DAA Assignment 3

Nitin Raj Singh IIT2016132
Chirag Meel ISM2016006.
Lekhika Dugtal ITM2016008
Saurabh Kumar IWM2016502

1 Problem

Write an efficient algorithm to generate an (n x n) matrix filled with randomly generated English alphabet characters. Find out all diagonal locations containing valid English words. Do the necessary experimentation and analysis with your algorithm.

2 Introduction and Literature Survey

In this problem, first we have to generate a n*n matrix which will be filled with randomly generated english alphabetic characters (A-Z). Then we have to permute these random english characters (in either way horizontally or vertically or diagonally or backwards) to find valid words by checking them whether those words are present in dictionary or not.All the invalid combination of characters will be ignored while the those present in dictionary will be printed

One of the most naive algorithm of searching a word in the dictionary would be to store all the words present in the dictionary in some data structure like set or vector in c++ and then find if it is present in that set. We can use binary search on that data structure for searching the word. For applying binary search one of the requirement would be to keep the words in sorted order as present in a dictionary. What we notice in this algorithm is that as the dictionary can consist of thousands of words, the space complexity would be very large and also quite high time complexity.

Due to the above constraints on space and time, it would be better to use a trie tree.

3 Algorithm Design

3.1 Algorithm Approach

A Trie is a special data structure used to store strings that can be visualized like a graph. It consists of nodes and edges. Each node consists of at max 26 children and edges connect each parent node to its children. These 26 pointers are nothing but pointers for each of the 26 letters of the English alphabet A separate edge is maintained for every edge.

Strings are stored in a top to bottom manner on the basis of their prefix in a trie. All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

The approach would be to form words with the diagonal elements and pass substrings of the formed string for checking if it is present in the trie or not .

The words formed with diagonal elements start with first row or first column so we form words starting with first row or first column and go on concatenating diagonal elements to it till it reaches the end.Now we can form all substrings of diagonal strings ending with same character and check whether they are present in the trie or not.

In the trie during checking if a string is present or not we start with the root node and keep moving down the node as we encounter the next character.When we encounter the last alphabet of the string we check if that node is marked or not, if it is marked then the word is present otherwise the word is not present. In middle if any of the node is not present then we conclude that the word is not present.

Searching in a trie is like checking the substrings of the string being checked beginning with the same character .This way all the substrings of the strings formed with diagonal elements will be checked using the search function and the valid words present in the dictionary can be found with the help of trie.

