337. House Robber III

Medium | Tree | Depth-First Search | Dynamic Programming | Binary Tree | Leetcode Link

Problem Description

In this problem, you're given a binary tree that represents houses in a neighborhood. The root of the tree is the main entrance. Each house in this neighborhood is connected to one parent house and to two child houses, except for the leaf houses which do not have any child houses. This setup forms a binary tree structure. The goal is to calculate the maximum amount of money a thief can steal without alerting the police. The catch is that the thief cannot rob two directly-linked houses on the same night because this would trigger an alarm and alert the police.

Intuition

The intuition behind the solution is to use a depth-first search (DFS) technique and dynamic programming to explore all possible combinations and make the optimal choice at each house (or node). We perform DFS to reach the bottom of the tree and then make our way up, deciding at each step whether it's more profitable to rob the current house or not.

There are a few points we should consider to understand the solution:

- Robbing a house means we cannot rob its children, but we can rob its grandchildren.
- We should not rob two adjacent houses (in the way of the tree connections).

The solution uses a helper function, `dfs(root)`, which returns two values for each house: the maximum amount of money obtained by robbing the house (`root.val`) and not robbing the house. Therefore, for each node, we have two scenarios:

1. We rob the current house and therefore add its value to the total amount and can only add the values of not robbing its children to the total (`root.val + lb + rb`).
2. We do not rob the current house and hence we take the maximum amounts that we can obtain whether we robbed or did not rob each of its children (`max(la, lb) + max(ra, rb)`).

At each step, we make the decision that gives more money. We accumulate these decisions to calculate our final answer. The efficiency of this approach lies in the fact that we're only visiting each node once and computing the optimal outcome at each node using the results from its children.

Solution Approach

The solution makes use of a bottom-up approach to dynamic programming. Here's a step-by-step breakdown of the algorithm used in the `rob` function:

1. Define the recursive function `dfs` inside the `rob` function. The `dfs` function takes a node (`root`) from the binary tree as its parameter.

2. The `dfs` function returns a tuple (`int`, `int`) that contains two values:
   - The first value is the maximum amount of money that can be robbed if the current house is robbed this night.
   - The second value is the maximum amount if the current house is not robbed this night.
3. When `dfs` is called on a `None` node (an empty subtree), it returns `(0, 0)` since there's nothing to rob.

4. When `dfs` is called on a non-empty node, it first calls itself recursively for both the left child (`root.left`) and right child (`root.right`). These recursive calls return the best possible outcomes for robbing/not-robbing from the left and right subtrees. Let's denote these returns as `(la, lb)` for the left subtree and `(ra, rb)` for the right subtree.

5. With these results from the left and right children, it calculates what would happen if we rob the current house. If we rob the current house (`root.val`), we can't rob its direct children due to the problem's constraints but we can add what we could rob from the grandchildren (or next level down). So we create a value `root.val + lb + rb`.

6. If we don't rob the current house, we can take the best outcomes from robbing or not robbing its children. We determine this using `max(la, lb) + max(ra, rb)`.

7. The final return statement of the `dfs` function returns the tuple with these two values, which gets passed up the tree.

8. Finally, the `rob` function initiates this process by calling `dfs(root)` - starting the recursive calculation from the root of the binary tree. It then returns the maximum of the two values returned by the `dfs` call, which represents the maximum amount of money that can be robbed without alerting the police.

At the end of this recursive process, the solution has efficiently computed the optimal choice (to rob or not to rob each house) at every step without needing to check all possible combinations explicitly.

This algorithm is efficient because each node is visited only once due to the recursive nature of DFS, and the decision at each node is made using already-computed information from its children. The use of dynamic programming enables us to store and use the results of subproblems, preventing redundant calculations.

Example Walkthrough

Let's illustrate the solution approach with a small example binary tree of houses where the values represent the money in each house:

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

The `rob` function starts by calling the `dfs` function on the root of the tree. Let's walk through the process:

1. The 'dfs' function is called on the root node (house with value 3). This node is not a leaf and it has two children.

2. The 'dfs' function is then recursively called on the left child (house with value 2) and the right child (house with value 5).

3. Starting with the left child (value 2), it's not a leaf and has one right child (value 3). The 'dfs' function calls on this right child, which is a leaf. The call returns `(0, 0)` since there are no further children.

