Monday, April 14, 2025 10:40 AM

Alien Dictionary

Alien Dictionary

Difficulty: **Hard** Accuracy: **47.81%** Submissions: **155K+** Points: **8**

A new alien language uses the English alphabet, but the order of letters is unknown. You are given a list of **words[]** from the alien language's dictionary, where the words are claimed to be **sorted lexicographically** according to the language's rules.

Your task is to determine **the correct order of letters** in this alien language based on the given words. If the order is valid, return a string containing the unique letters in lexicographically increasing order as per the new language's rules. If there are multiple valid orders, return any one of them.

However, if the given arrangement of words is inconsistent with any possible letter ordering, return an empty string **("")**.

> A string **a** is lexicographically smaller than a string **b** if, at the first position where they differ, the character in **a** appears earlier in the alien language than the corresponding character in **b**. If all characters in the shorter word match the beginning of the longer word, the shorter word is considered smaller.

Note: Your implementation will be tested using a driver code. It will print **true** if your returned order correctly follows the alien language's lexicographic rules; otherwise, it will print **false**.

Examples:

Input: words[] = ["baa", "abcd", "abca", "cab", "cad"]
Output: true
Explanation: A possible corrct order of letters in the alien dictionary is "bdac".
The pair "baa" and "abcd" suggests 'b' appears before 'a' in the alien dictionary.
The pair "abcd" and "abca" suggests 'd' appears before 'a' in the alien dictionary.
The pair "abca" and "cab" suggests 'a' appears before 'c' in the alien dictionary.
The pair "cab" and "cad" suggests 'b' appears before 'd' in the alien dictionary.
So, 'b' → 'd' → 'a' → 'c' is a valid ordering.

Input: words[] = ["caa", "aaa", "aab"]
Output: true
Explanation: A possible corrct order of letters in the alien dictionary is "cab".
The pair "caa" and "aaa" suggests 'c' appears before 'a'.
The pair "aaa" and "aab" suggests 'a' appear before 'b' in the alien dictionary.
So, 'c' → 'a' → 'b' is a valid ordering.

Input: words[] = ["ab", "cd", "ef", "ad"]
Output: ""
Explanation: No valid ordering of letters is possible.
The pair "ab" and "ef" suggests "a" appears before "e".
The pair "ef" and "ad" suggests "e" appears before "a", which contradicts the ordering rules.

Constraints:
$1 \leq$ words.length ≤ 500
$1 \leq$ words[i].length ≤ 100
words[i] consists only of lowercase English letters.

[Approach 1) Using Kahn's Algorithm:-

Kahn's Algorithm is ideal for this problem because we are required to determine the order of characters in an alien language based on a sorted dictionary. This naturally forms a Directed Acyclic Graph (DAG) where an edge $u \rightarrow v$ means character u comes before character v.

before character V.

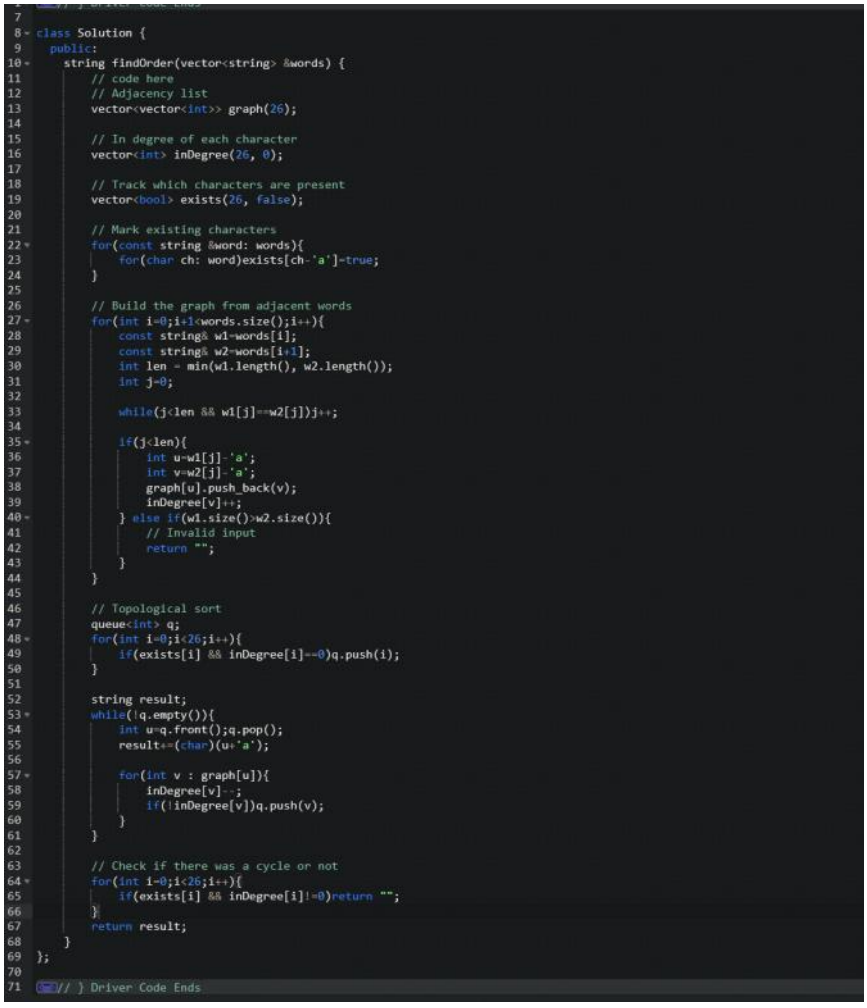
Kahn's Algorithm is a BFS-based topological sort, which efficiently determines a valid linear order of nodes (characters) in a DAG. It's particularly useful here because it handles dependencies using in-degrees, making it simple to identify characters with no prerequisites & process them in correct order. It also easily detects cycles, if a valid topological sort is not possible (eg: cyclic dependency), it returns early.

