1430. Check If a String Is a Valid Sequence from Root to Leaves Path in a **Binary Tree Depth-First Search Breadth-First Search Binary Tree** Medium

Problem Description

these root-to-leaf sequences in the binary tree.

node with the corresponding element in the given sequence arr.

ensure we compare the correct node value with the correct sequence element.

In this problem, we are given a binary tree where each root-to-leaf path represents a sequence of numbers corresponding to the values of the nodes along the path. The task is to determine if a given sequence, represented by an array arr, matches any of

means that each value in the given array arr must match the value of the corresponding node in the tree as we traverse from the root to a leaf, and the sequence should end at a leaf node. A leaf node is defined as a node with no children, implying that it doesn't have a left or right child node. In simple terms, we need

The sequence is considered valid if it corresponds exactly to the sequence of node values from the root node to a leaf node. This

to check if there's a path in the given binary tree such that the concatenation of the node values along the path is exactly the same as the given sequence arr.

Intuition The solution to this problem is based on <u>Depth-First Search</u> (DFS), which is a fundamental traversal algorithm that explores as far

as possible along each branch before backtracking. The idea is to traverse the tree from the root, comparing the value at each

Here's the thinking process for arriving at the DFS solution:

 We start the DFS from the root of the <u>tree</u>. • At any given node, we check if the node's value matches the corresponding element in the sequence. If not, we return False because this path can't possibly match the sequence. • We also keep track of the index u within the sequence arr that we're currently checking. We increment this index as we move down the tree to

- If the current node's value matches the current sequence element and we're at the last element in the sequence (i.e., u == len(arr) 1), we check if the current node is also a leaf (it has no children). If it is a leaf, the sequence is valid and we return True. If it's not a leaf, the sequence is invalid because it hasn't ended at a leaf node.
- both the left and right children, incrementing the index u. • The invocation of DFS on a child node returns True if that subtree contains a path corresponding to the remaining portion of the sequence.

• If the current node's value matches the current sequence element but we're not yet at the end of the sequence, we recursively perform DFS on

- If DFS on both the left and right children returns False, this means that there is no valid continuation of the sequence in this part of the tree, and we return False.
- The initial call to DFS starts with the root node and the first index of the sequence arr.

By applying this approach, we incrementally check each root-to-leaf path against the sequence until we either find a valid path

- that matches the sequence or exhaust all paths and conclude that no valid sequence exists in the binary tree.
- The implementation of the solution utilizes <u>Depth-First Search</u> (DFS), a classic algorithm often used to explore all paths in a <u>tree</u> or graph until a condition is met or all paths have been explored.

because it means we cannot form the desired sequence along this path.

True.

class Solution:

return dfs(root, 0)

Example Walkthrough

Sequence to match: [1, 0, 3]

Solution Implementation

self.val = val

class Solution:

class TreeNode {

class Solution {

};

};

public:

class Solution {

};

TypeScript

TreeNode() {}

self.left = left

self.right = right

def dfs(node, index):

// Definition for a binary tree node.

int value; // Node's value

this.value = value;

this.right = right;

this.left = left;

Definition for a binary tree node.

return False

if index == len(sequence) - 1:

TreeNode left; // Reference to the left child

TreeNode(int value) { this.value = value; }

TreeNode right; // Reference to the right child

TreeNode(int value, TreeNode left, TreeNode right) {

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

bool isValidSequence(TreeNode* root, vector<int>& sequence) {

if (!node || node->val != sequence[index]) return false;

// Define a lambda function for depth-first search

return depthFirstSearch(root, 0);

// Define a depth-first search (DFS) function

if (index === sequence.length - 1) {

return !node.left && !node.right;

return false;

};

if (!node || node.val !== sequence[index]) {

if not node or node.val != sequence[index]:

If we are at the last index, check if it's a leaf node

return node.left is None and node.right is None

return dfs(node.left, index + 1) or dfs(node.right, index + 1)

return False

if index == len(sequence) - 1:

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

// Function to determine if a given sequence is a valid path from root to a leaf in the binary tree

function<bool(TreeNode*, int)> depthFirstSearch = [&](TreeNode* node, int index) -> bool {

// If we are at the last element of the sequence, we should also be at a leaf node

// Continue the search in the left or right child, incrementing the sequence index

if (index == sequence.size() - 1) return !(node->left) && !(node->right);

// Function to determine if a given sequence is a valid path from root to a leaf in the binary tree

// If we are at the last element of the sequence, we should also be at a leaf node

// Continue the search in the left or right child, incrementing the sequence index

return depthFirstSearch(node.left, index + 1) || depthFirstSearch(node.right, index + 1);

// Return false if current node is null or the value does not match the sequence at the current index

function isValidSequence(root: TreeNode | null, sequence: number[]): boolean {

const depthFirstSearch: DepthFirstSearchFunction = (node, index) => {

// Return false if current node is null or value does not match the sequence at current index

return depthFirstSearch(node->left, index + 1) || depthFirstSearch(node->right, index + 1);

// Initial call to depthFirstSearch starting with the root node and the first element of the sequence

def __init__(self, val=0, left=None, right=None):

Tree structure:

•

Solution Approach

The next crucial check determines if we have reached the end of our sequence with u == len(arr) - 1. If the current node is at this index of the sequence array, we must also verify if it is a leaf. The condition root.left is None and root.right is

None confirms whether the current node has no children. If both conditions hold, we have found a valid sequence and return

+ 1) and dfs(root.right, u + 1). Here, we increment the index u by 1 indicating that we're moving to the next element in

Finally, we use an or operator between the calls to dfs on the left and right children because a valid sequence can exist in

either subtree. If either subtree call returns True, it means we have a match, and so we return True for this path.

the leaf of a path) or if the value of root. val does not match the sequence value arr[u]. In either case, we return False

We start by defining a recursive function dfs within the method isValidSequence. This function takes two arguments: root,

Within the dfs function, we first check if the current node root is None (indicating we've reached a non-existent node beyond

representing the current node in the tree, and u, which is the index of the current element in the sequence array arr.

