# 872. Leaf-Similar Trees

**Problem Description** 

In the given problem, we are asked to compare two binary trees to check if they are 'leaf-similar'. Binary trees are a type of data

**Depth-First Search Binary Tree** 

structure in which each node has at most two children, referred to as the left child and the right child. A leaf node is a node with no children. The 'leaf value sequence' is the list of values of these leaf nodes read from left to right. Two binary trees are considered 'leaf-similar' if the sequences of their leaf values are identical. For example, the leaf value sequence of the tree depicted in the problem is (6, 7, 4, 9, 8) as those are the values of the

nodes that have no children, read from left to right. To solve this problem, we need to collect the leaf sequences from both trees and then compare them to determine if the two trees are 'leaf-similar'.

#### To solve this problem, we can use a <u>Depth-First Search</u> (DFS) on both trees. DFS is an algorithm for traversing or searching <u>tree</u> or graph data structures. In the context of a binary tree, it starts at the root and explores as far as possible along each branch

Intuition

before backtracking. Here's why DFS is suitable for this problem: 1. DFS can be easily implemented using recursion.

3. We can build the leaf sequence during the DFS by adding the node's value to our sequence only when we hit a leaf node (a node with no children).

- The provided solution uses the DFS method dfs to traverse each tree starting from the root and collect the leaves values. The function dfs recursively visits the left and right children of the current node and stores the node's value if and only if the node is

2. While traversing the tree using DFS, we have a clear path to the leaf nodes, which are the focus of this problem.

the solution compares the two lists of leaf values, and if they are equal, it returns true, indicating the two trees are leaf-similar. If not, it returns false.

a leaf (i.e., it has neither a left nor a right child). This collected sequence of values is then returned as a list for each tree. Finally,

Solution Approach The solution is implemented in Python, and it focuses on a <a href="Depth-First Search">Depth-First Search</a> (DFS) approach to traverse through the binary trees. The implementation defines a nested function dfs within the leafSimilar method, with the purpose of performing the actual DFS traversal, searching the entire tree for its leaves, and building the leaf sequence.

and right subtrees.

1. The dfs function is defined, which takes a single argument, root, representing the current node in the tree. 2. Upon each invocation of dfs, the function first checks if the current root node is None. If it is, it returns an empty list as there are no leaves to gather from a None subtree. 3. If the current node is not None, the dfs function first recursively calls itself with root.left and root.right to search through the entire left

4. The leaves are gathered by this part of the code: ans = dfs(root.left) + dfs(root.right). This code concatenates the left and right subtree

leaves into one sequence.

complexity of O(N), where N is the total number of nodes in the tree.

Tree2

leaf value 1. The sequence for Tree1 is now [6, 7, 1].

node of node 6. The sequence for Tree2 is [3, 7, 6].

indicating that Tree1 and Tree2 are not leaf-similar.

3 and stops moving further down this branch.

Here's the step-by-step breakdown of the algorithm:

- 5. Finally, the function checks if the node is a leaf node, that means both root.left and root.right are None. If it is a leaf, it returns a list containing the leaf's value: [root.val]. If the node is not a leaf, it returns the concatenated list of leaf values from both the left and right
- subtrees. 6. The main function leafSimilar calls the dfs function for both root1 and root2 trees, collecting the sequences of leaf values as lists. 7. The solution concludes by comparing these two lists with dfs(root1) == dfs(root2). If they are identical, it means that the leaf value

sequences are the same, and therefore the two trees are considered leaf-similar and True is returned. If the sequences differ in any way, the

conquer nature of the problem. The code is efficient and concise, exploring each node exactly once, which results in a time

- function returns False. This solution uses recursion as a natural way to navigate a tree's structure and gather leaf values, exploiting the divide-and-
- **Example Walkthrough** Let's take two small binary trees as an example to illustrate the solution approach: Consider the following binary trees, Tree1 and Tree2:

## For Tree1, given the binary tree structure, the leaves are the nodes with the values 6, 7, and 1. Therefore, the leaf value

sequence for Tree1 is [6, 7, 1].

