100. Same Tree

Easy Tree Depth-First Search Breadth-First Search Binary Tree

Problem Description

The problem presents us with two binary trees and asks us to determine if they are the same. To be considered the same, every corresponding node in the two trees must have the same value and the same structure. This means that not only should the values of the nodes at the same positions match, but the shape of the trees must be identical as well. There should be no node in one tree that does not have a corresponding node in the same position of the other tree. This comparison should be consistent all the way down to the leaves of both trees.

Intuition

The intuition behind the solution to this problem involves recursion and understanding the nature of binary trees. Given two binary trees, the simplest case to compare them is when both trees are empty; they are obviously the same. The next step is to check the root nodes of both trees. If one is empty and the other is not, or if they have different values, then the trees are not the same.

However, if the root nodes match in value (and neither is `None`), we must then continue the comparison with their children. To do this, we apply the same comparison process to the left child of both trees and the right child of both trees.

This process is a typical example of a Depth-First Search (DFS) because it explores as far as possible along each branch before backtracking. In the provided solution, recursive function calls are made to compare the left children of `p` and `q`, and the right children of `p` and `q`, ensuring that each subtree is identical in both structure and value. This recursion continues until all nodes have been visited, or a mismatch is found.

This approach works efficiently since it has the ability to terminate early as soon as a mismatch is detected, without needing to check the rest of the tree.

Solution Approach

The solution to this problem uses a recursive Depth-First Search (DFS) pattern to explore both trees simultaneously. The DFS algorithm repeatedly explores a branch as deeply as possible before backtracking. This approach is implemented by recursively checking each node pair of both trees.

Let's walk through the implementation in the provided code:

- The base case checks if `p` and `q` are both `None`. If both are `None`, it means we've reached the end of both branches, the trees are identical so far, and we return `True`.
- The next base case checks if one of the trees is `None` and the other is not, or if the current nodes' values do not match (`p.val != q.val`). Since the trees need to be structurally identical and have the same values, in these cases we return `False`.
- If none of the base cases terminate the function, the code proceeds to check both the left and right subtrees by calling `isSameTree` recursively for `p.left` and `q.left`, and then for `p.right` and `q.right`.
- The return value is the result of joining these two boolean checks with an `and` operator. The `and` operator ensures that both subtrees (left and right) need to be identical for the function to return `True`.

This recursion will "unravel" when it reaches the bottom of the trees (the leaves), at each stage combining the results from left and right comparisons until it reaches the initial call at the root. If all nodes are matched perfectly throughout the recursion, then the final result would be `True`, meaning both binary trees are the same.

The DFS pattern requires no additional data structures as it operates directly on the tree nodes and utilizes the system's call stack for its recursion.

Overall, the algorithm runs in `O(n)` time complexity where `n` is the number of nodes in the trees, as every node is visited once. The space complexity can be up to `O(n)` as well, particularly in the case of a completely unbalanced tree where a call stack proportionate to the number of nodes would be needed.

Example Walkthrough

Let's consider two small binary trees `A` and `B` as an example to illustrate the solution approach.

Tree `A`:

```
 1
/ \
2   3
```

Tree `B`:

```
 1
/ \
2   3
```

Both trees appear to be structurally identical and also have the same node values corresponding to each position, but we need to use our algorithm to confirm this.

We start our recursive function `isSameTree` and check the root nodes of both trees `A` and `B`. The roots of both trees have the value `1`. Since neither is `None` and both values match, we proceed with the recursive checks on the children.

Next, we compare the left children of the two trees (`A.left` and `B.left`) which are `2` and `2`. We call `isSameTree(A.left, B.left)`:

- Since both values match and there are no more children to these nodes, the recursive call for these left children will eventually return `True`.

Next, we move on to compare the right children of the two trees (`A.right` and `B.right`) which are `3` and `3`. We call `isSameTree(A.right, B.right)`:

- Again, since both values match and there are no further children, the recursive call for these right children will also return `True`.

Since both recursive calls resulted in `True`, we continue by combining these two results with an `and` operator:

`isSameTree(A.left, B.left) and isSameTree(A.right, B.right)` which is `True and True`, resulting in `True`.

Finally, as we have no more nodes left to compare and all matched perfectly, the unraveled recursive calls return back to the initial call invoking the root nodes, confirming that Tree `A` and Tree `B` are indeed the same. The `isSameTree` function would, therefore, return `True` for these two trees.

