Imani Palmer

6/3/2013

Anagram Solver

An anagram is the rearrangement of the letters of a word or a phrase into another word or phrase. Computers have been used to enhance the efficiency of finding anagrams in words and phrases. The solution described below takes in an input of letters. The output is a collection of words or phrases, sorted from the phrase with the fewest words to the one with the most words, in alphabetical order. For example, with the letters given: *itsp*. This anagram program generates the following list of words: pits, spit, tips. There are two possible design methods in that would solve this problem.

The first approach is called brute-force search. Brute-force search finds a solution to a problem by considering all potential solutions and selecting the correct one. For example, to find all of the anagrams of the letters: *itsp*. First, find all of the permutations of the letters where *i* is the first letter formed: *itsp, itps, ispt,istp, ipst, ipts*. This continues for all the possible permutations of letters where *t, s,* and *p* are the first letter. As each possible word is formed there is a check with the dictionary to see if this word is actually a word. If it is a word, it becomes a part of the output. If it is not a word, program continues. The runtime of this approach is bounded by the number of possible solutions, that for N letters there are *N!*. In this example there are 4 letters there are 24 possible solutions. The number of solutions to the problem tends to grow quickly, as the difference between a 4 letter word there are 24 possible solutions and 8 letter words there are 40,320 possible solutions. There is another way to solve for anagrams programmatically.

The second approach incorporates backtracking and recursion. Backtracking allows the solution to proceed forward to a solution until it becomes apparent that no solution can be achieved along the current path. At that point, one undoes the solution to the point where they can again proceed forward. Recursion is a method where a solution to a problem depends on solutions to smaller parts of the same problem. In the beginning, the input, *itsp,* is read. Then, the method *findAnagrams* is called. The input is passed in as an array of chars. This array of chars is saved under a local copy named local. Next, a for-loop goes through each letter in the original array. The first character *i* is appended to a StringBuilder and is removed from the local array. Then, there is a call to the *searchPrefix* method that compares the StringBuilder to the given dictionary and returns certain values. If the StringBuilder is not a word or a prefix within the dictionary, 0 is returned. If the StringBuilder is a prefix but not a valid word, 1 is returned. If the StringBuilder is a word but not a prefix, 2 is returned. If the StringBuilder is both a word and a prefix, 3 is returned. In case of StringBuilder, *i,* the *searchPrefix* method returns 1. Since 1 is returned recursion takes place and the *findAnagrams* method is called again, passing in the StringBuilder, *i,* and the local copy of the array without the character *i.* Once again, the local copy of the array is set to local, and the for-loop is entered. The next character *t* is appended to *i* and StringBuilder, *it,* is formed. *searchPrefix* is called and returns the value three. Now, there is a check to see if all the letters are used. If all the letters are used then the solution is added. However, not all of the characters are used. Recursion is used, *findAnagrams* is called. When *findAnagrams* is called the passing in the StringBuilder *it* and the local copy of the array without the characters *it.* The local copy of the array is set and the for-loop is entered. The next character *s* is appended and *its* is formed. The *searchPrefix* method is called and the value zero is returned. This is where backtracking occurs. The addition of *s* made this neither a word nor a prefix and therefore *s* is unnecessary. *s* is removed from the StringBuilder and the StringBuilder backtracks to *it*. Then, the next character *p* is appended and *itp* is formed. The *searchPrefix* method is called and the value zero is returned. This is neither a valid word nor prefix. *p* is removed from the StringBuilder and the StringBuilder backtracks to *it.* Backtracking removes that char from my candidate solution and stop going down a path of impossible permutations. This eliminates many of the execution paths travelled by the brute-force approach. Recursion and backtracking improves on the brute-force approach.

The main challenges I faced when developing this program was the ability to remove duplicates and sort the solutions alphabetically. In order to accomplish both of these challenges, I used another data structure the HashSet. I added my ArrayList of solutions to a HashSet. The HashSet automatically removes all duplicate items. In order to get my solutions in alphabetical order I used the HashSet method *sort*. A better possible implementation method would be to check for duplicates inside the *findAnagrams* function. There are two ways to implement this anagram solver.

The first method is the MyDictionary implementation. MyDictionary is based off the data structure ArrayList. The add method: adds a new string to the end of list. If a string is out of order, the list is sorted. The search method: iterates through an ArrayList, until the end or until the key is passed up. When the key is passed up, iterate through the key and the current string in the ArrayList character by character. Then, check if the string had ended in the ArrayList, stop using the string. The benefits of a sorted ArrayList leads to an decrease in the runtime of the *search* method from Theta(N) to Theta(lgN). However, the amount of space taken up by this data structure is equal to the number of words contained in the ArrayList. An improvement on the MyDictionary implementation would be a decrease in space.

The second method is the De La Briandias Tree (dlb) implementation. In a dlb implementation, the data structure is a node is a linked list of nodes. The advantage of a dlb is that it can save much space compared to MyDictionary. A picture of a dlb node is demonstrated.

character (bit pattern)

reference to child node (next level)

reference to sibling node (same level)

The search method: start from the first character of the given string. Sequentially search the linked list of sibling nodes pointed by the root. If the character is found, follow the child link and repeat the same procedure with the next character in the string and the child’s linked list. If the search fails before reaching the end of the string, the string is neither a word nor a prefix. If the last character in the string is reached, continue search its child’s link list. If a string terminator is found, the search string is contained in the DLB as a word. If some other characters are also found, it is contained in the DLB and also it is a prefix of some strings. If only other characters are found and the string terminator is not found, it is a prefix of some strings contained in the DLB. An illustration of a DLB is pictured below containing these words: app, apple, bat, and back.

l

/

e

/

$

/

a

/

t

$

/

Root

a

p

/

p

/

$

b

/

c

/

k

/

$

/

The worst case number of character comparisons required for a search is number of key level of characters multiplied by the number of valid characters. The average search time is the number of levels to get to the end of the key. I was unable to complete the DLB implementation of this program. However, I suspect that the first three runtimes of this program would run in times of under one minute because of the faster search method. I believe that test four and test five would run for longer because of the number of possible solutions is really high. These numbers are demonstrated below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Test1** | **Test2** | **Test3** | **Test4** | **Test5** |
| MyDictionary | 1:22 | 4:43 | 10:23 | 15:45 | 30:00+ |

The runtime of grows as the program travels through each test. The runtime grows as the number of solutions grows. The number of solutions for test one was ten. The number of solutions for test two was seventy-one. As the number of solutions grew by seven fold, the runtime also increased. However, in test three the number of solutions was only about sixty. I believe the explanation for this phenomenon is in the possible number of solutions in the results of monty python. I believe that monty python has many one word and two word solutions however, since none of them materialized as actual solutions this accounts for the increase in runtime. Another oddity is in test four with the number of solutions around three hundred and fifty and test five with the number of solutions to be three hundred and forty. These runtimes also vary for a similar reason. That an addenda has many possible solutions and first must be checked before the ability to find true solutions of three and four words. This would ring similar for the DLB implementation. In conclusion, the various implementation methods have their advantages and disadvantages, where the number of solutions determines the runtime.