979. Distribute Coins in Binary Tree

Medium | Tree | Depth-First Search | Binary Tree | Leetcode Link

Problem Description

You are provided with the `root` of a binary tree, where each node contains a certain number of coins (`node.val`). The total number of coins in the tree is equal to the total number of nodes (`n`). Your task is to redistribute the coins so that every node has exactly one coin. Redistribution is done by moving coins between adjacent nodes (where adjacency is defined by the parent-child relationship). On each move, you can take one coin from a node and move it to an adjacent node. This problem asks you to find the minimum number of moves required to achieve a state where every node has exactly one coin.

Intuition

The intuition behind the solution is to use a post-order traversal, which visits the children of a node before visiting the node itself. The number of moves for a single node is the number of excess coins it has or needs. When a node has more coins than it needs (more than 1), the excess can be moved to the parent. Conversely, if it needs coins (has less than 1 coin), it requests coins from its parent.

To compute the number of moves, call the function recursively on the left and right children. The number of moves required for the left subtree is the absolute value of extra/deficit coins in the left subtree and similarly for the right subtree. The current balance of coins for the node is the node's value plus the sum of moves from left and right (which can be positive, negative, or zero) minus one coin that is needed by the node itself. If this balance is positive, it means the node has extra coins to pass up to its parent. If it's negative, it needs coins from its parent. The total number of moves is accumulated in a variable, which is the sum of the absolute values of extra/deficit coins from both subtrees.

This approach works because it ensures that at each level of the tree, we move the minimum necessary coins to balance out the children before considering the parent. This ensures we don't make redundant moves.

Solution Approach

The solution uses a depth-first search (DFS) approach to traverse the binary tree and calculate the balance of coins at each node.

Here's a step-by-step walkthrough of the implementation:

1. Define a nested helper function, `dfs`, which will perform the DFS traversal. This function accepts a single argument: the current node being visited (initially the `root` of the tree).
2. The base case of the recursion is to return `0` if the `root` passed to the function is `None`, which means you've reached a leaf node's nonexistent child.
3. Recursively call `dfs` on the left child of the current node and store the result in `left`, which represents the number of excess or deficit coins in the left subtree.
4. Do the same for the right child and store the result in `right`.
5. The main logic of the solution is encapsulated in these two points:
   - The total number of moves required (`ans`) is increased by `abs(left) + abs(right)`. This represents the total number of moves needed to balance the left and right subtrees of the current node.
   - The `dfs` function returns `left + right + root.val - 1`. This return value represents the balance of the current node after accounting for its own coin (`root.val`), and it's meant to be either passed up to the parent (if positive) or requested from the parent (if negative).
6. The `ans` variable is defined in the outer function scope but is modified inside the `dfs` function. This is done using the `nonlocal` keyword, which allows the inner function to modify `ans` that is defined in the non-local (outer) scope.
7. Initialize `ans` to `0` before starting the DFS. This variable will accumulate the total number of moves required to balance the tree.
8. After the DFS completes, `ans` contains the total number of moves, and the function returns this value.

The algorithm efficiently calculates the required number of moves using a post-order traversal (visit children first, then process the current node), which helps to avoid redundant moves. It does not require any additional data structures beyond the function call stack used for recursion.

Example Walkthrough

Let's take a simple binary tree as an example to illustrate the solution approach.

Consider this binary tree with 3 nodes:

```
    3
   / \
  0   0
```

Here `3`, `0`, and `0` are the values at each node, respectively, indicating the number of coins they hold. We need to redistribute these coins such that each node has exactly one coin.

1. Starting at the root, the `dfs` function is called on the root node which has the value `3`.
2. Since the root is not `None`, we continue by calling `dfs` on the left child, which has the value `0`.
3. The left child has no children, so both calls to its children return `0`.
4. Now, the `dfs` function processes the left child with value `0`. Since it needs one coin, the return value of the `dfs` function will be `-1` for this node.
5. Similarly, `dfs` is called on the right child, which also has the value `0`, and the process is the same as the left child. The return value is `-1`.
6. With both children processed, we come back to the root node. We use the return values from the left and right child (`-1` from each) to calculate the number of moves required for the root.
7. The total moves at the root node are `abs(-1) + abs(-1) = 2`. This is because we need to move one coin to each child.
8. The current balance for the root node after these moves is `3 + (-1) + (-1) - 1 = 0`. Since the root now has exactly 1 coin (which is our goal for every node), it needs no further action.
9. The `ans` variable gets updated in each recursive call, and after the entire tree is traversed, it holds the value `2`, which is the minimum number of moves required to redistribute the coins.

Finally, after the DFS traversal finishes, we find that the minimum number of moves required for this example is `2`.

