572. Subtree of Another Tree

Easy Tree Depth-First Search Binary Tree String Matching Hash Function

Problem Description

In this problem, given two binary trees `root` and `subRoot`, we need to determine if the second tree, `subRoot`, is a subtree of the first tree `root`. In essence, we're checking if `subRoot` exists within `root` with the exact structure and node values. A binary tree's subtree includes any node and all its descendants, and this definition allows a binary tree to be a subtree of itself.

Intuition

To solve this problem, we use a depth-first search (DFS) strategy. The logic revolves around a helper function `dfs` that compares two trees to see if they are identical. This is where the heavy lifting occurs, as we must compare each node of the two trees, verifying they have the same values and the same structure.

There are two core scenarios where the trees are deemed to be the same:

1. Both trees are empty, which means that we've reached the end of both trees simultaneously, so they are identical up to this point.
2. The current nodes of the trees are equal, and the left and right subtrees are also identical by recursion.

For other cases, if one node is `None` and the other isn't, or if the node values differ, we return `False`.

The main `isSubtree` function calls `dfs` with `root` and `subRoot` as parameters. If `dfs` returns `True`, then `subRoot` is indeed a subtree of `root`. If not, the function checks recursively if `subRoot` is a subtree of either the left or the right subtree of `root`. This ensures that we check all possible subtrees within `root` for a match with `subRoot`.

The search continues until:

- We find a subtree that matches `subRoot`.
- We exhaust all nodes in `root` without finding a matching subtree, in which case we conclude `subRoot` is not a subtree of `root`.

This solution is elegant and direct, leveraging recursion to compare subtrees of `root` with `subRoot`.

Solution Approach

The implementation of the provided solution follows the intuition of utilizing depth-first search (DFS). Here's a breakdown of the approach and implementation details:

- A nested function `dfs(root1, root2)` is defined within the `isSubtree` method. The purpose of `dfs` is to compare two nodes from each tree and their respective subtrees for equality.

- `dfs` uses recursion to navigate both trees simultaneously. If at any point the values of root1 and root2 do not match, or one is `None` and the other is not, `False` is returned, indicating the trees are not identical at those nodes.

- In `dfs`, three conditions are evaluated for each pair of nodes:

  1. If both nodes are `None`, the subtrees are considered identical, and it returns `True`.
  2. If only one of the nodes is `None`, it means the structures differ, hence it returns `False`.
  3. If neither of the nodes is `None`, it checks whether the values are identical and proceeds to call `dfs` recursively on the left and right subtrees (`dfs(root1.left, root2.left)` and `dfs(root1.right, root2.right)`). If all these checks pass, `True` is returned.
- In the main function `isSubtree`, which takes `root` and `subRoot` as arguments, we first ensure that `root` is not `None`. If `root` is `None`, there's no tree to search, so the function returns `False`.

- The function recursively calls itself while also invoking `dfs`. It first calls `dfs` with `root` and `subRoot` to check for equality at the current nodes. If this does not yield a match, the function recursively checks the left and right subtrees of `root` with the calls `self.isSubtree(root.left, subRoot)` and `self.isSubtree(root.right, subRoot)` respectively.

- Essentially, the solution method takes advantage of recursive tree traversal. First, it attempts to match the `subRoot` tree with the `root` using `dfs` and if unsuccessful, it moves onto `root`'s left and right children, trying to find a match there, continuing the process until a match is found or all possibilities are exhausted.

The time complexity of the algorithm is O(n * m), with 'n' being the number of nodes in the `root` tree and 'm' being the number of nodes in the `subRoot` tree, considering the worst-case scenario where we need to check each node in `root` against `subRoot`.

The space complexity is O(n) due to the recursion, which could potentially go as deep as the height of the tree `root` in the worst-case scenario.

Example Walkthrough

Let's imagine we have the following two binary trees:

```
Tree `root`:

      3
     / \
    4   5
   / \
  1   2

Tree `subRoot`:

    4
   / \
  1   2
```

We want to determine if `subRoot` is a subtree of `root`. According to the solution described above, we would perform the following steps:

1. We call `isSubtree` with `root` (node with value 3) and `subRoot` (node with value 4).

2. Inside `isSubtree`, we invoke `dfs(root, subRoot)`:

   - It starts with comparing the root nodes of both trees. Since their values (3 and 4) do not match, `dfs` returns `False`.
3. Since `dfs` did not find `subRoot` in the current `root` node, `isSubtree` now calls itself recursively to check if `subRoot` is a subtree of the left subtree of `root`.

   - We now repeat the process for `root.left` (node with value 4) and `subRoot` (node with value 4).
   - We call `dfs(root.left, subRoot)` and check the conditions:
     1. Both nodes are not `None`.

     2. Both nodes have the same value (4).

     3. We recursively call `dfs` on their left and right children:

        - For the left children of `root.left` (node with value 1) and `subRoot` (node with value 1), the values match, and they are both not `None`. The recursion on the left children calls `dfs(root.left.left, subRoot.left)` which would return `True`.

        - For the right children of `root.left` (node with value 2) and `subRoot` (node with value 2), the values also match, and they are both not `None`. The recursion on the right children calls `dfs(root.left.right, subRoot.right)` which would return `True`.

   - As all conditions for the `dfs` function are satisfied and both subcalls of `dfs` returned `True`, the `dfs(root.left, subRoot)` will return `True`.
4. Since `dfs(root.left, subRoot)` returned `True`, `isSubtree` returns `True`. We conclude that `subRoot` is indeed a subtree of `root`, and our search is complete.