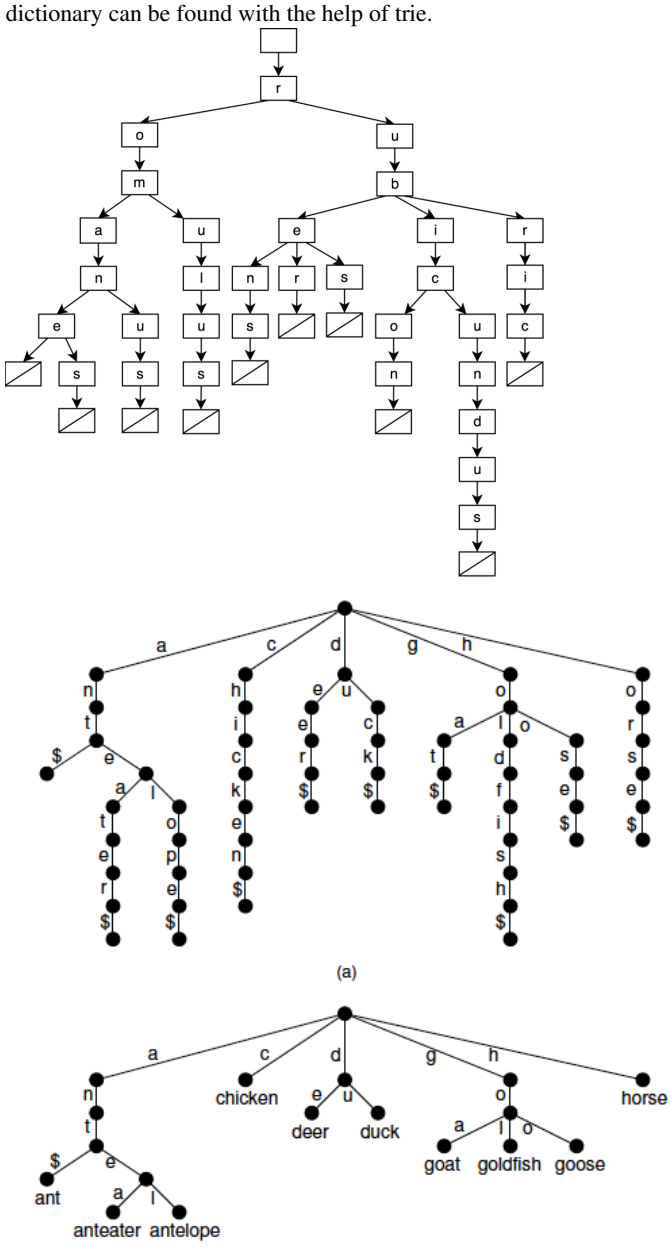

Fig- 1 The figure showing how the words are stored in a trie.

3.2 Algorithm Pseudo Code

```
/* Initialisation                                   */
Node newnode();
Node tmp = new Node();
for i ⇐ 0 to 26 do
    tmp → nextchar[i] ⇐ null;
end
tmp → valid ⇐ false;
return tmp;

/* Insertion of node through function               */
Node insert ( root, word );
len ⇐ length(word);
Nodeptr ⇐ root;
for i ⇐ 0 to len do
    Val = word[i]- 97;
    if ptr → nextchar[Val] = null then
        Node tmp = newnode();
        ptr → nextchar[Val] ⇐ tmp;
    end
    ptr ⇐ ptr → nextchar[Val];
end
ptr → valid ⇐ true;

/* funtion to find index of node                    */
Void findIndex ()
 for j ⇐ 1 to n do
    Cnt = 1;
    for i ⇐ n to 1 do
        Word[cnt] ⇐ mat[i][j];
        cnt ⇐ cnt + 1;
    end
    for i ⇐ 1 to n do
        search( root, word, i, j );
    end
end

Void search( root,word,row ,column,pos,len)
 node ∗ ptr ⇐ root;
for i ⇐ pos to count do
    if ptr → nextchar[A[i] − 97] ≠ NULL then
        ptr ⇐ ptr → nextchar[A[i] − 97];
        if ptr → valid = 1 then
            print"Wordstartingatrow + 1, column + 1;
            for j ⇐ pos to i do
                printA[j];
            end
        end
    end
    else
        return;
    end
end
```

4 Analysis

Time complexity analysis of insert function

The complexity of the insert function is totally dependant on the length of thw word being inserted.

$$T = 1+1+(3+1+2+k+2+2)*lengthOfWord+1$$

$$T \propto 3+(8+k)*lengthOfWord$$

Time complexity analysis of the search() function

Worst Case :

The best case would be when the word we are searching for is there in the dictionary and thus the function will have to traverse the tree for all the length of the word

$$T = 1+(3+3+3+1+1)*lengthOfWOrd$$

$$T \propto 1+11m$$

Best Case :

The best case complexity will be when the node will be null in the first if condition and it will return from there.

$$T = 1+3+3+1$$

$$T \propto 8$$

Time complexity analysis of the findIndex() function

The findIndex() function iterates through all of the diagonals of the matrix to search for the words present.

$$T = ((3+3+1+1+1)n*(n-1)/2+(3+k)n*(n-1)/2+(3+1+1+1+1)n)*2$$

$$T \propto (12+k)n*(n-1)/2+7n/2$$

Where k = time taken by the search() function.

