02/06/2023

Q1 Longest common prefix

i/p → [flower, flow, flight]
o/p → fl

Optimal solution does not come from tries. Optimal solution can come from brute force.

flower
→ Prefix (Reading from left to right)

Approach-1
Lets assume we have 4 strings code, coder, coding, codehelp. Pick one string.

code

	c	o	d	e
coder	✓	✓	✓	✓
coding	✓	✓	✓	x
codehelp	✓	✓	✓	✓

Hence cod is the longest common prefix. We can do this approach via 2 loops. Outer for loop will run on code and inner for loop will run for n-1 strings

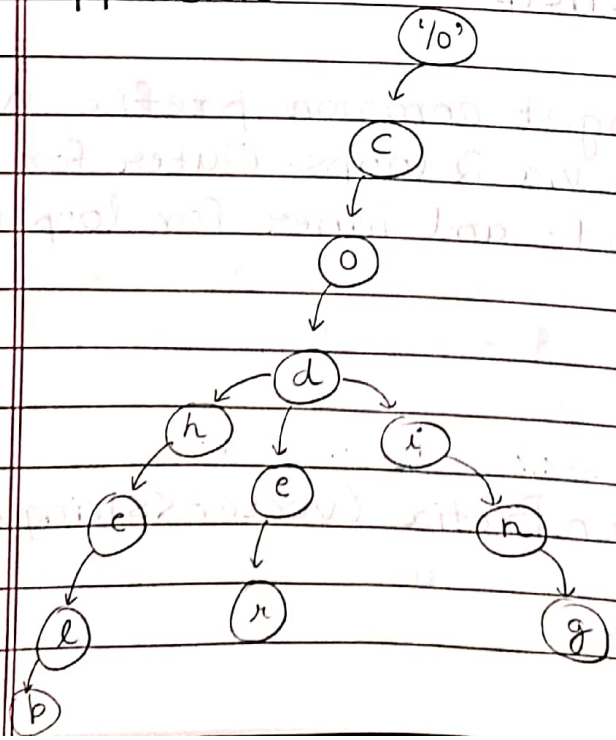
Code

```
String longestCommonPrefix (vector<string>&strs)
{
    string ans = "";
```

```
// Loop for 1st string
for (int i = 0 ; i< strs[0].length() ; i++) {
        char ch = strs[0][i];
bool match = true;
//compare ch with all string's jth index
for (int j = 1 ; j< strs.size() ; j++) {
    // invalid                          // ↑ not equal
    if ( strs[j].size() < i || ch != strs[j][i]){
            match = false;
            // Hence exit as prefix not common
            break;
    }
    // Simply break as prefix match not true
    if (match == false)
            break;
    else // Insert character in ans
        ans.push_back(ch);
}
 return ans; // longest common prefix
}
```

Approach-2

code
coder
coding
codehelp



1) Insert all characters of strings in trie.
2) Whenever we find that there are multiple child, then stop but if we find that a node is terminal, then stop.

Code

```
void findLCP (string first, string &ans,
TrieNode * root){
        // Here i can do mistake (handle empty string)
        if (root → isTerminal)
                return ;
        for (int i=0 ; i<first.length(); i++) {
                char ch = first [i];
                if (root → childCount == 1) {
                        ans.push_back (ch);
                        // Move to next child
                        int index = ch - 'a';
                        root = root → children [index];
                }
                else { // More than one children
                        break;
                } // Terminal found before multiple child
                if (root → isTerminal)
                        break;
        }
}

string longestcommonprefix (vector <string> &strs){
        TrieNode * root = new TrieNode ('-');
        // Insert characters in trie
        for (int i=0; i<strs.size(); i++) {
                insertWord (root, strs [i]);
        }
```

```
string ans = " ";
// Move forward by using characters of 1st string
string first = strs[0];
findLCP(first, ans, root);
return ans;
}
```

Q2 Suggestions when string is written in search bar.

i/p → codef

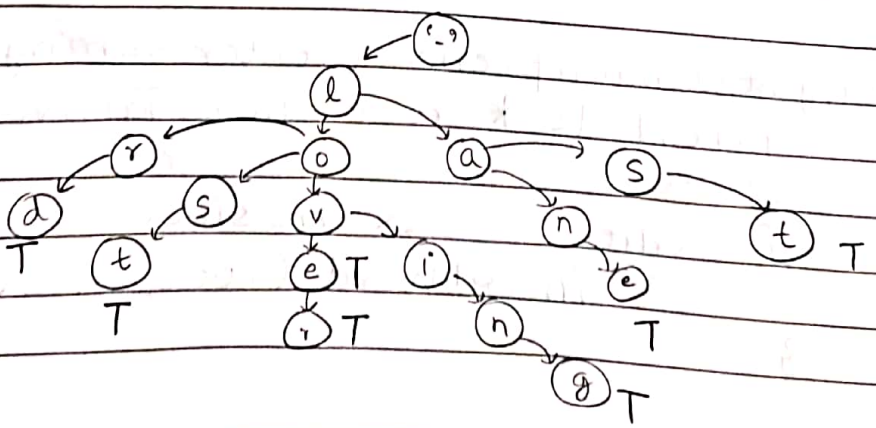
o/p →

c ⇒ 7 strings
co ⇒ 7 strings
cod ⇒ 7 strings
code ⇒ 5 strings
codef ⇒ 1 string

Here we are storing the vector in which some strings have been mentioned

Dry run

vector → [love, loving, lover, lane, last, lost, lord]



i/p ⇒ lovi

	last character
l → 7 strings	l
lo → 5 strings	o
lov → 3 strings	v
lovi → 1 string	i

On the basis of last character, recursively print all the possible strings

Code (Refer to video once)

```
void storeSuggestion(TrieNode * curr,
vector<string> & temp, string & prefix) {
    //Store answer when we arrive on terminal
    if(curr → isTerminal) {
        temp.push_back(prefix);
    }

    for(char ch = 'a'; ch <= 'z'; ch++) {
        int index = ch - 'a';
        TrieNode * next = curr → children[index];
        if(next != NULL) {
            //children exist
            prefix.push_back(ch);
            storeSuggestion(next, temp, prefix);
            //Backtracking
            prefix.pop_back();
        }
    }
}
vector<vector<string>> getSuggestions(TrieNode
* root, string input) {
```

```
TrieNode* prev = root;
vector<vector<string>> output;
string prefix = "";
for(int i=0; i<input.length(); i++){
    char lastCh =                input[i];
    int index = lastCh - 'a';
    TrieNode* curr = prev->children[
    index];

    if(curr==NULL)
        break;
    else{
        vector<string> temp;
        prefix.push_back(        lastCh);
        storeSuggestion(curr, temp, prefix);
        output.push_back(temp);
        prev = curr;
    }
}
return output;
}
```

Time complexity = $O(nm^2)$

n → no. of strings

m → average length of string.