My Notes on Diameter of a Binary Tree (LeetCode Problem)

1. Problem Statement

Given the root of a binary tree, return the length of the **diameter** of the tree.

- The **diameter** is the longest path between any two nodes in the tree.
- This path **may or may not pass through the root**.
- The **length** is the number of **edges** in this path.

Example:

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

Diameter = 3 (Path: 4 → 2 → 5 or 4 → 2 → 1 → 3)

---

2. My Initial Thought Process

When I first saw this problem, my approach was simple:

1. I knew that the diameter of a tree is determined by the **longest path** between any two nodes.
2. This longest path could pass **through the root** or be entirely **inside one of the subtrees**.
3. To compute this, I decided to:
   - Find the **height (or level)** of the left and right subtree for each node.
   - Sum them up (leftHeight + rightHeight) to get the **diameter** at that node.
   - Keep track of the maximum diameter encountered so far.
   - Finally, return this maximum value.

At first, this approach seemed intuitive, so I implemented it.

---

3. My Code

```
class Solution {
public:
  void helper(TreeNode* root, int* maxDeci) {
    if (root == NULL) return;
```

```
        int Deci = level(root->right) + level(root->left);
        *maxDeci = max(*maxDeci, Deci);
        helper(root->left, maxDeci);
        helper(root->right, maxDeci);
    }
    int level(TreeNode* root) {
        if (root == NULL) return 0;
        return 1 + max(level(root->right), level(root->left));
    }
    int diameterOfBinaryTree(TreeNode* root) {
        int maxDeci = 0;
        helper(root, &maxDeci);
        return maxDeci;
    }
};
```

---

4. What I Realized? (Issue in My Approach)

After implementing my code, I noticed a serious inefficiency:

- My level(root) function was **recomputing heights** multiple times for the same nodes.
- Each node calls level(root->left) and level(root->right), which in turn recursively calls itself again, leading to a **lot of redundant computations**.
- This resulted in an **$O(N^2)$ time complexity**, which is very slow for large trees.

Example of Redundant Computation:

If a node has left and right children, I was recalculating the height of its left and right subtree **from scratch** every time instead of storing it.

---

5. Conclusion

- My initial approach was conceptually correct but **inefficient**.
- The excessive recomputation of height made the solution **slow for large trees**.
- A more **optimized approach** would compute both **height and diameter** in a single traversal using **post-order traversal (DFS)**.
- The current code still works, but can be **improved significantly**.