Using the defined algorithm, we managed to determine that `subRoot` is a subtree of `root` effectively through recursive depth-first search. Each step of recursion allowed us to compare corresponding nodes and their children, validating that `subRoot` not only shares the same values but also the exact structure and node placement within `root`.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def is_subtree(self, root: TreeNode, sub_root: TreeNode) -> bool:
        # Helper function that checks if two trees are identical
        def is_same_tree(tree1, tree2):
            # If both trees are empty, they are identical
            if tree1 is None and tree2 is None:
                return True
            # If one tree is empty and the other is not, they are not identical
            if tree1 is None or tree2 is None:
                return False
            # Check if the current nodes have the same value
            # and recursively check left and right subtrees
            return (tree1.val == tree2.val and
                    is_same_tree(tree1.left, tree2.left) and
                    is_same_tree(tree1.right, tree2.right))

        # If the main tree is empty, sub_root cannot be a subtree of it
        if root is None:
            return False
        # Check if the current trees are identical, or if the sub_root is a subtree
        # of the left subtree or right subtree of the current root
        return (is_same_tree(root, sub_root) or
                self.is_subtree(root.left, sub_root) or
                self.is_subtree(root.right, sub_root))

# Note: Do not modify the method names such as `is_subtree` as per the instructions.
```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
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 {
    /**
     * Determines if a binary tree has a subtree that matches a given subtree.
     *
     * @param root The root node of the main tree.
     * @param subRoot The root node of the subtree that we are looking for in the main tree.
     * @return boolean indicating if the subtree is present in the main tree.
     */
    public boolean isSubtree(TreeNode root, TreeNode subRoot) {
        // If the main tree is null, subRoot cannot be a subtree
        if (root == null) {
            return false;
        }
        // Check if the tree rooted at this node is identical to the subRoot
        // or if the subRoot is a subtree of the left or right child
        return isIdentical(root, subRoot) || isSubtree(root.left, subRoot)
            || isSubtree(root.right, subRoot);
    }

    /**
     * Helper method to determine if two binary trees are identical.
     *
     * @param root1 The root node of the first tree.
     * @param root2 The root node of the second tree.
     * @return boolean indicating whether the trees are identical or not.
     */
    private boolean isIdentical(TreeNode root1, TreeNode root2) {
        // If both trees are empty, then they are identical
        if (root1 == null && root2 == null) {
            return true;
        }
        // If both are not null, compare their values and check their left & right subtrees
        if (root1 != null && root2 != null) {
            return root1.val == root2.val && isIdentical(root1.left, root2.left)
                && isIdentical(root1.right, root2.right);
        }
        // If one is null and the other isn't, they are not identical
        return false;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Function to check if a given subtree `subRoot` is a subtree of the main tree `root`
    bool isSubtree(TreeNode* root, TreeNode* subRoot) {
        // If the main tree root is null, then `subRoot` cannot be a subtree
        if (!root) return false;

        // Check if trees are identical or if `subRoot` is a subtree of either left or right subtrees
        return isIdentical(root, subRoot) || isSubtree(root->left, subRoot) || isSubtree(root->right, subRoot);
    }

    // Helper function to check if two trees are identical
    bool isIdentical(TreeNode* treeOne, TreeNode* treeTwo) {
        // If both trees are empty, they are identical
        if (!treeOne && !treeTwo) return true;
        // If one of the trees is empty, they are not identical
        if (!treeOne || !treeTwo) return false;

        // Compare the values and recursively check left and right subtrees
        return treeOne->val == treeTwo->val &&
               isIdentical(treeOne->left, treeTwo->left) &&
               isIdentical(treeOne->right, treeTwo->right);
    }
};
```

Typescript Solution

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

/**
 * Performs a deep-first-search to check if the two given trees
 * are identical.
 * @param {TreeNode | null} root The node of the first tree.
 * @param {TreeNode | null} subRoot The node of the second tree (subtree candidate).
 * @return {boolean} True if both trees are identical, otherwise false.
 */
const isIdentical = (root: TreeNode | null, subRoot: TreeNode | null): boolean => {
  // If both nodes are null, they are identical by definition.
  if (!root && !subRoot) {
    return true;
  }

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

  // Check recursively if the left subtree and right subtree are identical.
  return isIdentical(root.left, subRoot.left) && isIdentical(root.right, subRoot.right);
};

/**
 * Checks if one tree is subtree of another tree.
 * @param {TreeNode | null} root The root of the main tree.
 * @param {TreeNode | null} subRoot The root of the subtree.
 * @return {boolean} True if the second tree is a subtree of the first tree, otherwise false.
 */
function isSubtree(root: TreeNode | null, subRoot: TreeNode | null): boolean {
  // If the main tree is null, there can't be a subtree.
  if (!root) {
    return false;
  }

  // If the current subtrees are identical, return true.
  // Otherwise, continue the search in the left or right subtree of the main tree.
  return isIdentical(root, subRoot) || isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(m*n)`, where `m` is the number of nodes in the `root` tree and `n` is the number of nodes in the `subRoot` tree. For each node of the `root`, we perform a depth-first search (DFS) comparison with the `subRoot`, which is `O(n)` for each call. Since the DFS might be called for each node in the `root`, this results in `O(m*n)` in the worst case.

Space Complexity

The space complexity of the given code is `O(max(h1, h2))`, where `h1` is the height of the `root` tree and `h2` is the height of the `subRoot` tree. This is due to the space required for the call stack during the execution of the depth-first search. Since the recursion goes as deep as the height of the tree, the maximum height of the trees dictates the maximum call stack size.