Solution Implementation

Python

```python
class TreeNode:
    # TreeNode class definition
    def __init__(self, val=0, left=None, right=None):
        self.val = val   # value of the node
        self.left = left  # left child
        self.right = right  # right child

class Solution:
    def isSameTree(self, tree1: Optional[TreeNode], tree2: Optional[TreeNode]) -> bool:
        """
        Checks if two binary trees are the same.

        Args:
        tree1 (Optional[TreeNode]): The root node of the first binary tree.
        tree2 (Optional[TreeNode]): The root node of the second binary tree.

        Returns:
        bool: True if both trees are identical, False otherwise.
        """

        # If both trees are None, they are the same (both empty)
        if tree1 == tree2:
            return True

        # If one of the trees is None or the values of current nodes are different,
        # then the trees are not the same.
        if tree1 is None or tree2 is None or tree1.val != tree2.val:
            return False

        # Otherwise, check recursively for the left and right subtrees.
        # Both subtrees must be the same for the entire trees to be considered the same.
        return self.isSameTree(tree1.left, tree2.left) and \
               self.isSameTree(tree1.right, tree2.right)
```

Java

```java

// Definition for a binary tree node.
class TreeNode {
    int val;    // value of the node
    TreeNode left;  // left child
    TreeNode right; // right child

    // Constructor to create a new node with no children
    TreeNode() {}

    // Constructor to create a new node with a specified value
    TreeNode(int val) {
        this.val = val;
    }

    // Constructor to create a new node with a specified value and specified left and right children
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        // If both trees are empty, they are the same.
        if (p == null && q == null) return true;

        // If one of the trees is empty or the values of current nodes don't match,
        // the trees aren't the same.
        if (p == null || q == null || p.val != q.val) return false;

        // Recursively check if the left subtree of both trees are the same
        // AND the right subtree of both trees are the same.
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

C++

```cpp

// Definition for a binary tree node.
struct TreeNode {
    int val;   // Value of the node
    TreeNode *left;  // Pointer to the left child node
    TreeNode *right;  // Pointer to the right child node
    // Constructor to initialize the node with a value. By default, left and right pointers are null.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor to create a node with a value, left child, and right child.
    TreeNode(int x, TreeNode *leftChild, TreeNode *rightChild) : val(x), left(leftChild), right(rightChild) {}
};

class Solution {
public:
    // Function to check if two binary trees are the same.
    bool isSameTree(TreeNode* p, TreeNode* q) {
        // If both nodes are the same instance or both are null, trees are the same.
        if (p == q) return true;
        // If one of the nodes is null or the values don't match, trees are not the same.
        if (!p || !q || p->val != q->val) return false;

        // Check recursively if left subtrees are the same and right subtrees are the same.
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

TypeScript

```typescript

// Definition for a binary tree node.
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

/**
 * Determines if two binary trees are the same.
 * Two binary trees are considered the same if they are structurally identical,
 * and the nodes have the same values.
 *
 * @param {TreeNode | null} tree1 - The root node of the first binary tree.
 * @param {TreeNode | null} tree2 - The root node of the second binary tree.
 * @return {boolean} - True if both trees are the same, false otherwise.
 */
function isSameTree(tree1: TreeNode | null, tree2: TreeNode | null): boolean {
    // Check if both nodes are null, which implies that we've reached the end of both trees.
    if (tree1 === null && tree2 === null) {
        return true;
    }

    // If one of the nodes is null or if the values of the nodes do not match, the trees aren't the same.
    if (tree1 === null || tree2 === null || tree1.val !== tree2.val) {
        return false;
    }

    // Recursively check the left and right subtrees.
    return isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right);
}


class TreeNode:
    # TreeNode class definition
    def __init__(self, val=0, left=None, right=None):
        self.val = val   # value of the node
        self.left = left  # left child
        self.right = right  # right child

class Solution:
    def isSameTree(self, tree1: Optional[TreeNode], tree2: Optional[TreeNode]) -> bool:
        """
        Checks if two binary trees are the same.

        Args:
        tree1 (Optional[TreeNode]): The root node of the first binary tree.
        tree2 (Optional[TreeNode]): The root node of the second binary tree.

        Returns:
        bool: True if both trees are identical, False otherwise.
        """

        # If both trees are None, they are the same (both empty)
        if tree1 == tree2:
            return True

        # If one of the trees is None or the values of current nodes are different,
        # then the trees are not the same.
        if tree1 is None or tree2 is None or tree1.val != tree2.val:
            return False

        # Otherwise, check recursively for the left and right subtrees.
        # Both subtrees must be the same for the entire trees to be considered the same.
        return self.isSameTree(tree1.left, tree2.left) and \
               self.isSameTree(tree1.right, tree2.right)
```

Time and Space Complexity

The time complexity of the given `isSameTree` function is `O(N)` where `N` is the number of nodes in the smaller of the two trees being compared. This occurs because the function is called recursively on each node of the trees until it either finds a discrepancy or checks all pairs of corresponding nodes.

The space complexity is `O(log(N))` in the best case (balanced trees) due to the recursive stack, which would reach at most the depth of the tree. In the worst case (completely unbalanced trees), the space complexity would be `O(N)` since the recursive calls would occur in a linear fashion, leading to a call stack size equal to the number of nodes in the tree.