Samuel Birus

CS 1501: Algorithms

Dr. Ramirez

Summary of Anagram Solver Algorithm

To set up the program I first thought about how to find all of the permutations for a given string of letters. The easiest way to do so was to use recursion. The recursive method took 5 parameters: a prefix string, a suffix string, a solvedWords array, a phrase array, and the dictionary. The solvedWords array stored the permutations that were words in the dictionary. The phrase array was used to solve multiword solutions. It was an effective method to solve the multiword solutions except that I could not get them to output to the form I wanted to store into the solvedWords array. Every phrase/single word had a set of brackets surrounding them which made sorting the array by size something that I could not do. However the solvedWords array was sorted alphabetically with no regard to different sized permutations.

Within the permutation generator (called genPerm in my file) an if - else if - else statement checked for three things. First whether or not the prefix length was 0, meaning it had to add the first letter to the permutation. This prevented checking a blank string against the dictionary which would return neither a word or prefix and thus end the program. A for loop was also added here to rotate through all the possible beginning letters. Second the permutation would be built normally until the suffix was 0. This is where pruning was implemented so that I didn’t get a run time of O(n!). Every prefix was checked against the dictionary, if it returned neither then the algorithm returned up a level and didn’t try to keep adding words. In the case of a prefix the algorithm recursed again. For a word, the word was added to the phrase array and then recursed again. The prefix and word case was slightly more challenging as it had to first try to find it as a prefix, then when it backtracked it would then try it as a word. Thirdly if the suffix length was 0 then it had to add the phrase to the solvedWords array. This section had to be implemented to prevent an out of bounds exception occurring in the previous part when iterating through the each letter of the suffix.

There were a couple decoding/debugging issues that I faced along the way. The main debugging issue was figuring out to do to prevent the out of bounds exception on the string suffix. After some thought the if the suffix length was 0 loop was determined to be the best option. This also helped me organize my thoughts when starting out to implemented the multiword solutions section.

A table follows of the approximate run times for each test file.

|  |  |  |
| --- | --- | --- |
| Test File | MyDictionary time | DLB time |
| Test1 | 40 sec | < 1 sec |
| Test2 | 3.5 min | 1 sec |
| Test3 | 40 min | 25 sec |
| Test4 | > 2 hrs | 2 min |
| Test5 | > 2 hrs | 40 min |

The fact that some of the MyDictionary times and the test5 DLB time were really high could mean that my implementation of multiword solutions was inefficient for long anagrams.

For the worst case the asymptotic analysis of the MyDictionary would be O(n2) since it iterates through the arraylist and then through the string. For the DLB it would be at worst O(kn) where n is the length of the string and k is the number of possible sibling nodes at each level. This is rarely achieved though as the pruning in the algorithm prevents a lot of unnecessary searches.