Python Solution

```python
# 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 distributeCoins(self, root: Optional[TreeNode]) -> int:
        # Depth-first search (DFS) function to traverse the tree and redistribute coins
        def dfs(node):
            # If the current node is None, we do not need to redistribute any coins
            if node is None:
                return 0

            # Recursively redistribute coins for the left and right subtrees
            left_balance = dfs(node.left)
            right_balance = dfs(node.right)

            # Use nonlocal to modify the ans variable declared in the parent function's scope
            nonlocal moves_count
            # Increment the total moves by the absolute amount of coins to move from/left child and right child
            moves_count += abs(left_balance) + abs(right_balance)

            # Return the net balance of coins for the current subtree
            # Net balance is the coins to be redistributed, i.e., current node's coin plus left and right balance,
            # and we subtract 1 because the current node should have 1 coin after redistribution
            return left_balance + right_balance + node.val - 1

        # Initialize the moves counter to 0
        moves_count = 0
        # Start DFS from the root to distribute coins and calculate the moves
        dfs(root)
        # Return the total number of moves required to distribute the coins
        return moves_count

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val; // Node's value
    TreeNode left; // Left child
    TreeNode right; // Right child

    // Constructor to create a leaf node.
    TreeNode(int val) {
        this.val = val;
    }

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

class Solution {
    private int totalMoves; // To track the total number of moves needed

    // Distributes coins in the binary tree such that every node has exactly one coin
    public int distributeCoins(TreeNode root) {
        totalMoves = 0; // Initialize the total moves to 0
        postOrderTraversal(root); // Start post-order traversal from the root
        return totalMoves; // After traversal, totalMoves has the answer
    }

    // Helper function to perform post-order traversal of the binary tree
    private int postOrderTraversal(TreeNode node) {
        // Base case: if current node is null, return 0 (no moves needed)
        if (node == null) {
            return 0;
        }

        // Recurse on the left subtree
        int leftMovesRequired = postOrderTraversal(node.left);
        // Recurse on the right subtree
        int rightMovesRequired = postOrderTraversal(node.right);

        // Calculate the total moves needed by adding up the moves required by both subtrees
        totalMoves += Math.abs(leftMovesRequired) + Math.abs(rightMovesRequired);

        // Return the net balance of coins for this node
        // The net balance is the sum of left and right balances plus the coins at the node minus one (for the node itself)
        return leftMovesRequired + rightMovesRequired + node.val - 1;
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructors
    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:
    int distributeCoins(TreeNode* root) {
        int totalMoves = 0; // This will store the total number of moves needed to balance the coins

        // The Depth First Search (DFS) function computes the number of moves required
        // starting from the leaves up to the root of the tree.
        // It returns the excess number of coins that need to be moved from the current node.
        function<int(TreeNode*)> dfs = [&](TreeNode* node) -> int {
            // Base case: if the current node is null, return 0 since there are no coins to move.
            if (!node) {
                return 0;
            }

            // Recursive case: solve for left and right subtrees.
            int leftMoves = dfs(node->left);
            int rightMoves = dfs(node->right);

            // The number of moves made at the current node is the sum of absolute values of each subtree’s excess coins.
            // Because moves from both left and right need to pass through the current node.
            totalMoves += abs(leftMoves) + abs(rightMoves);

            // Return the number of excess coins at this node: positive if there are more coins than nodes,
            // negative if there are fewer. A value of -1 means just the right amount for the node itself.
            return leftMoves + rightMoves + node->val - 1;
        };

        // Call the DFS function starting from the root of the tree.
        dfs(root);

        // Return the total number of moves needed to distribute the coins.
        return totalMoves;
    }
};

```

Typescript Solution

```typescript
// Function to distribute coins in a binary tree to ensure each node has exactly one coin.
// Each move allows for a coin transfer between a parent and a child node (either direction).
// The function returns the minimum number of moves needed to achieve this state.
function distributeCoins(root: TreeNode | null): number {
    // Variable to keep track of the total number of moves needed.
    let totalMoves = 0;

    // Helper function for depth-first search (DFS) to compute the number of moves each subtree needs.
    function dfs(node: TreeNode | null): number {
        // If we've reached a null node, return 0 as there are no coins to move.
        if (!node) {
            return 0;
        }

        // Recursively dfs into the left and right subtrees.
        const leftExcessCoins = dfs(node.left);
        const rightExcessCoins = dfs(node.right);

        // Add the absolute values of excess coins from left and right subtrees to the totalMoves.
        // The absolute value is used because it takes the same number of moves whether the coin needs to be moved in or out.
        totalMoves += Math.abs(leftExcessCoins) + Math.abs(rightExcessCoins);

        // Return the excess number of coins at this node: positive if there are too many coins, negative if not enough.
        // Since each node should end up with 1 coin, we subtract 1 from the sum of current node value and excess coins from both chi
        return leftExcessCoins + rightExcessCoins + node.val - 1;
    }

    // Start the DFS from the root of the tree.
    dfs(root);

    // Return the total number of moves calculated using the DFS helper.
    return totalMoves;
}

```

Time and Space Complexity

The given Python code performs a Depth-First Search (DFS) on a binary tree to distribute coins so that every node has exactly one coin.

Time Complexity:

The time complexity of the DFS function is `O(n)`, where `n` is the number of nodes in the binary tree. This is because each node in the tree is visited exactly once during the DFS traversal.

Space Complexity:

The space complexity of the DFS function is `O(h)`, where `h` is the height of the binary tree. This space is used by the call stack during recursion. In the worst case, if the tree is skewed, the height `h` can be `n`, making the space complexity `O(n)`. However, in a balanced binary tree, the space complexity would be `O(log n)`.