

[dev.to](#)

Solution: Binary Tree Cameras

seanpgallivan

6-7 minutes

*This is part of a series of Leetcode solution explanations ([index](#)). If you liked this solution or found it useful, **please like** this post and/or **upvote** [my solution post on Leetcode's forums](#).*

[Leetcode Problem #968 \(Hard\): Binary Tree Cameras](#)

Description:

(Jump to: [Solution Idea](#) || Code: [JavaScript](#) | [Python](#) | [Java](#) | [C++](#))

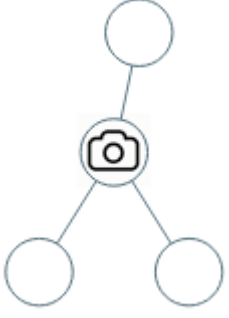
Given a binary tree, we install cameras on the nodes of the tree.

Each camera at a node can monitor **its parent, itself, and its immediate children**.

Calculate the minimum number of cameras needed to monitor all nodes of the tree.

Examples:

|

Example 1:	
Input:	[0,0,null,0,0]
Output:	1
Explanation:	One camera is enough to monitor all nodes if placed as shown.
Visual:	
Example 2:	
Input:	[0,0,null,0,null,0,null,null,0]
Output:	2
Explanation:	At least two cameras are needed to monitor all nodes of the tree. The above image shows one of the valid configurations of camera placement.

Example 2:	
Visual:	

Constraints:

- The number of nodes in the given tree will be in the range $[1, 1000]$.
- Every node has value 0.

Idea:

(Jump to: [Problem Description](#) || Code: [JavaScript](#) | [Python](#) | [Java](#) | [C++](#))

(Jump to: [Problem Description](#) || [Solution Idea](#))

(Note: This is part of a series of Leetcode solution explanations. If you like this solution or find it useful, **please upvote** this post.)

Idea:

One of the first realizations that we can make is that we never need to place a camera on a leaf, since it would always be better to place a camera on the node *above* a leaf. This should lead us to thinking that we need to start from the bottom of the binary tree and work our way up.

This naturally calls for a **depth first search (DFS)** approach with a **recursive** helper function (**dfs**). We can navigate to the lowest part of the tree, then deal with placing cameras on the way back up the recursion stack, using the **return** values to pass information from child to parent.

First, we should consider the different information that will be necessary to pass up to the parent about the child node, and in fact there are only three:

- Nothing below needs monitoring.
- A camera was placed below and can monitor the parent.
- An unmonitored node below needs a camera placed above.

The next challenge is to identify the different scenarios that we'll face once we've collected the values (**val**) of the children of the current **node**. Again, there are three scenarios:

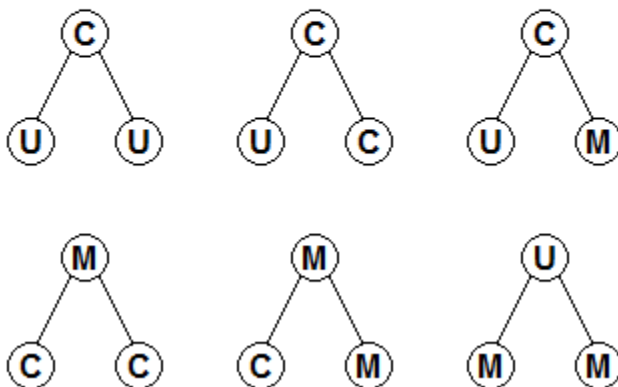
- No child needs monitoring, so hold off on placing a camera and instead **return** a value that indicates that the parent will have to place one.
- One or more of the children need monitoring, so we will have to place a camera here. We'll want to **return** a value indicating that the parent will be monitored.

- One of the children has a camera and the other child either has a camera or doesn't need monitoring (otherwise we would trigger the second scenario instead). This tree is fully monitored, but has no monitoring to provide to the parent; it will **return** the same value as a **null** branch.

U = Unmonitored. Needs camera above.

M = Monitored (or null). Needs nothing.

C = Camera. Can monitor parent.



With all this in mind, we can let the **return** value indicate how we move from one state to another. At each **node** if the combined **val** from below is greater than **2**, then we need to place a camera. If so we should increment our counter (**ans**) before moving on.

One last tricky piece is the **root** node. If the **root** node returns a value indicating that it still needs a camera, we should add **1** to **ans** before we **return** it.

- **Time Complexity:** $O(N)$ where N is the number of nodes in the binary tree
- **Space Complexity:** $O(M)$ where M is the maximum depth of the binary tree, which can range up to N , for the recursion stack

Javascript Code:

(Jump to: [Problem Description](#) || [Solution Idea](#))

```
var minCameraCover = function(root) {  
  let ans = 0  
  const dfs = node => {  
    if (!node) return 0  
    let val = dfs(node.left) + dfs(node.right)  
    if (val === 0) return 3  
    if (val < 3) return 0  
    ans++  
    return 1  
  }  
  return dfs(root) > 2 ? ans + 1 : ans  
};
```

❏ ❏ ❏

Python Code:

(Jump to: [Problem Description](#) || [Solution Idea](#))

```
class Solution:  
    ans = 0  
    def minCameraCover(self, root: TreeNode) -> int:  
        def dfs(node: TreeNode) -> int:  
            if not node: return 0  
            val = dfs(node.left) + dfs(node.right)  
            if val == 0: return 3  
            if val < 3: return 0
```

```
        self.ans += 1
        return 1
    return self.ans + 1 if dfs(root) > 2 else
self.ans
```

```
[] []
```

Java Code:

(Jump to: [Problem Description](#) || [Solution Idea](#))

```
class Solution {
    private int ans = 0;
    public int minCameraCover(TreeNode root) {
        return dfs(root) > 2 ? ans + 1 : ans;
    }
    public int dfs(TreeNode node) {
        if (node == null) return 0;
        int val = dfs(node.left) + dfs(node.right);
        if (val == 0) return 3;
        if (val < 3) return 0;
        ans++;
        return 1;
    }
}
```

```
[] []
```

C++ Code:

(Jump to: [Problem Description](#) || [Solution Idea](#))

```
class Solution {
public:
    int minCameraCover(TreeNode* root) {
        return dfs(root) > 2 ? ans + 1 : ans;
    }
    int dfs(TreeNode* node) {
        if (!node) return 0;
        int val = dfs(node->left) + dfs(node->right);
        if (val == 0) return 3;
        if (val < 3) return 0;
        ans++;
        return 1;
    }
private:
    int ans = 0;
};
```

[[[[]]]]