968. Binary Tree Cameras

Hard | Tree | Depth-First Search | Dynamic Programming | Binary Tree

Problem Description

In this problem, we are provided with a binary tree and tasked with placing cameras on the tree nodes. Each camera has a limited range, being able to monitor the node it is placed on, its parent, and its immediate children. Our goal is to determine the minimum number of cameras required so that every node in the tree is under surveillance. This is an optimization problem where we want to minimize the total number of cameras used while ensuring that no node is left unmonitored.

This kind of problem might imply a need for strategic placement of cameras to cover as many nodes as possible. It also implies that we should be looking for a solution that, possibly through recursion or dynamic programming, allows us to optimize coverage at each step while considering the impact of camera placement on covering parent and children nodes.

Intuition

The intuition behind the solution is based on a post-order traversal of the binary tree, where we analyze the tree from the bottom up. This way, we can make decisions on camera placements starting at the leaf nodes and moving upwards. The idea is that we want to place cameras as low in the tree as possible to free up the nodes above them to potentially cover more territory with fewer cameras.

In the provided solution, `dfs(root)` is our recursive function that returns a tuple `(a, b, c)`:

- `a` represents the minimum number of cameras needed if a camera is placed on the current node.
- `b` represents the minimum number of cameras needed if the current node is covered by a camera, but not necessarily having a camera placed on it. This could mean that either of the children has a camera, which also covers the parent.
- `c` represents the minimum number of cameras if the children are covered but neither the children nor the current node has a camera. This scenario is assuming the current node will be covered by a camera placed on its parent.

The solution applies a bottom-up approach by considering three different scenarios for camera placement on a given node and its children:

1. Place a camera at the current node, and find the minimum number of cameras required for the left and right subtrees.
2. Cover the current node without placing a camera on it, which means one of its children must have a camera.
3. Assume the current node is covered by placing a camera on its parent.

The approach involves a dynamic programming mindset where each node's state is determined by the optimal states of its children. By the time the recursion comes back up to the root of the tree, we have found the minimum number of cameras by considering all possible configurations from the bottom up.

At the end, we compare the number of cameras needed if a camera is placed at the root (`a`) and the number of cameras needed if the root is covered by its children (`b`) and return the fewer of the two as our answer.

Solution Approach

The solution provided uses a recursive function, `dfs(root)`, which follows the depth-first search pattern. This function is key to calculating the optimal camera placement at each node using dynamic programming concepts.

Here's a step-by-step breakdown:

1. Base Case: If `root` is `None` (indicating we've reached a leaf's child), return `(inf, 0, 0)`. This implies:
   - If we place a camera here (which we can't because it's `None`), the cost is `infinity` (representing an invalid placement).
   - If we cover this node (which is not needed because it's `None`), no camera is needed, so the cost is 0.
   - If we don't place a camera and it's not covered (not applicable here), the cost is also 0.
2. Recursive Case: Call `dfs()` recursively on the left and right child nodes of the current node to compute `la`, `lb`, `lc` for the left child and `ra`, `rb`, `rc` for the right child. These variables represent the same three state costs for the left and right subtrees, as described previously:
   - `la`, `ra`: Minimum cameras needed if a camera is placed on the left or right child, respectively.
   - `lb`, `rb`: Minimum cameras if the left or right child is covered but does not have a camera.
   - `lc`, `rc`: Minimum cameras if the left or right child's children are covered, but neither the child nor its children have a camera.
3. Compute the minimum number of cameras needed with different scenarios:
   - If a camera is placed at the current node (`a`), it's 1 (for the camera at the current node) plus the minimum of the three possibilities for both the left and right children (`la + ra`, `la + rb`, `lb + ra`, `lb + rb`).
   - If the current node is covered without a camera on it (`b`), we can't have `lc` or `rc` because this means no camera is present in the subtrees and the current node wouldn't be covered. Thus, we only consider `la + rb` (camera on left child covers the current node), `la + ra` (camera on either child), or `lb + ra` (camera on right child covers the current node).
   - If the current node doesn't have a camera, and it's children are covered (`c`), it implies `lb + rb`—both children must be covered without a camera on the current node, assuming the current node will be covered by its parent.
4. The values (a, b, c) are returned to the parent call according to the recursive progression.

