1948. Delete Duplicate Folders in System
Problem Description
In this problem, we have a file system with many duplicate folders due to a bug. We are given a 2D array paths, where paths[i] is an array representing an absolute path to the ith folder in the file system.  Two folders (not necessarily on the same level) are identical if they contain the same non-empty set of identical subfolders and
Two folders (not necessarily on the same level) are identical if they contain the same non-empty set of identical subfolders and underlying subfolder structure. The folders do not need to be at the root level to be identical. If two or more folders are identical, then we need to mark the folders as well as all their subfolders.
The file system will delete all the marked folders and their subfolders once and then return the 2D array ans containing the remaining paths after the deletion. The paths can be returned in any order.
Example:  Let's walk through an example to better understand the problem.  Given the input paths:
[
["c", "x", "v"], ["a", "x", "y"] ]
The file system looks like:
- a - x - y - b - c - x - y
As we can see, folders "a" and "c" have the same structure and same subfolders. Hence, we need to mark and delete them along
with their subfolders.  After deleting, the remaining file system looks like:
- b
So, the output ans should be: [["b"]].  Solution Explanation
To solve this problem, we can utilize a Trie data structure to represent the folder structure. Each Trie node will have a children map and a deleted flag. The key in the children map will be the folder name (a string) and the value will be the child Trie node
map and a deleted flag. The key in the children map will be the folder name (a string) and the value will be the child Trie node representing the subfolder.  We can follow these steps to remove the duplicate folders:
<ol> <li>Create and populate the Trie based on the given input paths.</li> <li>Traverse the Trie recursively and build a unique representation of the subtree rooted at each Trie node. We can use the subtree string representation to maintain a map pointing to the Trie nodes with the same subtree strings.</li> </ol>
<ol> <li>Check the map created in step 2. If any subtree string has multiple Trie nodes, those nodes (and their subfolders) are duplicates, so we mark them as deleted.</li> <li>Starting from the root, traverse the Trie again and construct the remaining paths by ignoring the marked Trie nodes.</li> </ol>
Let's discuss the solution with an example.
paths = [     ["a"],     ["c"],     ["a", "x"],
["a", "x"], ["c", "x"], ["b"], ["c", "x", "y"], ["a", "x", "y"]
First, we populate the Trie based on the paths (step 1):
<ul><li>root</li><li>a</li></ul>
- a - x - b - c
- x - y  The traversal of the Trie to build subtree string representations (step 2) results in:
The traversal of the Trie to build subtree string representations (step 2) results in: $root \implies "((a(x(y)))(b)(c(x(y))))"$
Tool => "(a(x(y)))(b)(c(x(y))))"  a => "(x(y))"  x => "(y)"  y => "()"  b => "()"  c => "(x(y))"
The subtree string to Trie nodes map (subtreeToNodes in the code) will have:
{    "()": [v, b],    "(v)": [x],
"(y)": [x], "(x(y))": [a, c], "((a(x(y)))(b)(c(x(y))))": [root] }
As we can see, the subtree string "(x(y))" has two Trie nodes, so they are duplicates (step 3). Mark the nodes "a" and "c" as deleted:
<pre>- root (not deleted)    - a (deleted)         - x (deleted)         - y (deleted)         - c (deleted)         - x (deleted)         - y (deleted)         - y (deleted)</pre>
Now, traverse the Trie again to construct the remaining folder paths (step 4):
["b"]  Here is the final C++ implementation of the solution provided:
cpp struct TrieNode {
<pre>unordered map<string, shared_ptr<trienode="">&gt; children; bool deleted = false; }; class Solution {</string,></pre>
<pre>public:   vector<vector<string>&gt; deleteDuplicateFolder(vector<vector<string>&gt;&amp; paths) {     vector<vector<string>&gt; ans;     vector<string> path;</string></vector<string></vector<string></vector<string></pre>
<pre>unordered_map<string, vector<shared_ptr<trienode="">&gt;&gt; subtreeToNodes; sort(begin(paths), end(paths));</string,></pre>
<pre>for (const vector<string>&amp; path : paths) {     shared_ptr<trienode> node = root;     for (const string&amp; s : path) {         if (!node-&gt;children.count(s))             node-&gt;children[s] = make_shared<trienode>();</trienode></trienode></string></pre>
<pre>node = node-&gt;children[s]; } </pre>
<pre>buildSubtreeToRoots(root, subtreeToNodes);  for (const auto&amp; [_, nodes] : subtreeToNodes)    if (nodes.size() &gt; 1)    for (shared ptr<trienode> node : nodes)       node-&gt;deleted = true;  constructPath(root, path, ans);</trienode></pre>
<pre>return ans; } private:</pre>
<pre>shared_ptr<trienode> root = make_shared<trienode>();  string buildSubtreeToRoots(     shared_ptr<trienode> node,     unordered map<string, vector<shared_ptr<trienode="">&gt;&gt;&amp; subtreeToNodes) {</string,></trienode></trienode></trienode></pre>
<pre>unordered map<string, vector<shared_ptr<trienode="">&gt;&gt;&amp; subtreeToNodes) {    string subtree = "(";    for (const auto&amp; [s, child] : node-&gt;children)       subtree += s + buildSubtreeToRoots(child, subtreeToNodes);    subtree += ")";    if (subtree != "()")       subtreeToNodes[subtree].push_back(node);    return subtree; }</string,></pre>
<pre> void constructPath(shared ptr<trienode> node, vector<string>&amp; path,</string></trienode></pre>
<pre>for (const auto&amp; [s, child] : node-&gt;children)   if (!child-&gt;deleted) {     path.push_back(s);     constructPath(child, path, ans);     path.pop_back(); }</pre>
<pre> if (!path.empty())     ans.push_back(path); };</pre>
This solution's time complexity will be $O(N * L)$ where N is the number of paths and L is the length of the strings involved. The

