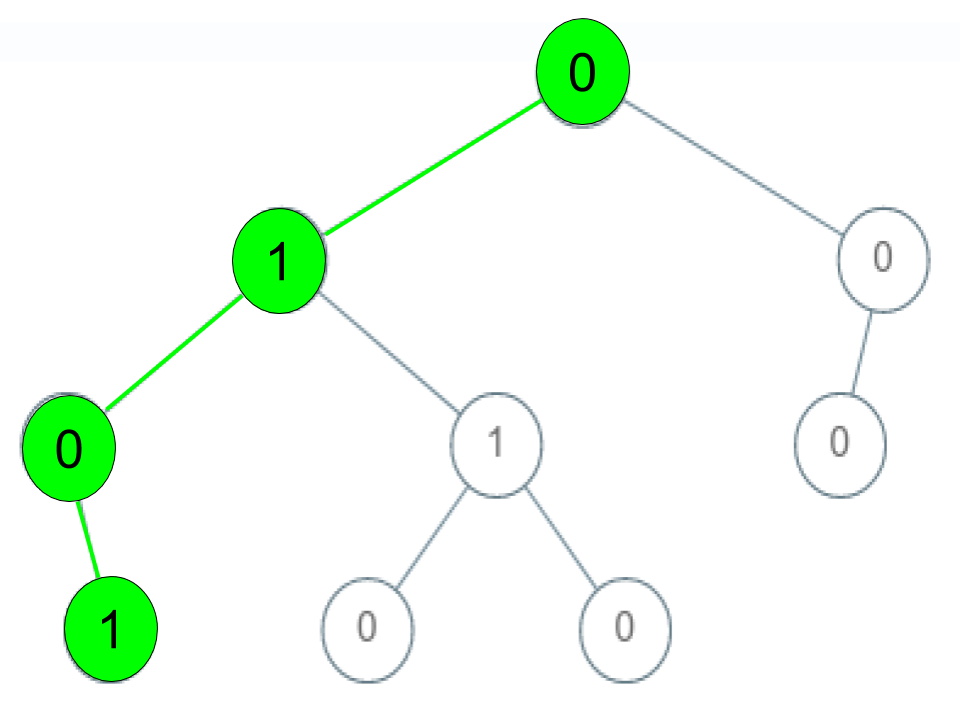
Given a binary tree where each path going from the root to any leaf form a **valid sequence**, check if a given string is a **valid sequence** in such binary tree.

We get the given string from the concatenation of an array of integers **arr** and the concatenation of all values of the nodes along a path results in a **sequence** in the given binary tree.

**Example 1:**

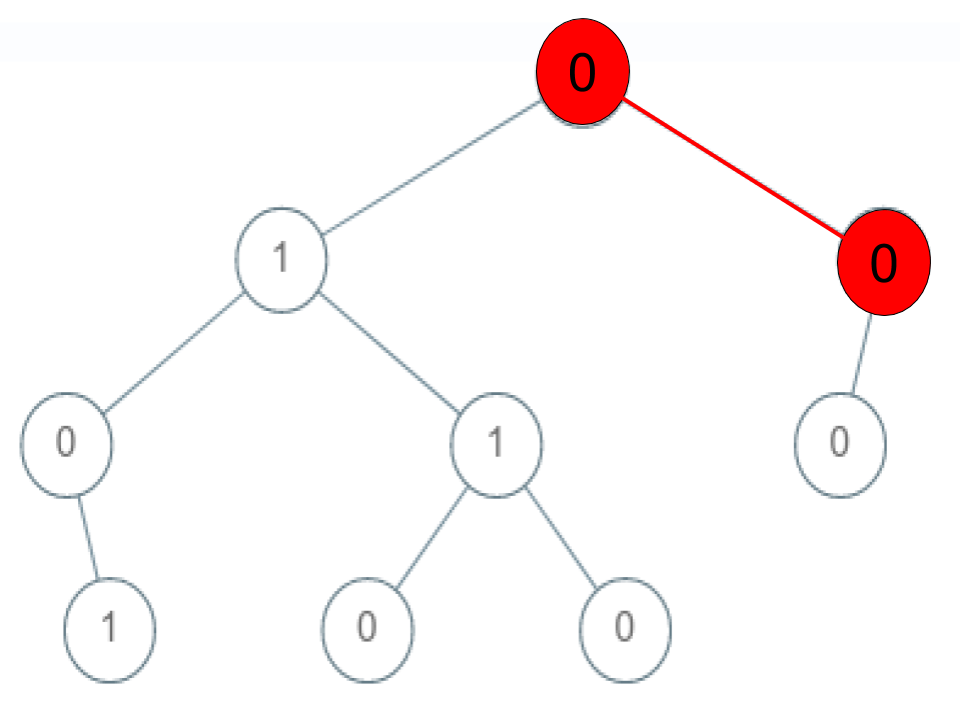


**Input:** root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,0,1]

**Output:** true

**Explanation:**The path 0 -> 1 -> 0 -> 1 is a valid sequence (green color in the figure).Other valid sequences are:0 -> 1 -> 1 -> 0,0 -> 0 -> 0