**Step-by-Step Walkthrough:** 

Tree1

For Tree1, the dfs function begins at the root node 3, and then recursively searches the left child 5. From node 5, it further

For Tree2, the leaves are the nodes with the values 3, 7, and 6. Thus, the leaf value sequence for Tree2 is [3, 7, 6].

Now we want to determine if Tree1 and Tree2 are leaf-similar according to the problem statement using the DFS approach.

traverses its children 6 (a leaf node) and 2. The function finds leaf 6 and continues to explore 2, down to leaf 7.

Simultaneously, it explores the right child 1 of the root node 3 which is also a leaf node. Here, the dfs function collects the

For Tree2, the dfs function starts at the root node 5, then traverses the left child 3. Node 3 is a leaf so it records the value

After collecting the leaf sequences from both trees using DFS, the solution now compares the leaf value sequences of both

Afterwards, it explores the right side of 5, going down to 6 which is a leaf node. It also records 7 which is the left child leaf

trees: [6, 7, 1] for Tree1 and [3, 7, 6] for Tree2. Since the leaf sequences are not identical (since [6, 7, 1] is not the same as [3, 7, 6]), our function will return False,

def leafSimilar(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> bool:

# If the current node is None, return an empty list

leaves = dfs(root.left) + dfs(root.right)

# Compare the leaf value sequences of both trees

public boolean leafSimilar(TreeNode root1, TreeNode root2) {

// Return the comparison result of leaves

TreeNode() : val(0), left(nullptr), right(nullptr) {}

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

// Method to check if two binary trees are leaf-similar.

// Compare the leaf value sequence of two trees

// Helper method to perform DFS and collect leaf nodes.

vector<int> leaves = getLeafValues(node->left);

if (leaves.empty()) leaves.push\_back(node->val);

vector<int> rightLeaves = getLeafValues(node->right);

vector<int> getLeafValues(TreeNode\* node) {

if (!node) return {};

return leaves;

// Recurse on left subtree.

return getLeafValues(root1) == getLeafValues(root2);

bool leafSimilar(TreeNode\* root1, TreeNode\* root2) {

TreeNode(int x, TreeNode \*1, TreeNode \*r) : val(x), left(l), right(r) {}

// Base case: if current node is null, return an empty vector.

// Recurse on right subtree and append its result to `leaves` vector.

leaves.insert(leaves.end(), rightLeaves.begin(), rightLeaves.end());

// If current node is a leaf node, add its value to `leaves` vector.

// Perform DFS on both trees and store the leaf values

List<Integer> leavesOfTree1 = traverseAndCollectLeaves(root1);

List<Integer> leavesOfTree2 = traverseAndCollectLeaves(root2);

# Helper function to perform a depth-first search (DFS) to find all leaf values in order

# Recursively explore the left and right children, and accumulate leaf values

# Otherwise, it means this is an internal node and 'leaves' contains its subtree's leaves

// Method to check if two given trees have leaves with the same sequence when traversed left-to-right

# If 'leaves' is empty, it means this is a leaf node, so return its value

comparison step at the end of the process is a simple equality check. This example demonstrates the solution approach on a smaller scale, which is directly applicable to larger and more complex tree structures.

Following this approach, we can see that the DFS algorithm efficiently finds the leaf sequences for both trees, and the

def init (self, val=0, left=None, right=None): self.val = val self.left = left self.right = right class Solution:

### Java // Class for the Solution containing the method to check if two binary trees are leaf-similar

class Solution {

TreeNode \*left:

class Solution {

**}**;

public:

TreeNode \*right;

def dfs(root):

if root is None:

return []

return leaves or [root.val]

return dfs(root1) == dfs(root2)