space complexity will be O(N \* L) as well due to the Trie data structure.## Python Solution Now, let's implement the same solution in Python: python from collections import defaultdict class TrieNode: \_\_init\_\_(self): def self.children = defaultdict(TrieNode) self.deleted = False class Solution: def deleteDuplicateFolder(self, paths: List[List[str]]) -> List[List[str]]: ans = []path = [] subtree\_to\_nodes = defaultdict(list) paths.sort() root = TrieNode() for p in paths: node = root for s in p: node = node.children[s] self.build\_subtree\_to\_nodes(root, subtree\_to\_nodes) for nodes in subtree to\_nodes.values(): if len(nodes) > 1: for node in nodes: node.deleted = True self.construct\_path(root, path, ans) return ans def build\_subtree\_to\_nodes(self, node: TrieNode, subtree\_to\_nodes: dict) -> str: subtree = "(" for s, child in node.children.items(): subtree += s + self.build\_subtree\_to\_nodes(child, subtree\_to\_nodes) subtree += ")" if subtree != "()": subtree\_to\_nodes[subtree].append(node) return subtree def construct path(self, node: TrieNode, path: List[str], ans: List[List[str]]): for s, child in node.children.items(): if not child.deleted: path.append(s) self.construct\_path(child, path, ans) path.pop() if path: ans.append(path[:]) This Python solution has the same time complexity O(N \* L) and space complexity O(N \* L). **JavaScript Solution** Finally, let's implement the solution in JavaScript: javascript class TrieNode { constructor() { this.children = new Map(); this.deleted = false; class Solution { deleteDuplicateFolder(paths) { const ans = []; const path = []; const subtreeToNodes = new Map(); paths.sort(); const root = new TrieNode(); for (const p of paths) {

let node = root;

for (const s of p) {

if (nodes.length > 1) {

this.constructPath(root, path, ans);

buildSubtreeToNodes(node, subtreeToNodes) {

if (!subtreeToNodes.has(subtree))

return ans;

let subtree = "(";

if (subtree !== "()") {

constructPath(node, path, ans) {

if (!child.deleted) {

path.push(s);

ans.push(Array.from(path));

path.pop();

if (path.length > 0)

subtree += ")";

return subtree;

if (!node.children.has(s))

node = node.children.get(s);

this.buildSubtreeToNodes(root, subtreeToNodes);

for (const nodes of subtreeToNodes.values()) {

for (const [s, child] of node.children.entries()) {

subtreeToNodes.set(subtree, []);

for (const [s, child] of node.children.entries()) {

this.constructPath(child, path, ans);

subtreeToNodes.get(subtree).push(node);

subtree += s + this.buildSubtreeToNodes(child, subtreeToNodes);

for (const node of nodes) {

node.deleted = true;

node.children.set(s, new TrieNode());