113. Path Sum II

Medium · Tree · Depth-First Search · Backtracking · Binary Tree — Leetcode Link

Problem Description

In this problem, we are given a binary tree and an integer `targetSum`. Our task is to find all the different paths from the root of the binary tree to any of its leaves such that the sum of node values along each path is equal to `targetSum`. A *root-to-leaf path* is defined as a path that begins at the root node and ends at a leaf node, where a leaf node is a node that does not have any children. The solution should return all such paths as lists of node values.

Intuition

The intuition behind the solution is to traverse the tree in a depth-first search (DFS) manner. We can start from the root and recursively visit each node's children until we reach the leaf nodes. As we traverse the tree, we can keep track of the sum of the values of the nodes along the path and the nodes themselves in a list.

When we reach a leaf node (a node with no children), we check if the sum of the path is equal to `targetSum`. If it is, we found a valid path and add a copy of the current path list to an answer array that we maintain to store all valid paths. If the sum is not equal to `targetSum`, we simply backtrack and continue exploring other nodes and paths.

The key aspects of the solution are:

- Starting from the root, perform DFS to explore all paths to leaf nodes.
- Keep a running sum of node values for the current path being explored.
- Keep a temporary list to track the node values for the current path.
- If a leaf node is reached where the running sum is equal to `targetSum`, add the current path to the solution list.
- After processing a node, backtrack by popping the last node value from the temporary path list before returning to the parent node.

This approach allows us to systematically explore all potential paths in the tree and collect the ones that satisfy the condition without missing any.

Solution Approach

The solution uses a recursive depth-first search (DFS) strategy to explore all possible root-to-leaf paths in the binary tree.

Let's walk through the key components of the implementation:

- The `dfs` function is a helper function that we define within the `pathSum` method. It takes two arguments:
  - The `root` of the current subtree that we are exploring.
  - A running sum `s` that starts at 0 and accumulates the sum of node values from the root to the current node.
- A list `t` is used to maintain the current path, representing the sequence of node values from the root to the current node being visited.
- A list `ans` is maintained to store all successful paths that sum up to `targetSum`.

The `dfs` function works as follows:

1. If the current node `root` is `None`, we have hit a dead-end and need to terminate this branch of recursion.
2. We add the current node's value to the running sum `s` and append this value to `t` to track the path.
3. If the current node is a leaf (neither `left` nor `right` child exists), and the running sum `s` equals `targetSum`, we have found a successful path. We make a shallow copy of `t` using `t[:]` and append this copy to `ans` to record the successful path.
4. We recursively call `dfs` on the `left` and `right` children of the current node, with the updated running sum `s`.
5. After exploring both children, we backtrack by popping the last element from `t`, ensuring that `t` reflects the path for its predecessors as we return to the parent node.

By calling `dfs(root, 0)` initially, we begin the search from the root of the entire tree with an initial sum of `0`. We keep exploring recursively as described above until all possible paths have been checked for the condition.

The return value of the `pathSum` function is `ans`, which contains lists of node values for all the root-to-leaf paths that add up to `targetSum`.

This approach effectively employs DFS traversal and backtracking to explore the tree's paths and return only those that meet the given criteria.

Example Walkthrough

Let's consider a binary tree and a `targetSum` of 22 for our example:

```
      5
     / \
    4   8
   /   / \
  11  13  4
 /  \      \
7    2      5
```

We want to find all root-to-leaf paths where the sum of the node values is equal to `targetSum`.

