2673. Make Costs of Paths Equal in a Binary Tree

Medium Greedy Tree Array Dynamic Programming Binary Tree Leetcode Link

Problem Description

The problem provides us with a *perfect binary tree* with nodes numbered sequentially from 1 to `n`. In this tree, each non-leaf node has exactly two children: the left child is `2 * i` and the right child is `2 * i + 1`, where `i` is the node number of the parent. A cost array of size `n` tells us the cost associated with each node, and the index of the cost array is `0`-indexed, meaning `cost[i]` corresponds to the cost of node `i + 1`.

The objective is to ensure that the *total cost from root to any leaf* is the same for all paths. To achieve this, you can increment the cost of any node by `1` as many times as needed. The task is to find out *the minimum number of increments* required to make all root-to-leaf path costs equal.

Key concepts to understand:

- **Perfect binary tree**: A binary tree where all non-leaf nodes have two children and all leaves are at the same level.
- **Path cost**: The sum of costs of all nodes from the root to a leaf node.
- **Minimum increments**: The least number of single-unit increments to the nodes' costs to equalize the path costs.

Intuition

The intuition behind this problem is to ensure that for any two sibling nodes, the paths from these nodes to their respective leaves have the same cost. If one path has a higher cost to a leaf than the other, we will have to increment the cost of nodes on the cheaper path.

This condition must hold true at every parent node in the tree, as ensuring this locally at every node will result in equal path costs for all leaf nodes.

Here's the thought process for arriving at the solution:

1. Perform a *depth-first search (DFS)* from the root to the leaves. This will allow us to calculate the costs from a node to its leaves recursively.
2. For every node, calculate the costs for the left and right paths independently using DFS.
3. For sibling nodes, find the difference in these costs and add this difference to a global `ans` variable which keeps the running total of all increments needed.
    - The smaller path cost will be increased by this difference to match the larger one.
4. The new cost of each node will be its own cost plus the higher of its children's path costs (after any increments to equalize them).
5. Recursively apply this process until all leaf nodes' path costs have been handled.
6. The `ans` variable will contain the sum of differences after the DFS is completed, which is the answer.

This approach works because it systematically ensures each node's subtree contributes equally to the final path costs, adhering to the perfect binary tree's inherent symmetry, which dictates that the paths from root to leaf are balanced.

Solution Approach

The solution approach uses recursion, which is inherent in a depth-first search (DFS) pattern, to journey through the perfect binary tree. While this may look complex on the surface, it breaks down into a series of simple operations. Let's dissect the code and the related algorithms, data structures, or patterns used:

1. **Depth-First Search (DFS):**
    - The `dfs` function is a recursive function that takes a node `i` as an argument.
    - The base case for recursion happens when the current node `i` does not have children, indicating a leaf node. At this point, it returns the cost of the leaf node (`cost[i - 1]`).
2. **Binary Tree Structure and Node Indices:**
    - In a perfect binary tree, the left child of node `i` is at index `2 * i`, and the right child is at `2 * i + 1`.
    - The solution uses this formula to recursively call `dfs` on child nodes and calculate their path costs up to the leaves.
3. **Cost Calculation and Comparisons:**
    - For a non-leaf node, recursive `dfs` calls are made for both left and right children.
    - The `l` and `r` variables store the maximum path costs from the current node to the leaves of the left and right subtrees, respectively.
    - We ensure both subtrees have equal contribution to the path cost by adding the difference between `l` and `r` to the global `ans` variable. The `max(l, r) - min(l, r)` operation calculates the cost needed to balance the two paths.
4. **Global State Variable:**
    - The `ans` variable accumulates the total increments needed. It's declared as `nonlocal` within `dfs` so that its state is maintained across recursive calls without passing it as an argument.
5. **Returning Computed Costs:**
    - After calculation and potentially equalizing the costs of the paths from the left and right children, `dfs` returns the cost of the current node (`cost[i - 1]`) plus the maximum of the two child path costs (`max(l, r)`). This ensures that the cost of this node contributes to the total path cost accurately.
6. **Initiation and Return Value:**
    - The DFS traversal is initiated by calling `dfs(1)` from the root of the tree, which is node 1.
    - After the DFS completes, the `ans` variable has accumulated all the necessary increments to balance the tree's cost paths, and it is returned as the solution to the problem.