4. For the left child (value 2), we now decide:
   - If we rob this house, we can only add what we could rob from its grandchild (0) since it's a leaf. Hence, `2 + 0 = 2` for robbing the house.
   - If we don't rob the house, we take the best of robbing or not robbing the grandchild (which are both 0). Therefore, not robbing yields `max(0, 0) = 0`.

   So, the 'dfs' call for the left child (value 2) returns `(2, 0)`.

5. Moving on to the right child (value 5), it's also not a leaf and has one right child (value 1). The 'dfs' function is called on this right child, which is a leaf, and it returns `(0, 0)`.

6. For the right child (value 5), the decision is:
   - If we rob the house, we add its value to what we could rob from its grandchild (0), hence `5 + 0 = 5`.
   - If we don't rob the house, we consider the grandchild and get `max(0, 0) = 0`.

   Thus, the 'dfs' call for the right child (value 5) returns `(5, 0)`.

7. Now, we're back at the root house (value 3). We have the figures for robbing/not robbing its children, so we make our calculations:
   - If we rob the root house, we can't rob its children, but we can include what we could get from its grandchildren. Thus, we have `3 + 0 (from not robbing left child) + 0 (from not robbing right child) = 3`.
   - If we don't rob the root, we look at the best results from its children and combine those: `max(2, 0) + max(5, 0) = 2 + 5 = 7`. Normally we would have different numbers to consider here if the children were robbed or not, but in our case, the values are the same since the grandchildren nodes are leaves (hence giving 0).

   Therefore, the 'dfs' call for the root returns `(3, 7)`.

8. Finally, the 'rob' function takes the maximum of the two values from the root's 'dfs' return value, which is `max(3, 7) = 7`. This means the maximum amount of money the thief can steal without alerting the police is 7.

This small example showcases the essence of the algorithm. The recursive nature of the 'dfs' function allows for an efficient and elegant bottom-up approach, only visiting each node once but always making the optimal decision based on the precomputed outcomes of its children.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def rob(self, root: Optional[TreeNode]) -> int:
        # Helper function to perform depth-first search on the tree.
        def dfs(node: Optional[TreeNode]) -> (int, int):
            # If the current node is None, return a tuple of zeros.
            if node is None:
                return 0, 0

            # Recursively calculate the values for the left subtree.
            # left_with_root is the maximum amount of money that can be robbed
            # including the current node's left child.
            # left_without_root is the maximum amount that can be robbed
            # without including the current node's left child.
            left_with_root, left_without_root = dfs(node.left)

            # Recursively calculate the values for the right subtree.
            # right_with_root is the maximum amount of money that can be robbed
            # including the current node's right child.
            # right_without_root is the maximum amount that can be robbed
            # without including the current node's right child.
            right_with_root, right_without_root = dfs(node.right)

            # When robbing the current node, we cannot rob its children.
            with_current = node.val + left_without_root + right_without_root

            # When not robbing the current node, we can choose to rob or not rob each child independently.
            without_current = max(left_with_root, left_without_root) + max(right_with_root, right_without_root)

            # Return a tuple of the maximum amount of money that can be robbed with and without the current node.
            return with_current, without_current

        # Start DFS from the root and calculate the maximum amount that can be robbed.
        return max(dfs(root))

# Note: The Optional[TreeNode] type hint requires importing Optional from typing.
# If not already imported, you need to add: from typing import Optional

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
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 {
    /**
     * Computes the maximum amount of money that can be robbed from the binary tree.
     *
     * @param root The root of the binary tree.
     * @return The maximum amount of money that can be robbed.
     */
    public int rob(TreeNode root) {
        int[] results = robSubtree(root);
        // The maximum of robbing current node and not robbing the current node
        return Math.max(results[0], results[1]);
    }

    /**
     * Performs a depth-first search to find the maximum amount of money
     * that can be robbed from the current subtree.
     *
     * @param node The current node of the binary tree.
     * @return An array containing two elements:
     *         [0] - The maximum amount when the current node is robbed.
     *         [1] - The maximum amount when the current node is not robbed.
     */
    private int[] robSubtree(TreeNode node) {
        if (node == null) {
            // Base case: If the current node is null, return 0 for both cases.
            return new int[2];
        }
        // Results from left and right subtrees.
        int[] leftResults = robSubtree(node.left);
        int[] rightResults = robSubtree(node.right);