**Example 2:**

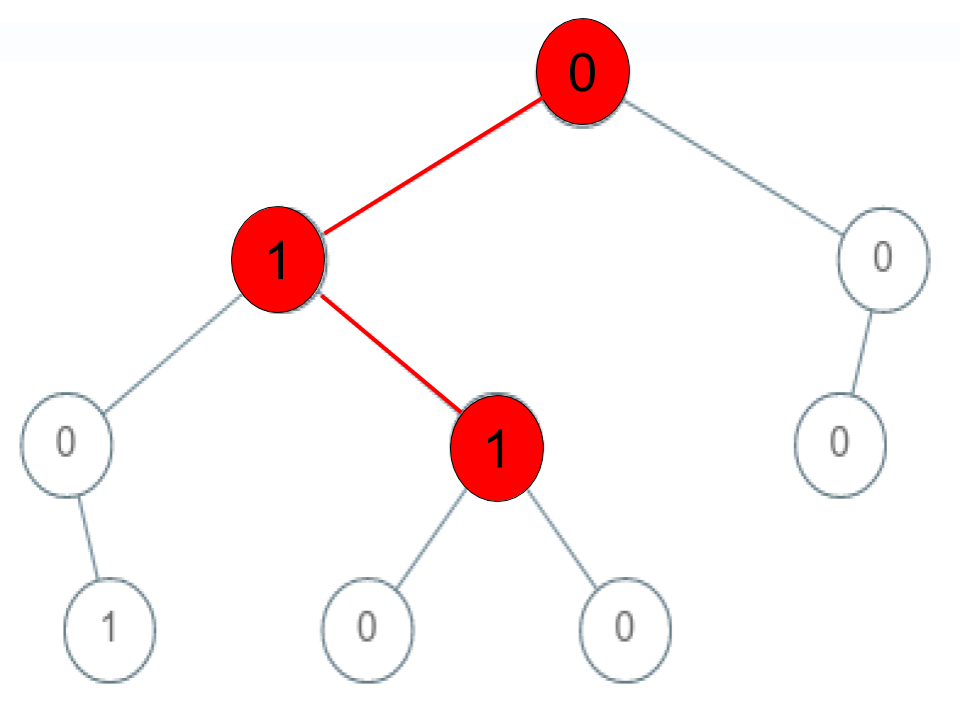


**Input:** root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,0,1]

**Output:** false

**Explanation:** The path 0 -> 0 -> 1 does not exist, therefore it is not even a sequence.

**Example 3:**



**Input:** root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,1]

**Output:** false

**Explanation:** The path 0 -> 1 -> 1 is a sequence, but it is not a valid sequence.

**Constraints:**

1 <= arr.length <= 5000

0 <= arr[i] <= 9

Each node's value is between [0 - 9].

**Attempt 1: 2023-03-10**

**Solution 1: DFS (10 min)**

import java.util.ArrayList;

import java.util.List;

public class TreeSolution {

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

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

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

this.val = val;

this.left = left;

this.right = right;

}

}

public boolean isValidSequence(TreeNode root, int[] arr) {

return helper(root, arr, 0);

}

private boolean helper(TreeNode root, int[] arr, int index) {

// Base condition:

// If the current node is null or the value does not match the sequence, return false.

if(root == null) {

return false;

}

if(root.val != arr[index]) {

return false;

}

// Important: Watch out wrong condition to put in 'if' as 'root.left

// == null && root.right == null', the 'if' condition must be 'index

// == arr.length - 1' rather than 'root.left == null && root.right == null',

// it can test out by input arr = {1,3} and tree as below, it will

// error out as 'ArrayIndexOutOfBoundsException', because arr = {1,3}

// length = 2 and last index = 1, but when DFS traverse tree, if we go

// to path 1->3->6, when arr exhaust, the root still not null and will

// move on one more recursion level, at that moment, root.val != arr[index]

// = arr[2] will out of index, so we need first check if 'index ==

// arr.length - 1' as condition and then check if that's a leaf node by

// 'root.left == null && root.right == null' to guarantee arr not exhaust

/\*\*

\* 1

\* / \

\* 2 3

\* / \ / \

\* 4 5 6 7

\*

\* arr = {1,3} -> error out on index out of boundary for node = 6

\*/

//if(root.left == null && root.right == null) {

// return index == arr.length - 1;