5. At the root level, we now have the minimum number of needed cameras for covering the root node itself (`a`) or just covering it by its children (`b`). The last value _ is ignored as it only applies when the node's parent is responsible for covering it, which doesn't make sense for the root node since it has no parent.

6. Finally, we return the minimum of `a` and `b`, representing the minimum number of cameras to cover the entire tree, whether by placing a camera at the root or by having it covered by cameras placed on its children.

By employing the dynamic programming approach, we avoid redundant calculations, and each node only requires a constant number of computations, resulting in an efficient overall algorithm to solve the problem.

Example Walkthrough

To illustrate the solution approach, let's use a simple binary tree example:

```
    1
   / \
  2   3
 / \
4   5
```

Here is a walk-through of how the algorithm would work on this tree:

1. During the post-order traversal, `dfs()` is called on node 4. Since node 4 is a leaf, it has no children. According to our base case, it returns `(inf, 0, 0)` - no cameras are needed here as it has no child, and the cost of placing a camera is `inf` as it's not required.

2. Similarly, `dfs()` on node 5 returns `(inf, 0, 0)`.

3. The recursion moves up to node 2. `dfs(2)` calls `dfs(4)` and `dfs(5)`, receiving from both `(inf, 0, 0)`. Now it calculates the minimum cameras for itself - `a`, `b`, and `c`:

   - a: 1 + `min(lb, lc) + min(rb, rc)` = 1 + `min(0, 0) + min(0, 0)` = 1 camera.
   - b: `min(la + min(rb, rc), ra + min(lb, lc))` but `la` and `ra` are `inf`, so this isn't considered.
   - c: `lb + rb` = `0 + 0` = 0 cameras (it assumes it will be covered by a camera on its parent).

   `dfs(2)` returns `(1, inf, 0)` to its parent.

4. `dfs()` on node 3, which is a leaf and has no children, also returns `(inf, 0, 0)`.

5. Finally, `dfs()` is called on node 1, the root. It gets `(1, inf, 0)` from the left child (node 2) and `(inf, 0, 0)` from the right child (node 3), and calculates:

   - a: 1 + `min(1, inf, 0) + min(inf, 0, 0)` = 2 cameras.
   - b: `min(la + min(rb, rc), ra + min(lb, lc))` = `min(1 + 0, inf + 0)` = 1 camera (it takes the left child's `la` because `ra` is `inf`).
   - `c` is not considered for the root.

   At the root level, we compare `a` and `b`, which are 2 and 1, respectively, and since 1 is less, we need only one camera for node 1 to cover the entire tree.

So the answer for the minimum number of cameras needed for this tree is 1, which covers all nodes.

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 minCameraCover(self, root: Optional[TreeNode]) -> int:
        """
        Calculates the minimum number of cameras needed to monitor all nodes in a binary tree.
        :param root: The root of the binary tree.
        :return: The minimum number of cameras.
        """

        # Define a limit value as a large number as a substitute for infinity
        limit_value = float('inf')

        def dfs(node):
            """
            Post-order traversal to determine the state of each node for the camera placement.
            :param node: The current tree node being processed.
            :return: A tuple with three elements representing different states:
                (a) the minimum number of cameras needed if placing a camera at this node.
                (b) the minimum number of cameras needed if the parent of this node has a camera.
                (c) the minimum number of cameras needed if this node is covered but itself and its parent do not have a camera.
            """
            # Base case: if the node is None, we return infinity for a and 0 for both b and c.
            if node is None:
                return limit_value, 0, 0

            # Recursive case: compute the state values for both left and right subtrees.
            left_a, left_b, left_c = dfs(node.left)
            right_a, right_b, right_c = dfs(node.right)