Solution Implementation

# Definition for a binary tree node.

**Python** 

TreeNode:

```
return leaves0fTree1.equals(leaves0fTree2);
    // Recursive method that performs DFS and collects leaf node's values
    private List<Integer> traverseAndCollectLeaves(TreeNode node) {
        // Base case: if the current node is null, return an empty list
        if (node == null) {
            return new ArrayList<>();
        // Traverse the left subtree and collect its leaf nodes
        List<Integer> leaves = traverseAndCollectLeaves(node.left);
        // Traverse the right subtree and collect its leaf nodes
        leaves.addAll(traverseAndCollectLeaves(node.right));
        // Check if current node is a leaf node (as it will have no children after above traversals)
        if (leaves.isEmpty()) {
            // If it's a leaf, add its value to the list
            leaves.add(node.val);
        // Return the list of leaf node's values
        return leaves;
C++
#include <vector>
// Definition for a binary tree node.
struct TreeNode {
    int val;
```

```
// Define a basic structure for a TreeNode.
class TreeNode {
   val: number:
```

**TypeScript** 

**}**;

```
left: TreeNode | null;
    right: TreeNode | null;
    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
// Function to check if two binary trees are leaf-similar.
// Returns true if they are leaf-similar, otherwise false.
function leafSimilar(root1: TreeNode | null, root2: TreeNode | null): boolean {
    // Compare the leaf value sequence of two trees
    return getLeafValues(root1).toString() === getLeafValues(root2).toString();
// Helper function to perform DFS and collect leaf nodes.
// Takes a node and returns an array of leaf values.
function getLeafValues(node: TreeNode | null): number[] {
    // Base case: if current node is null, return an empty array.
    if (!node) return [];
    // Recurse on left subtree.
    let leaves: number[] = getLeafValues(node.left);
    // Recurse on the right subtree and append its result to the 'leaves' array.
    let rightLeaves: number[] = getLeafValues(node.right);
    leaves = leaves.concat(rightLeaves);
    // If the current node is a leaf, add its value to the 'leaves' array.
    if (!node.left && !node.right) leaves.push(node.val);
    return leaves;
// Usage Example:
// const tree1 = new TreeNode(1, new TreeNode(2), new TreeNode(3));
// const tree2 = new TreeNode(1, new TreeNode(3), new TreeNode(2));
// console.log(leafSimilar(tree1, tree2)); // Outputs: false
# Definition for a binary tree node.
class TreeNode:
   def init (self, val=0, left=None, right=None):
       self.val = val
       self.left = left
       self.right = right
class Solution:
   def leafSimilar(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> bool:
```

# Helper function to perform a depth-first search (DFS) to find all leaf values in order

# Recursively explore the left and right children, and accumulate leaf values

# Otherwise, it means this is an internal node and 'leaves' contains its subtree's leaves

are "leaf similar," which means they have the same sequence of leaf values when traversed in depth-first order.

# If 'leaves' is empty, it means this is a leaf node, so return its value

# If the current node is None, return an empty list

leaves = dfs(root.left) + dfs(root.right)

# Compare the leaf value sequences of both trees

maximum height of the second tree, if the trees are highly skewed.

(recursive call stack only) or O(N+M) if yes (including the output list).

## The time complexity of the algorithm is determined by the depth-first search (DFS) function, which visits every node in the binary tree exactly once. Therefore, the time complexity is O(N+M), where N is the number of nodes in the first tree and M is the number

**Time Complexity** 

def dfs(root):

if root is None:

return []

return leaves or [root.val]

return dfs(root1) == dfs(root2)

Time and Space Complexity

of nodes in the second tree.

The DFS function is applied to both trees separately, so we traverse every node of both trees once, leading to the combined time complexity.

The provided Python code defines a function leafSimilar that takes the roots of two binary trees and returns True if the trees

**Space Complexity** The space complexity is mainly governed by the call stack of the recursive DFS calls and the space used to store the leaf values.

In the worst case, the depth of the recursion could be 0(H1+H2), where H1 is the maximum height of the first tree and H2 is the

## Additionally, the space required to store the leaf values is equal to the total number of leaves in both trees. However, since the DFS function concatenates lists and returns them, if we consider the output as a part of the space complexity, it would be equal

to the total number of nodes (like a post-order traversal), which in the worst case would be O(N+M), where both trees are full trees, and each node is stored in the list exactly once. If the output space is not considered in the space complexity (which is common in analysis), then we mainly consider the space

taken by the call stack, leading to the space complexity of 0(H1+H2). Thus, the space complexity of the algorithm depends on whether the output space is included, making it either 0(H1+H2) if not