We'll use the DFS approach to navigate through this tree and look for such paths. Using the solution approach described earlier, we will track both the current sum of the path and the path of node values itself as we progress. Here's how it would work, step-by-step:

1. We start with the root node (5), add it to our path and running sum `s` is now 5.
2. We move to the left child (4), include it in our path. So, our path is now `[5, 4]` and `s` becomes 9.
3. Continuing deeper, we reach node 11 and our path is `[5, 4, 11]`, with `s` updating to 20.
4. We move to node 7, and now the path becomes `[5, 4, 11, 7]` and `s` is 27. Since this is a leaf, we check if `s` is equal to `targetSum` (22). It is not, so we backtrack to 11.
5. We then visit the right child of 11, which is node 2, making the path `[5, 4, 11, 2]` and `s` is now 22. This is a leaf node and the sum matches `targetSum`, so this path is added to our answer list.
6. After checking both children of node 11, we backtrack to node 4 and then to node 5, and proceed to its right child (8). Our path is now `[5, 8]` and `s` becomes 13.
7. Node 8 has two children, so we explore the left child first (node 13). Although now `s` becomes 26, since 13 has no children, we quickly backtrack to 8 without changing our path list.
8. Moving to the right child of node 8 (node 4), the path and sum update to `[5, 8, 4]` and `s` of 17, respectively.
9. Node 4 has one child, which is a leaf (node 5). We visit it, and the path becomes `[5, 8, 4, 5]` and `s` sums up to 22, which is equal to `targetSum`. We add this path to the list of valid paths.
10. All nodes are visited, and we've now found all the required paths. We populate these into our answers list.

The result of the search using our example tree and `targetSum` is the following list of paths:

```
[
  [5, 4, 11, 2],
  [5, 8, 4, 5]
]
```

The DFS approach has successfully identified all the paths whose node values sum up to the `targetSum`.

Python Solution

```python
# Here we define a TreeNode class for the binary tree nodes.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.current_left = left
        self.current_right = right

class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:
        # Helper method to perform depth-first search.
        def dfs(node, current_sum):
            nonlocal targetSum, paths, current_path
            if node is None:
                return

            # Add the current node's value to the running sum.
            current_sum += node.val
            # Add the current node's value to the current path being evaluated.
            current_path.append(node.val)

            # Check if we're at a leaf node and the running sum equals the target sum.
            if node.current_left is None and node.current_right is None and current_sum == targetSum:
                # If so, add a copy of the current path to the list of paths.
                paths.append(current_path[:])

            # Continue the search on the left and right subtrees.
            dfs(node.current_left, current_sum)
            dfs(node.current_right, current_sum)

            # We've finished exploring from this node, so backtrack and remove the node value from the current path.
            current_path.pop()

        # This will hold all the paths that sum to the target.
        paths = []
        # Temp list to hold the current path being checked.
        current_path = []
        # Initialize the DFS from the root node with a sum of 0.
        dfs(root, 0)
        return paths

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 * This class defines the structure of the tree node which includes
 * a value, and references to the left and right child nodes.
 */
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;
    }
}

/**
 * Solution class where we define the method to find all root-to-leaf paths
 * that add up to a given target sum.
 */
class Solution {
    // A list to store all the paths that sum to the target
    private List<List<Integer>> paths = new ArrayList<>();
    // A temporary list to keep track of the current path
    private List<Integer> currentPath = new ArrayList<>();

    /**
     * Finds all paths from root to leaves that sum to targetSum.
     * @param root The root of the tree.
     * @param targetSum The sum that each path needs to add up to.
     * @return A list of all paths that sum to targetSum.
     */
    public List<List<Integer>> pathSum(TreeNode root, int targetSum) {
        dfs(root, targetSum);
        return paths;
    }

    /**
     * Performs a depth-first search to find all paths with the target sum.
     * @param node The current node in the recursion.
     * @param remainingSum The remaining sum after subtracting the node's value.
     */
    private void dfs(TreeNode node, int remainingSum) {
        if (node == null) {
            // If we've reached a null node, just return.
            return;
        }

        // Subtract the node's value from the remaining sum and add the node's value to the current path
        remainingSum -= node.val;
        currentPath.add(node.val);

        // Check if it's a leaf node and the remaining sum is zero, which means we've found a valid path
        if (node.left == null && node.right == null && remainingSum == 0) {
            // If so, add a copy of the current path to the list of valid paths
            paths.add(new ArrayList<>(currentPath));
        }

        // Recurse deeper into the tree
        dfs(node.left, remainingSum);  // Go left
        dfs(node.right, remainingSum); // Go right

        // After exploring both sides, remove the current node's value before going back up the tree
        currentPath.remove(currentPath.size() - 1);
    }
}

```

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:
    // Main function to find all paths that sum to the target.
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        vector<vector<int>> allPaths; // Stores all paths that sum to targetSum.
        vector<int> currentPath; // Holds the current path being evaluated.