            # Minimum cameras if placing a camera at the current node.
            a = min(left_a, left_b, left_c) + min(right_a, right_b, right_c) + 1
            # Minimum cameras if the parent of this node has a camera.
            b = min(left_a + right_b, left_b + right_a, left_a + right_a)
            # Minimum cameras if this node is covered without needing a camera at the node and its parent.
            c = left_b + right_b

            return a, b, c

        # Initial call to dfs with the root node of the tree.
        min_camera_with_root, min_camera_with_parent, _ = dfs(root)

        # Return the minimum number of cameras between putting a camera at the root or at its children.
        return min(min_camera_with_root, min_camera_with_parent)

# The TreeNode class and Solution class would be used together to solve the problem for a specific binary tree by creating an instanc
# and calling the minCameraCover method with the root of the tree.
```

Java Solution

```java
// TreeNode represents a node in a binary tree with a value, and a left and right child.
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 {
    // Calculates the minimum cameras needed to cover the binary tree.
    public int minCameraCover(TreeNode root) {
        int[] result = postOrderTraversal(root);
        // The camera can be either on the current node or on its children.
        // The minimum of the two positions should be returned.
        return Math.min(result[1], result[2]);
    }

    // Performs a post-order traversal of the binary tree to determine where to place cameras.
    private int[] postOrderTraversal(TreeNode node) {
        if (node == null) {
            // If the node is null, we return large values for cases 0 and 1 because they are invalid;
            // and 0 for case 2 which means no camera needed when there is no node.
            return new int[] {Integer.MAX_VALUE / 2, 0, 0};
        }

        // Traverse the left child.
        int[] left = postOrderTraversal(node.left);
        // Traverse the right child.
        int[] right = postOrderTraversal(node.right);

        // Case 0: Place camera on this node.
        int placeCamera = 1 + Math.min(left[0], Math.min(left[1], left[2])) +
                              Math.min(right[0], Math.min(right[1], right[2]));

        // Case 1: No camera on this node. Minimum value from children if one of them has a camera or both.
        int noCameraHere = Math.min(left[0] + right[1], Math.min(left[1] + right[0], left[0] + right[0]));

        // Case 2: No camera on this node or children nodes. Both children nodes are covered.
        int noCameraAtChildren = left[1] + right[1];

        return new int[] {placeCamera, noCameraHere, noCameraAtChildren};
    }
}
```

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) {}
};

/**
 * Status to hold the state information for each tree node during DFS.
 */
struct Status {
    int withCamera;
    int withoutCameraCoveredByParent;
    int withoutCameraCoveredByChildren;
};

class Solution {
public:
    int minCameraCover(TreeNode* root) {
        auto [withCamera, withoutCameraCovered, _] = dfs(root);
        // We choose the minimum cameras needed whether the root has a camera or is covered by children.
        return min(withCamera, withoutCameraCovered);
    }

    Status dfs(TreeNode* node) {
        // If it's a null node, return a large number for withCamera since we don't place cameras on null nodes,
        // and 0 for the other two statuses as they need no cameras.
        if (!node) {
            return {INT_MAX / 2, 0, 0};
        }
        // Recursively calculate the status for left and right subtrees.
        auto [leftWithCamera, leftWithoutCameraCovered, leftWithoutCamera] = dfs(node->left);
        auto [rightWithCamera, rightWithoutCameraCovered, rightWithoutCamera] = dfs(node->right);

        // Case where the current node has a camera
        int withCamera = 1 + min({leftWithCamera, leftWithoutCameraCovered, leftWithoutCamera}) +
                             min({rightWithCamera, rightWithoutCameraCovered, rightWithoutCamera});

        // Case where the current node doesn't have a camera but is covered by a child
        int withoutCameraCoveredByChildren = leftWithoutCamera + rightWithoutCamera;

        // Case where the current node doesn't have a camera but is covered by its parent
        // Choose the minimum of children's cameras and children's coverage by their own children or parent
        int withoutCameraCoveredByParent = min({leftWithCamera + rightWithCamera,
                                                leftWithCamera + rightWithoutCameraCovered,
                                                leftWithoutCameraCovered + rightWithCamera});

        // Return the computed statuses for the current node.
        return {withCamera, withoutCameraCoveredByParent, withoutCameraCoveredByChildren};
    }
};
```