Step-by-step implementation:-

- Initialize an adjacency list graph[26], in-degree array inDegree[26], & existence tracker exists[26].
- Mark characters that appear in the input words using the exists array.
- Compare each adjacent pair of words to find the first differing character.
- Add a directed edge from the first differing character of the first word to the second word's & increment in-degree of the latter.
- Check for invalid prefix cases where a longer word appears before its prefix (eg: "abc" before

"ab").

- Push all characters with in-degree 0 into a queue as starting nodes for BFS.
- Pop from the queue, add to result & reduce in-degree of neighbors.
- Enqueue neighbors whose in-degree becomes 0 after reduction.
- Detect cycles by checking if all existing characters were processed (length of result).
- Return the result string if no cycle is detected; otherwise return an empty string.



```
class Solution {
  public:
    string findOrder(vector<string> &words) {
        // code here
        // Adjacency list
        vector<vector<int>> graph(26);

        // In degree of each character
        vector<int> inDegree(26, 0);

        // Track which characters are present
        vector<bool> exists(26, false);

        // Mark existing characters
        for(const string &word: words){
            for(char ch: word)exists[ch-'a']=true;
        }

        // Build the graph from adjacent words
        for(int i=0;i+1<words.size();i++){
            const string& w1=words[i];
            const string& w2=words[i+1];
            int len = min(w1.length(), w2.length());
            int j=0;

            while(j<len && w1[j]==w2[j])j++;

            if(j<len){
                int u=w1[j]-'a';
                int v=w2[j]-'a';
                graph[u].push_back(v);
                inDegree[v]++;
            } else if(w1.size()>w2.size()){
                // Invalid input
                return "";
            }
        }

        // Topological sort
        queue<int> q;
        for(int i=0;i<26;i++){
            if(exists[i] && inDegree[i]==0)q.push(i);
        }

        string result;
        while(!q.empty()){
            int u=q.front();q.pop();
            result+=(char)(u+'a');

            for(int v : graph[u]){
                inDegree[v]--;
                if(!inDegree[v])q.push(v);
            }
        }

        // Check if there was a cycle or not
        for(int i=0;i<26;i++){
            if(exists[i] && inDegree[i]!=0)return "";
        }
        return result;
    }
};

// } Driver Code Ends
```

T.C. $O(n*m)$ where n is size of array arr[], m is the size of each string arr[i].

S.C. $O(1)$

[Approach 2]: Using Depth First search

```cpp
class Solution {
  public:
    // Depth-first search function for topological sorting and cycle detection
    bool dfs(int u, vector<vector<int>> &graph, vector<int> &vis, vector<int> &rec, string &ans){
        // Mark the node as visited and part of the current recursion stack
        vis[u]=rec[u]=1;

        for(int v=0;v<26;v++){
            if(graph[u][v]){
                if(!vis[v]){
                    // Recurse and check for cycle
                    if(!dfs(v, graph, vis, rec, ans))return false;
                }else if(rec[v])return false;
            }
        }
        // Add the character to the result after visiting all dependencies
        ans.push_back((char)('a'+u));
        // Remove from recursion stack
        rec[u]=0;
        return true;
    };


    string findOrder(vector<string> &words) {
        // code here

        // Adjacency matrix for character precedence
        vector<vector<int>> graph(26, vector<int>(26, 0));
        // Marks if a character exists in the dictionary
        vector<int> exist(26, 0);
        // Marks if a character has been visited
        vector<int> vis(26, 0);
        // Recursion stack to detect cycles
        vector<int> rec(26, 0);
        // Resultant character order
        string ans="";

        // Step 1: Mark all charcters that appear in the input
        for(string word: words){
            for(char ch: word){
                exist[ch-'a']=1;
            }
        }

        // Build the graph
        for(int i=0;i+1<words.size();i++){
            const string &a = words[i], &b=words[i+1];
            int n=a.size(), m=b.size(), ind=0;

            // Find the first different character between a and b
            while(ind<n && ind<m && a[ind]==b[ind])ind++;
            if(ind!=n && ind==m)return "";

            if(ind<n && ind<m){
                graph[a[ind]-'a'][b[ind]-'a']=1;
            }

        }
        for(int i=0;i<26;i++){
            if(exist[i] && !vis[i]){
                if(!dfs(i, graph, vis, rec, ans)){
                    // Return empty string if a cycle is found
                    return "";
                }
            }
        }
        // Reverse to get the correct topological order
        reverse(ans.begin(), ans.end());
        return ans;
    }
};

// } Driver Code Ends
```

T.C. $O(n*m)$

S.C. $O(1)$