        // Helper function to perform DFS on the tree
        function<void(TreeNode*, int)> depthFirstSearch = [&](TreeNode* node, int remainingSum) {
            if (!node) // Base case: If the node is null, return.
                return;

            remainingSum -= node->val; // Subtract the node's value from the remaining sum.
            currentPath.emplace_back(node->val); // Add the current node's value to the path.

            // Check if it's a leaf node and the remaining sum is 0; if so, store the path.
            if (!node->left && !node->right && remainingSum == 0) {
                allPaths.emplace_back(currentPath);
            }

            // Continue searching in the left and right subtrees.
            depthFirstSearch(node->left, remainingSum);
            depthFirstSearch(node->right, remainingSum);

            // Backtrack: remove the last element before returning to the previous caller.
            currentPath.pop_back();
        };

        // Start the depth-first search with the original root and target sum.
        depthFirstSearch(root, targetSum);

        // Return all found paths after the search is complete.
        return allPaths;
    }
};

```

Typescript Solution

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

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Main function to find all paths that sum to the target value
 * @param root The root of the binary tree
 * @param targetSum The sum to find in the binary tree
 * @return An array of arrays, where each array represents a path that sums to targetSum
 */
const pathSum = (root: TreeNode | null, targetSum: number): number[][] => {
    const allPaths: number[][] = []; // To store all the valid paths.
    const currentPath: number[] = []; // To keep track of the current path.

    // Helper function to perform DFS (Depth First Search) on the binary tree
    const dfs = (node: TreeNode | null, remainingSum: number) => {
        if (!node) return; // Base case: if the current node is null, do nothing.

        remainingSum -= node.val; // Subtract the current node value from the remaining sum.
        currentPath.push(node.val); // Insert the current node value into the current path.

        // Check if it's a leaf node and the remaining sum is 0, which means we've found a valid path.
        if (!node.left && !node.right && remainingSum === 0) {
            allPaths.push([...currentPath]); // Add a copy of the current path to allPaths.
        }

        dfs(node.left, remainingSum); // Continue DFS on the left subtree.
        dfs(node.right, remainingSum); // Continue DFS on the right subtree.

        currentPath.pop(); // Backtrack: remove the last node value from the current path.
    };

    dfs(root, targetSum); // Start DFS with the root node and the initial targetSum.

    return allPaths; // Return all the valid paths.
};

export { TreeNode, pathSum }; // Exporting the TreeNode class and pathSum function for other modules to use.

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of nodes in the tree. This is because in the worst case, the algorithm will visit each node exactly once while performing depth-first search (DFS).

Space Complexity

The space complexity is more complex to analyze due to the space taken by the recursion stack and the space required to store the paths.

1. **Recursion Stack**: The space taken by the recursion stack is proportional to the height of the tree. In the worst case for a skewed tree, the height can be `n` leading to `O(n)` complexity. In the best case for a balanced tree, the height will be `log(n)`, leading to `O(log(n))` complexity.
2. **Path Storage**: For every leaf node, we potentially have a path equal in size to the tree's height. In the worst case (all paths are saved), this results in `O(n * h)` space, where `h` is the height of the tree. However, since the height can range from `log(n)` for a balanced tree to `n` for a skewed tree, the space complexity for storing paths ranges from `O(n * log(n))` to `O(n^2)`.

Considering these aspects, the total space complexity of the code can be `O(n * h)` where `h` is the height of the tree. The exact bound depends on the shape of the tree. In the case of a balanced tree, the space complexity would be `O(n * log(n))`, while for a completely unbalanced tree, it would be closer to `O(n^2)`.