Distribute Coins in a Binary Tree

979. Distribute Coins in Binary Tree

Solved

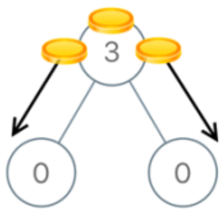
Medium | Topics | Companies

You are given the `root` of a binary tree with `n` nodes where each `node` in the tree has `node.val` coins. There are `n` coins in total throughout the whole tree.

In one move, we may choose two adjacent nodes and move one coin from one node to another. A move may be from parent to child, or from child to parent.

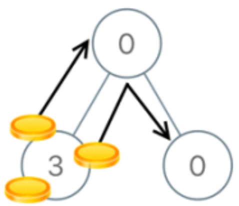
Return *the **minimum** number of moves required to make every node have **exactly** one coin.*

Example 1:




```
Input: root = [3,0,0]
Output: 2
Explanation: From the root of the tree, we move one coin to its left child, and one coin to its right child.
```

Example 2:




```
Input: root = [0,3,0]
Output: 3
Explanation: From the left child of the root, we move two coins to the root [taking two moves]. Then, we move one coin from the root of the tree to the right child.
```

Constraints:

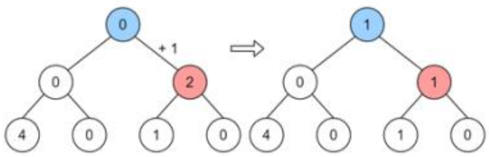
- The number of nodes in the tree is `n`.
- `1 <= n <= 100`
- `0 <= Node.val <= n`
- The sum of all `Node.val` is `n`.

Approach 1: DFS:-

we need to ensure each node contains one coin.

Let's start with an example. How do we obtain a coin for the root node.

I/p: [0, 0, 2, 4, 0, 1, 0]



The right child gives 1 coin to its parent

we could give the blue ~~root~~ node a coin from its red right child. However, this is not an optimal move, as then, a coin from the leftmost node in the tree with 4 coins must be passed to the red node's child that has 0 coins.

From the root, it's hard to determine how to optimally distribute the coins because we don't have enough information about the subtrees.

</> Code

C++ Auto

```cpp
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int distributeCoins(TreeNode* root) {
    moves = 0;
        dfs(root);
        return moves;
    }
    private:
    int moves;
    int dfs(TreeNode* current){
        if(current == nullptr)return 0;

        // Calculate the coins each subtree has available to exchange
        int leftCoins = dfs(current->left);
        int rightCoins = dfs(current->right);

        // Add the total number of exchange to moves
        moves += abs(leftCoins) + abs(rightCoins);

        // The number of coins current has available to exchange
        return (current->val - 1)+leftCoins+rightCoins;
    }
};
```

Complexity Analysis

Let n be the number of nodes in the tree.

- Time complexity: $O(n)$

  Traversing the tree using DFS costs $O(n)$, as we visit each node exactly once and perform $O(1)$ of work at each visit.

- Space complexity: $O(n)$

  The space complexity of DFS, when implemented recursively, is determined by the maximum depth of the call stack, which corresponds to the depth of the tree. In the worst case, if the tree is entirely unbalanced (e.g., a linked list or a left/right skewed tree), the call stack can grow as deep as the number of nodes, resulting in a space complexity of $O(n)$.