Typescript Solution

```typescript
// Define the structure for a binary tree node
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        // 'val' defaults to 0 if not provided
        this.val = val === undefined ? 0 : val;
        // 'left' and 'right' children default to null if not provided
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}

/**
 * Calculate the minimum number of cameras needed to cover the entire binary tree.
 * @param {TreeNode | null} root - The root of the binary tree.
 * @return {number} The minimum number of cameras needed.
 */
function minCameraCover(root: TreeNode | null): number {
    /**
     * Depth-first search helper to determine the state values for each node.
     * @param {TreeNode | null} node - The current node in the binary tree.
     * @return {number[]} An array representing three states:
     *                     0: Minimum cameras needed if a camera is placed at this node.
     *                     1: Minimum cameras needed if a camera is not placed at this node,
     *                        but the node is covered by its children.
     *                     2: Minimum cameras needed if a camera is not placed at this node,
     *                        but the node is covered by a camera placed at its parent.
     */
    function depthFirstSearch(node: TreeNode | null): number[] {
        // If the current node is null, return large numbers for states 0 and 1,
        // and 0 for state 2 since a null node doesn't need coverage.
        if (!node) {
            return [Infinity, 0, 0];
        }
        // Recursively get the state values of the left and right subtrees.
        const [leftMinWithCamera, leftMinWithoutCamera, leftCoveredByParent] = depthFirstSearch(node.left);
        const [rightMinWithCamera, rightMinWithoutCamera, rightCoveredByParent] = depthFirstSearch(node.right);

        // State 0: Place a camera at the current node.
        const minWithCamera = 1 + Math.min(leftMinWithCamera, leftMinWithoutCamera, leftCoveredByParent) +
                                  Math.min(rightMinWithCamera, rightMinWithoutCamera, rightCoveredByParent);

        // State 1: Don't place a camera at the current node, cover it via children.
        const minWithoutCamera = Math.min(leftMinWithCamera + rightMinWithCamera,
                                          leftMinWithCamera + rightMinWithoutCamera,
                                          leftMinWithoutCamera + rightMinWithCamera);

        // State 2: Node is covered by a camera at the parent node.
        const coveredByParent = leftMinWithoutCamera + rightMinWithoutCamera;

        // Return the computed state values for the current node.
        return [minWithCamera, minWithoutCamera, coveredByParent];
    }

    // Obtain the state values from the root of the tree.
    const [minWithRootCamera, minWithoutRootCamera, _] = depthFirstSearch(root);
    // The minimum number of cameras needed is the smaller of the two states:
    // having a camera on the root or not having a camera on the root.
    return Math.min(minWithRootCamera, minWithoutRootCamera);
}
```

Time and Space Complexity

Time Complexity

The time complexity of this function is determined by the number of recursive calls made during the execution. The function `dfs` traverses each node in the binary tree exactly once. Since there are `N` nodes in the binary tree, and at each node, we are performing a constant amount of work, plus the recursive calls to the left and the right children, the time complexity is `O(N)` where `N` is the number of nodes in the tree.

Space Complexity

The space complexity is primarily determined by the height of the tree as it affects the depth of the recursive call stack. In the worst case, the binary tree could be skewed, i.e., each node has only one child, making the height of the tree `O(N)`, which would be the space complexity due to the call stack. In the best case, where the tree is perfectly balanced, the height of the tree would be `O(logN)`, which would be the space complexity.

However, the space complexity also includes the additional variables used in each call of the `dfs` function, which only add constant space. Therefore, the overall space complexity is `O(H)` where `H` is the height of the tree, which ranges between `O(logN)` for a balanced tree and `O(N)` for a skewed tree.