        // Robbing the current node
        int robNode = node.val + leftResults[1] + rightResults[1];
        // Not robbing the current node (taking the max of robbing or not robbing children)
        int notRobNode = Math.max(leftResults[0], leftResults[1]) + Math.max(rightResults[0], rightResults[1]);

        // An array of two elements corresponding to robbing or not robbing the current node
        return new int[] {robNode, notRobNode};
    }
}

```

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) {}
};

class Solution {
public:
    int rob(TreeNode* root) {
        // A function to perform a depth-first search (DFS) on the tree.
        // It returns a pair, where the first value is the maximum amount of money
        // that can be robbed when the current node is robbed, and the second value is the
        // maximum amount that can be robbed when the current node is not robbed.
        function<pair<int, int>(TreeNode*)> dfs = [&](TreeNode* node) -> pair<int, int> {
            if (!node) {
                // If the current node is null, return (0,0) since no money can be robbed.
                return {0, 0};
            }

            // Postorder traversal: calculate results for the left and right subtrees.
            auto [left_with_rob, left_without_rob] = dfs(node->left);
            auto [right_with_rob, right_without_rob] = dfs(node->right);

            // When the current node is robbed, its children cannot be robbed.
            int with_rob = node->val + left_without_rob + right_without_rob;
            // When the current node is not robbed, the maximum of rob and not_rob from each
            // of its children can be summed up for the maximum result.
            int without_rob = max(left_with_rob, left_without_rob) + max(right_with_rob, right_without_rob);

            // Pair representing the maximum amounts if the current node is robbed or not.
            return {with_rob, without_rob};
        };

        // Get the maximum values for the root, rob and not_rob.
        auto [root_with_rob, root_without_rob] = dfs(root);

        // Return the maximum of the two for the root node, deciding to rob it or not.
        return max(root_with_rob, root_without_rob);
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
  constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
    this.val = (val === undefined ? 0 : val);
    this.left = (left === undefined ? null : left);
    this.right = (right === undefined ? null : right);
  }
}

/**
 * Given a binary tree where each node has a value, returns the maximum amount of
 * money you can rob without robbing any two directly-connected houses.
 *
 * @param {TreeNode | null} root - The root of the binary tree.
 * @returns {number} The maximum amount of money that can be robbed.
 */
function rob(root: TreeNode | null): number {
  /**
   * Performs a depth-first search on the binary tree to calculate the maximum money
   * that can be robbed without directly robbing two connected nodes.
   *
   * @param {TreeNode | null} node - The current node being visited.
   * @returns {[number, number]} A tuple, where the first element is the maximum
   * money that can be robbed when the current node is included and the second element
   * is the maximum when the current node is excluded.
   */
  function performDfs(node: TreeNode | null): [number, number] {
    // Base case: If there's no node, return [0, 0] as there's nothing to rob.
    if (!node) {
      return [0, 0];
    }

    // Recursively perform DFS on the left and right subtrees.
    const [leftWithCurrent, leftWithoutCurrent] = performDfs(node.left);
    const [rightWithCurrent, rightWithoutCurrent] = performDfs(node.right);

    // Include the current node's value and add the money from child nodes
    // when the current node is not robbed (as you can't rob two directly connected nodes).
    const withCurrent = node.val + leftWithoutCurrent + rightWithoutCurrent;

    // Exclude the current node's value and take the maximum money from either robbing or not
    // robbing the child nodes.
    const withoutCurrent = Math.max(leftWithCurrent, leftWithoutCurrent) + Math.max(rightWithCurrent, rightWithoutCurrent);

    // Return the calculated values in a tuple.
    return [withCurrent, withoutCurrent];
  }

  // Perform the DFS on the root and return the maximum of the two scenarios:
  // robbing or not robbing the root node.
  return Math.max(...performDfs(root));
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because the `dfs` function is called exactly once for each node in the tree. During each call to `dfs`, it performs a constant amount of work (one addition, and a few comparisons) beside the recursive calls to the left and right children. Thus, the total work done is directly proportional to the number of nodes.

Space Complexity

The space complexity is `O(H)`, where `H` is the height of the binary tree. This accounts for the call stack used during the depth-first search. In the worst-case scenario (a skewed tree), the height of the tree can become `N`, resulting in a space complexity of `O(N)`. For a balanced tree, the height `H` is `log(N)`, so the space complexity would be `O(log(N))`. However, since `H` is always less than or equal to `N` and is not constant, `O(H)` is the more accurate representation of the space complexity.