Help Pradyumna!

Problem Description:

In the given problem you are given with N words as input to dictionary, and then Q queries for which you have to print the auto-complete suggestions for each query in lexicographic order.

● How to approach:

In the given problem we can use Trie data structure to make a dictionary, we can insert each of the N words in our Trie, and then we can proceed with the auto-complete using this Trie.

o Inserting in a Trie:

Every character is inserted as an individual Trie node. The children is an array of pointers (or references) to next level trie nodes. The character of each word acts as an index into the array children. If the input character is new or an extension of the existing key, we need to construct non-existing nodes of the character, and mark end of the word for the last node by using value. If value contains a non-zero value that means a word ends here. If the input character is a prefix of the existing character in Trie, we simply mark the last node of the key as the end of a word. Also, we maintain a count for the number of children of a particular node.

We maintain a count for total number of words in the trie while inserting a word.

```
struct trie_node
{
    int value;//contain a non-zero value if any word ends
    int child_cnt;
    trie_node *children[26];
    trie_node()
    {
        value=0;
        child_cnt=0;
        int i;
        for(i=0;i<26;i++)
            children[i]=NULL;
    }
};
struct trie
{
    int cnt;//to keep a count of number of words in the tr
    trie_node *root;//stores the root of the tree
};
trie *T;
void initialize()
{
    int i;
    T=new trie;
    T->cnt=0;
    T->root=new trie_node;
}
trie_node *create_node()
{
    trie_node *newnode=new trie_node;
    return newnode;
}
void insert_trie(string &S)
{
    int length=S.size(), i, idx;
    (T->cnt)++;
    trie_node *node=T->root;
    for(i=0;i<length;i++)
    {
        idx=S[i]-'a';
        if(node->children[idx]==NULL)
        {
            (node->child_cnt)++;
            node->children[idx]=create_node();
        }
        node=node->children[idx];
    }
    node->value=T->cnt;
}
```

- **Auto-Complete:**

Now the user gives a prefix of each search query, we need to give him all recommendations to auto-complete his query based on the strings stored in the Trie.

Given a query prefix, we search for all words having this query.

1. Search for given query using standard Trie search algorithm.
2. If query prefix itself is not present, return false to indicate the same and print no suggestions for the query.
3. If query is present and is end of word in Trie, print query.

4. If last matching node of query has no children, return.
5. Else recursively print all nodes under subtree of last matching node.

```
void solve(trie_node *node, string &A, int k)
{
    int n=node->child_cnt, i;
    if(node->value!=0)
    {
        for(i=0;i<k;i++)
            cout<<A[i];
        printf("\n");
    }
    for(i=0;i<26;i++)
    {
        if(node->children[i]!=NULL)
        {
            A[k]=(char)(i+97);
            solve(node->children[i], A, k+1);
        }
    }
}
bool search_trie(string &S)
{
    int length=S.size(), i, idx;
    trie_node *node=T->root;
    for(i=0;i<length;i++)
    {
        idx=S[i]-'a';
        if(node->children[idx]==NULL)
            return false;
        node=node->children[idx];
    }
    solve(node, S, length);
    return true;
}

int main()
{
    bool b;
    int n, k, i;
    string S;
    cin>>n;
    initialize();
    for(i=0;i<n;i++)
    {
        cin>>S;
        insert_trie(S);
    }
    cin>>k;
    for(i=1;i<=k;i++)
    {
        cin>>S;
        b=search_trie(S);
        if(b==0)
        {
            printf("No suggestions\n");
            insert_trie(S);
        }
    }
}
```