I was looking for was greater than 1, I would recursively call anagram solution method again, but decrease the number of words I was looking for by one, and use the remaining letters as my input string. This is kind of a recursive algorithm inside of a recursive algorithm, which had me confused, but eventually started giving some solutions I was happier with. However this solution was not complete because I couldn’t sort, or combine these small words. I moved to a different approach, that was explained in the first section.

Runtimes

I did a little bit of extra credit here, adding a timer to my program, running the program 5 times, taking the average then plotting all points for both the DLB and MyDictionary data structures. For some of the test files, it took too long to run the MyDictionary version of the code, so I’ll have to approximate for those times based on the others.

There were five test files provided, each run for each file is listed below in the tables/charts.

I have compressed my data to what is shown below.

Each test file was run 5 times, and the data point plotted here is the average duration over those 5 runs, for each of the 5 test files.

Each test file was run 5 times, and the data point plotted here is the average duration over those 5 runs, for each of the 4 test files.

Note: Test5.txt did not finish in under 4 hours so I did not have time to run 5 times before submission.

Unfortunately, I was only able to plot 4 points here because the last test file took too long to gather data from, So this is my function for converting a time from DLB to what it would be for MyDictionary. If the graph kept the same slope as it takes from .4-.8, test file 5 should take about 17000 seconds or about 4 hours and 45 minutes. The average dictionary load time for DLB and MyDictionary was .7 and .8 seconds respectively.

I found that the number of words in the dictionary would directly affect the dictionary load time. It has the potential to affect the search time, if we end up traversing each level the entire DLB it would then affect the asymptotic runtime, however I would not expect this to be the case in the normal case. This differs from the MyDictionary Class as the searchPrefix() method’s asymptotic runtime is affected by the number of words in the dictionary.

The number of characters in a word will affect the asymptotic runtime by increasing the amount of multiple word solutions, this depends on the assumption that both the dictionary used and input strings are “traditional”. If I am using a standard dictionary and my strings consist of different letters , including vowels, I can expect the runtime to increase. The runtime increasing with the number of characters depends on the possibility of words being created to increase, which in the end depends on the actual characters in the string itself.

The number of solutions found may increase the runtime of the program given that the solutions must be sorted at the end of the program. Sorting is not constant time, and with many solutions, this would make the asymptotic runtime of the program grow.

As the number of words in a solution grows, it similarly requires sorting. With more and more multiple word solutions, the algorithm used to find anagrams basically repeats itself, as in we can asymptotically add up the little recursive calls to get a final runtime. This bound is however the length of the string, as if we assume that single character words are the smallest possible words, a string of length N can at most have N words to it, so we multiply our runtime by N in the worst case.