If we are not at the end of the sequence, we continue our DFS on both the left and right subtrees by calling dfs(root.left, u

the sequence as we go one level down the tree.

return root.left is None and root.right is None

return dfs(root.left, u + 1) or dfs(root.right, u + 1)

Below are the details of the DFS implementation:

def isValidSequence(self, root: TreeNode, arr: List[int]) -> bool: def dfs(root, u): if root is None or root.val != arr[u]: return False **if** u == len(arr) - 1:

Here is the implementation of the recursive dfs function encapsulated in the Solution class and its method isValidSequence:

the corresponding index. It systematically explores all viable paths down the tree to determine if a valid sequence matching the given array exists.

To illustrate the solution approach, let's consider a small binary tree example and a given sequence to match:

proceed to call the dfs function recursively on both children with the next index (u + 1), which is 1.

True. This confirms that the sequence [1, 0, 3] is indeed a root-to-leaf path in the given binary tree.

We again proceed with the dfs function recursively on this node's children.

len(arr) - 1) and if the current node is a leaf (no children), which is true.

def isValidSequence(self, root: TreeNode, sequence: List[int]) -> bool:

If we are at the last index, check if it's a leaf node

return node.left is None and node.right is None

private int[] sequence; // Array to hold the sequence to validate against the tree nodes

Function to perform depth-first search on the tree

if not node or node.val != sequence[index]:

This approach effectively performs a depth-first traversal of the tree, comparing each node's value with the sequence value at

• The solution overall begins with the call to dfs(root, 0) where root is the tree's root node and 0 is the starting index of the sequence arr.

We want to determine if this sequence can be formed by traversing from the root to a leaf. Now let's perform a walk through using the dfs function step by step. We start at the root node with the value 1. Since arr[0] is 1, the first element of the sequence matches the root's value. We

First, we consider the left child of the root with the value 0. At this point, arr[1] is 0, which matches the current node's value.

For the left child of the node 0, we have the leaf node with the value 3. The sequence index we're looking for is u + 1 = 2,

which gives us arr[2] = 3, and it matches the leaf node's value. We check if this is the end of the sequence (since u ==

Considering the conditions are satisfied, the sequence [1, 0, 3] is found to be valid, and the dfs function returns True.

Since the dfs function has found a valid path that matches the given sequence, the entire isValidSequence method would return

- **Python** class TreeNode:
- # Otherwise, move to the left and right children and increment the index return dfs(node.left, index + 1) or dfs(node.right, index + 1) # Begin depth-first search from the root node starting at index 0 of the sequence return dfs(root, 0) Java

Ensure that the current node exists and the value matches the sequence at the current index

- // Entry method to start the process of validating sequence in the tree public boolean isValidSequence(TreeNode root, int[] sequence) { this sequence = sequence; // Assign the sequence to the instance variable return isPathExist(root, 0); // Begin depth-first search from the root of the tree // Helper method for depth-first search to validate the sequence private boolean isPathExist(TreeNode node, int index) { // If the current node is null or the value does not match the sequence, return false. if (node == null || node.value != sequence[index]) { return false; // Check if this node is a leaf and it's the last element in the sequence if (index == sequence.length - 1) { return node.left == null && node.right == null; // Move to the next index in the sequence and search in both left and right subtrees return isPathExist(node.left, index + 1) || isPathExist(node.right, index + 1); C++ // Definition for a binary tree node. struct TreeNode { int val; TreeNode *left; TreeNode *right; TreeNode() : val(0), left(nullptr), right(nullptr) {}
- // Definition for a binary tree node using TypeScript interfaces interface TreeNode { val: number; left: TreeNode | null; right: TreeNode | null; // Function to create a new TreeNode with default values function createTreeNode(value: number = 0, left: TreeNode | null = null, right: TreeNode | null = null): TreeNode { return { val: value, left: left, right: right **}**; // Type for the depth-first search function type DepthFirstSearchFunction = (node: TreeNode | null, index: number) => boolean;
- // Initial call to DFS starting with the root node and the first element of the sequence return depthFirstSearch(root, 0); class TreeNode: # Definition for a binary tree node. def __init__(self, val=0, left=None, right=None): self.val = val self.left = left self.right = right class Solution: def isValidSequence(self, root: TreeNode, sequence: List[int]) -> bool: # Function to perform depth-first search on the tree def dfs(node, index): # Ensure that the current node exists and the value matches the sequence at the current index
- Time and Space Complexity **Time Complexity**

Otherwise, move to the left and right children and increment the index

Begin depth-first search from the root node starting at index 0 of the sequence

The time complexity of the code is O(N), where N is the number of nodes in the binary tree. This is because in the worst-case scenario, we will have to visit every node to check if it's a part of the valid sequence. The dfs function is called for every node,

return dfs(root, 0)

but never more than once for a node, so we visit each node a maximum of one time. **Space Complexity**

The space complexity of the code is O(H), where H is the height of the binary tree. This is due to the recursive nature of the depth-first search algorithm, which will use stack space for each recursive call. In the worst case (completely unbalanced tree), this could be O(N), if the tree takes the form of a linked list (essentially, each node only has one child). However, in the best case (completely balanced tree), the height of the tree is log(N), and thus the space complexity would be O(log(N)).