By using DFS in this manner, the solution leverages the recursive nature of trees to perform calculations at each node, ensuring that each node's contribution to the path cost is accounted for precisely in a bottom-up fashion. This results in the entire tree having equal-cost paths from the root to each leaf, with the minimum number of increments needed.

Example Walkthrough

To illustrate the solution approach, let's use a simple perfect binary tree with four nodes, which looks like this:

```
      1(0)
     /   \
  2(1)   3(2)
  / \    / \
4(0) 5(1) 6(0) 7(0)
```

Here, the numbers inside the parentheses denote the initial cost associated with each node. The numbering of the nodes is sequential and represents the level-order traversal of the tree. For example, the root is node 1 with a cost of 0, left child of the root is node 2 with a cost of 1, and so on.

Let's walk through the process following the DFS solution approach:

1. Start the DFS traversal from the root, which is node 1.
2. For node 1, call the `dfs` function recursively on its left and right children (nodes 2 and 3).
3. For node 2, it is not a leaf but has children 4 and 5:
    - Call `dfs` on node 4 (leaf node), which returns its cost, 0.
    - Call `dfs` on node 5 (leaf node), which returns its cost, 1.
    - The left and right child costs (l and r) of node 2 are 0 and 1, respectively.
    - To balance this, we need to make 1 single increment to node 4, so the cost becomes 1 like its sibling node 5.
    - Update `ans` to 1 (number of increments needed so far).
    - The updated cost at node 2 will be its own cost 1 plus the maximum of its children's costs, which is 1, so the total becomes 2.
4. Similarly, for node 3:
    - It also has two children, nodes 6 and 7, both of which are leaf nodes with a cost of 0.
    - After calling `dfs` on nodes 6 and 7, we get the costs 0 for both, and no increments are needed since they are already equal.
    - The updated cost at node 3 will be its own cost 2 plus the maximum of its children's costs, which is 0, so the total becomes 2.
5. Back at the root node 1, we now have the costs from node 2 and node 3, both equal to 2 (after the necessary increment in step 3).
    - Here, no additional increment is needed since the path costs are already equal.
6. By the end of DFS, `ans` equals 1, which means we only needed to perform 1 increment to achieve equal costs from the root to all leaves.

In this example, the minimum number of increments is 1. We only needed to increment the cost of node 4 to equalize the path costs from the root to every leaf.

Python Solution

```python
from typing import List

class Solution:
    def minIncrements(self, n: int, costs: List[int]) -> int:
        # Define a depth-first search function to compute the minimum cost increment.
        def dfs(node_index: int) -> int:
            # Base case: If the current node is a leaf node, return its associated cost.
            if (node_index << 1) > n:
                return costs[node_index - 1]

            # Recursively find the min increment cost of the left and right children.
            left_cost = dfs(node_index << 1)
            right_cost = dfs(node_index << 1 | 1)

            # Update the total increments by adding the difference between the children's costs.
            nonlocal total_increments
            total_increments += abs(left_cost - right_cost)

            # Return the cost of this node plus the maximum cost of its children.
            return costs[node_index - 1] + max(left_cost, right_cost)

        # Initialize the total increments variable.
        total_increments = 0

        # Start the depth-first search from the root node (index 1).
        dfs(1)

        # Return the total minimum increments to balance the binary tree.
        return total_increments

# Example usage:
# solution = Solution()
# min_increments_result = solution.minIncrements(n, cost_list)
# print(min_increments_result)
```