//}

// Check if this node is a leaf and it's the last element in the sequence

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

return root.left == null && root.right == null;

}

// Move to the next index in the sequence and search in both left and right subtrees

return helper(root.left, arr, index + 1) || helper(root.right, arr, index + 1);

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / \ / \

\* 4 5 6 7

\*/

TreeSolution so = new TreeSolution();

TreeNode one = so.new TreeNode(1);

TreeNode two = so.new TreeNode(2);

TreeNode three = so.new TreeNode(3);

TreeNode four = so.new TreeNode(4);

TreeNode five = so.new TreeNode(5);

TreeNode six = so.new TreeNode(6);

TreeNode seven = so.new TreeNode(7);

one.left = two;

one.right = three;

two.left = four;

two.right = five;

three.left = six;

three.right = seven;

int[] arr = new int[]{1,3};

boolean result = so.isValidSequence(one, arr);

System.out.println(result);

}

}

Time Complexity: O(N)

Space Complexity: O(logN)

**Refer to**

<https://www.cnblogs.com/cnoodle/p/12812393.html>

题意是给一个二叉树和一个从根节点开始的sequence，请你判断这个sequence是否是一个有效的，从根节点到某个叶子节点的sequence。我只放一个例子好了，因为这个题比较直观。如果给出的sequence arr少任何一个节点，都要return false。

思路是**DFS深度遍历**。如果遍历到的当前节点没有左孩子也没有右孩子，说明是叶子节点了，则判断此时的深度 + 1是不是跟遍历arr的长度一致。需要检查的corner case有

当前深度大于arr.length

当前节点上的值跟遍历到的arr[index]不一样

当前节点为空 - 事实上在路径中的任何一个节点都不能为空

时间O(n)

空间O(n)

Java实现

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public boolean isValidSequence(TreeNode root, int[] arr) {

return dfs(root, arr, 0);

}

private boolean dfs(TreeNode n, int[] a, int depth) {

if (n == null || depth >= a.length || a[depth] != n.val) { // base cases.

return false;

}// key base case: a leave found.

if (n.left == null && n.right == null) { // credit to @The\_Legend\_ for making the code clean

return depth + 1 == a.length; // valid sequence?

}

return dfs(n.left, a, depth + 1) || dfs(n.right, a, depth + 1); // recurse to the children.

}

}

**Refer to**

<https://algo.monster/liteproblems/1430>

**Problem Description**

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

The sequence is considered valid if it corresponds exactly to the sequence of node values from the root node to a leaf node. This 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 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 Depth-First Search (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 node with the corresponding element in the given sequence arr.

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

We start the DFS from the root of the tree.

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

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.

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 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 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.

**Solution Approach**

The implementation of the solution utilizes Depth-First Search (DFS), a classic algorithm often used to explore all paths in a tree or graph until a condition is met or all paths have been explored.

Below are the details of the DFS implementation:

We start by defining a recursive function dfs within the method isValidSequence. This function takes two arguments: root, representing the current node in the tree, and u, which is the index of the current element in the sequence array arr.

Within the dfs function, we first check if the current node root is None (indicating we've reached a non-existent node beyond 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 because it means we cannot form the desired sequence along this path.

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 True.

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 + 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 the sequence as we go one level down the tree.

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.

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

class Solution:

    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:

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

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

        return dfs(root, 0)

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.

This approach effectively performs a depth-first traversal of the tree, comparing each node's value with the sequence value at the corresponding index. It systematically explores all viable paths down the tree to determine if a valid sequence matching the given array exists.

**Example Walkthrough**

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

Tree structure:

    1

   / \

  0   2

 / \

3   4

Sequence to match: [1, 0, 3]

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

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

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

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

**Java Solution**

// Definition for a binary tree node.

class TreeNode {

    int value; // Node's value

    TreeNode left; // Reference to the left child

    TreeNode right; // Reference to the right child

    TreeNode() {}

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

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

        this.value = value;

        this.left = left;

        this.right = right;

    }

}

class Solution {

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

    // 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);

    }

}

**Time and Space Complexity**

**Time Complexity**

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, 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)).

**Refer to**

[L124.P9.7.Binary Tree Maximum Path Sum](note://8907B88FF6D14ADCB5A4E284B0CB00D7)

[L257.Binary Tree Paths](note://0FCBE93DB9274C62B8DB410E76CC3C7E)

[L298.Binary Tree Longest Consecutive Sequence (Refer L257.Binary Tree Paths](note://AC092AED87E046E5A52ECF94B66C5E4F)

[L549.Binary Tree Longest Consecutive Sequence II (Refer L257.Binary Tree Pa](note://857146268EE34C6EAC3D963EBC1C1199)