Worst Case :

In worst case the search function will return in the first iteration only, so k= 11m, where m = length of the word which would be proportional to n, so we can replace it with n,

$$T \propto (12+n)*n*(n-1)/2+7n/2$$

$$0(n*n*n)$$

Best Case : In the best case the time taken by the search function is constant, which is equal to 8, so the time complexity becomes

$$T = (12+11)n*(n-1)/2+7n/2$$

$$T \propto 23n*(n-1)/2+7n/2$$

$$\Omega(n*n)$$

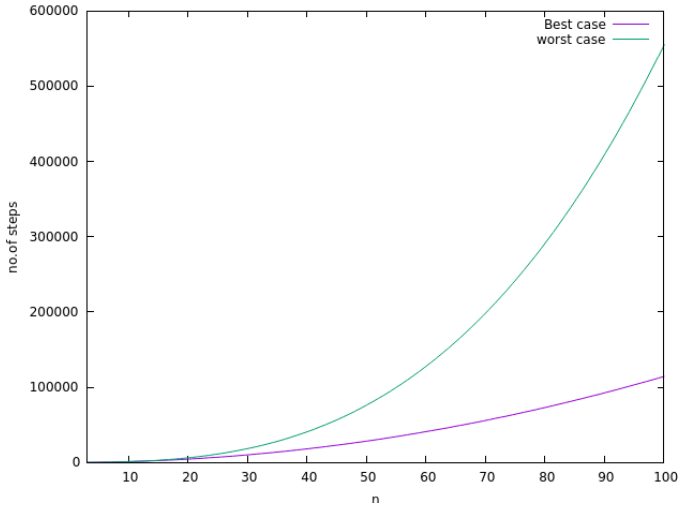

Fig- 2 The theoretically obtained time complexity graph for best and worst case.

5 EXPERIMENTAL STUDY

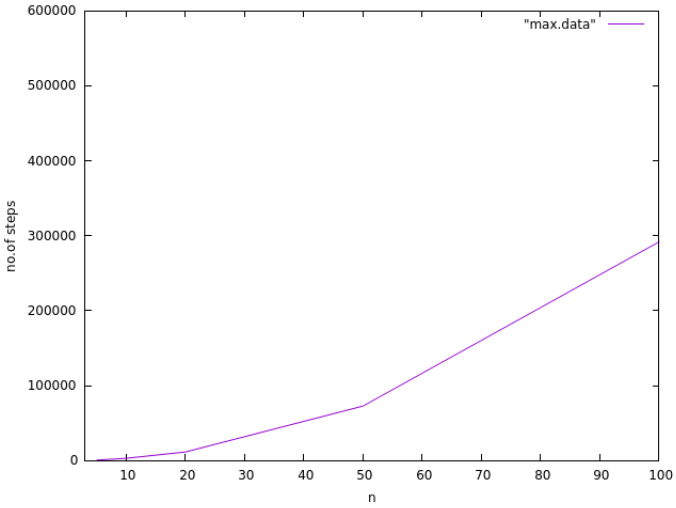

Fig- 3 The time complexity obtained for an average case.

On creating a random matrix using the rand function and doing the search operation for different values of n (i.e. for different size of the matrix), we find that the time taken is between the best case and the worst case, as expected.

6 DISCUSSION AND CONCLUSION

- Though the complexity of our algorithm is around n*n*n where n is the size of the matrix, yet it is pretty fast because of the trie tree. We see that trie tree is pretty efficient, it searches for a word in O(m) where m is the length of the word.
- Also the space complexity is less than the total sum of characters of all the words when the number of words are pretty large. A common application of a trie is storing a predictive text or autocomplete dictionary, such as found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries

7 REFERENCES

- How to solve it by Computer By R.G. Dromney
- Introduction to Algorithm 3rd Edition By Thomas H.Cormen
- overleaf.com
- tex.stackexchange.com
- latex.org