Java Solution

```java
class Solution {
    // Class variables to hold the input array and its size, and to accumulate the answer.
    private int[] nodeCosts;
    private int treeSize;
    private int totalIncrements;

    // Method to initiate the process and return the final answer.
    public int minIncrements(int treeSize, int[] nodeCosts) {
        this.treeSize = treeSize;
        this.nodeCosts = nodeCosts;
        totalIncrements = 0; // Initializing the total increments to 0.

        // Start the depth-first search from the root node, which is index 1 in this 1-indexed array.
        dfs(1);

        // Return the accumulated total increments needed for the tree.
        return totalIncrements;
    }

    // Recursive depth-first search to calculate the increments required.
    private int dfs(int index) {
        // Base case: If the node is a leaf, return its cost.
        if ((index << 1) > treeSize) {
            return nodeCosts[index - 1];
        }

        // Calculate the required increments for the left child.
        int left = dfs(index << 1);
        // Calculate the required increments for the right child.
        int right = dfs(index << 1 | 1);

        // Increment the total by the difference between the costs of left and right subtrees.
        totalIncrements += Math.abs(left - right);

        // Return the cost of this node plus the maximum of its children's cost.
        // This ensures that we are increasing the smaller child's cost to match the larger one's.
        return nodeCosts[index - 1] + Math.max(left, right);
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>
using namespace std;

class Solution {
public:
    // Method that computes the minimum increments needed for the binary tree
    // represented by 'cost', given the number of nodes 'n'.
    int minIncrements(int n, vector<int>& cost) {
        // Initialize the answer to 0
        int totalIncrements = 0;

        // Define a lambda function for depth-first search computation.
        // This lambda function is recursive and calculates the minimum increments
        // for the subtree rooted at node 'i'.
        function<int(int)> dfs = [&](int i) -> int {
            // If the current index 'i' has children beyond the range of the tree, return its cost.
            if ((i << 1) > n) {
                return cost[i - 1];
            }
            // Recursively compute the cost for the left child.
            int leftCost = dfs(i << 1);
            // Recursively compute the cost for the right child.
            int rightCost = dfs(i << 1 | 1);
            // Increment the answer by the difference between the left and right costs.
            totalIncrements += max(leftCost, rightCost) - min(leftCost, rightCost);
            // Return the cost for the current node which is the sum of its own cost
            // and the maximum of its children's costs.
            return cost[i - 1] + max(leftCost, rightCost);
        };

        // Start the DFS from the root node, which is indexed as 1.
        dfs(1);
        // Return the total number of increments computed.
        return totalIncrements;
    }
};
```

Typescript Solution

```typescript
function minIncrements(nodeCount: number, nodeCosts: number[]): number {
    // Initialize the variable to store the total number of increments required.
    let totalIncrements = 0;

    // Define a Depth-First Search function that calculates the cost to balance the tree starting from a given node index.
    const dfs = (nodeIndex: number): number => {
        // If the current node is a leaf node, return its cost.
        if (nodeIndex * 2 > nodeCount) {
            return nodeCosts[nodeIndex - 1];
        }

        // Recursively calculate the cost for the left and right children of the current node.
        const leftCost = dfs(nodeIndex * 2);
        const rightCost = dfs((nodeIndex * 2) | 1);

        // Increment the total number of increments required by the difference in costs of left and right children.
        totalIncrements += Math.abs(leftCost - rightCost);

        // Return the total cost of this subtree, which is the cost of the current node plus the maximum cost of its children.
        return nodeCosts[nodeIndex - 1] + Math.max(leftCost, rightCost);
    };

    // Start the DFS from the root node, which is at index 1.
    dfs(1);

    // Return the total number of increments required to balance the cost across the tree.
    return totalIncrements;
}
```

Time and Space Complexity

The given code performs a depth-first search (DFS) on a binary tree that is implicitly defined by the input value `n`. For every node `i`, the function `dfs` is called on 2 children if they exist: `i << 1` (left child, representing `2 * i`) and `i << 1 | 1` (right child, representing `2 * i + 1`). The tree is essentially a complete binary tree up to the level where the nodes have values less than or equal to `n`. At each call of `dfs`, it computes the maximal cost of making the subtree rooted at `i` balanced.

Time Complexity

Since each non-leaf node makes two recursive calls, and each recursive call works on half of the remaining nodes (binary tree property), the time complexity is determined by the number of nodes in the tree:

- For a complete binary tree with `n` nodes, the number of edges is `n - 1`.
- The complexity is proportional to the number of edges traversed, which is bound by `O(n)`.

Therefore, the overall time complexity of the code is `O(n)`, where `n` is the number of input nodes in the tree.

Space Complexity

The space complexity of the code is determined by two factors: the recursion call stack and the nonlocal variable `ans` used to accumulate results.

- The depth of the recursive call stack for a complete binary tree is `log(n)` where `n` is the number of nodes. This is because we only go as deep as the height of the tree.
- The variable `ans` does not depend on the size of the input but is a single integer used to accumulate the cost.

Thus, the space complexity is `O(log(n))` reflecting the depth of the recursion call stack for a complete binary tree of `n` nodes.