666. Path Sum IV

Medium Tree Depth-First Search Array Hash Table Binary Tree Leetcode Link

Problem Description

In this problem, we're given a binary tree with a depth smaller than 5, which is represented as an array of three-digit integers. The representation of the tree using integers has specific rules:

1. The first digit (`the hundreds`) indicates the depth of the binary tree node (`d`), which can range from 1 to 4.
2. The second digit (`the tens`) indicates the position of the node (`p`) within its level, similar to its position in a full binary tree, ranging from 1 to 8.
3. The third digit (`the units`) is a value of the node (`v`), which can be between 0 and 9.

The task is to find the sum of all root-to-leaf paths' values. The array is given in ascending order, and it's ensured that it represents a valid connected binary tree.

A path is defined as a sequence of nodes from the root of the tree to any leaf node, and the path sum is the sum of values of nodes along that particular path.

Intuition

To solve this problem, we can use Depth-First Search (DFS), a standard tree traversal algorithm that goes as deep as possible in one direction before backtracking. The goal is to traverse the tree from the root to each leaf and accumulate the values of the nodes along these paths.

Here's the intuition behind the solution approach:

1. **Represent the Tree Structure**: Since the tree is given as a sequence of integers, we first need to map this representation to one that we can use for DFS. The mapping is based on depth and position. We can use a dictionary to store the nodes, where each key represents the depth and position (without the value digit), and the corresponding value is the node's value. This will help us find a node's children.

2. **Implement DFS**: We'll define a recursive function that will take a node (represented as an integer with depth and position information) and a running total sum (initially set to 0). The function will perform the following actions:

   - Check if the current node is a leaf (has no children) and, if so, add the running total sum to the global answer.
   - Otherwise, call the function recursively for the left and right children, increasing the depth and calculating the position for each child based on the rules.
   - Accumulate the node's value to the running total sum as we traverse down the tree.
3. **Handle Edge Cases**: We're guaranteed that the array represents a valid tree. However, we do need to ensure that we're checking for the existence of children nodes using the mapping before trying to traverse them.

4. **Compute the Result**: By running DFS starting from the root, we can compute the sum of all the root-to-leaf paths. The root is represented as `11` since the depth is `1` and position is `1`.

This way, we can explore all paths, sum their values, and obtain the total sum required by the problem.

Solution Approach

To implement the solution to this problem, a few critical components are used:

1. **Dictionary Representation:** A dictionary, denoted as `mp` in the code, is created to efficiently store and access nodes using their hundreds and tens digits as keys (representing depth and position) and their units digit as the value. The dictionary comprehension `{num // 10: num % 10 for num in nums}` achieves this by dividing each number by 10 (to remove the value digit) and then mapping this to the remainder (the value digit).

2. **Depth-First Search (DFS):** The DFS is implemented through a recursive function, `dfs(node, t)`, where `node` is the current node being visited and `t` is the running total sum of values on the path from the root to the current node.

3. **Node Representation and Child Calculation:** Each node is represented as a two-digit number where the first digit correlates to the depth and the second digit corresponds to the position. To find children of a node at depth `d` and position `p`, we calculate the left child as `(d + 1) * 10 + (p * 2) - 1` and the right child as `l + 1`. The multiplication and addition here are based on the properties of a binary tree, where each level has twice as many nodes as the previous, and positions start on the leftmost side at 1 and increase to the right.

4. **Leaf Check and Sum Accumulation:** The function checks to see if the current node is a leaf, meaning it has no children in the mapping `mp`. If it's a leaf, the current path's sum (which includes the current node's value) is added to a global variable `ans`, which accumulates the total sum of all paths.

5. **Recursion:** The DFS function is called recursively on the node's children, passing the updated total sum `t` combined with the current node's value. This is done with `dfs(l, t)` and `dfs(r, t)` for the left and right child respectively.

6. **Global Variable:** Since Python doesn't support passing primitives by reference, and updating a local variable won't affect its value outside the current scope, a nonlocal declaration (`nonlocal ans`) is used to modify the `ans` variable, which holds the total path sum, within the recursive function.

Finally, the DFS is kicked off from the root of the tree, which always has a depth and position of `1` (hence node `11`), and an initial total path sum of `0`. The accumulated sum of all paths, stored in `ans`, is returned as the result after completing all recursions.

The efficiency of this approach lies in its use of recursion and the dictionary, which allows constant-time lookups for the existence of child nodes, as well as the ability to directly calculate children's identifiers from a given node's identifier without needing to construct the actual tree structure.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach.

Suppose we're given the following representation of a binary tree using an array of three-digit integers: `[111, 121, 122, 131, 132]`. Here's what each number means:

- `111`: Root node with a value of 1, at depth 1 and position 1.
- `121`: Left child of the root with a value of 1, at depth 2 and position 1.
- `122`: Right child of the root with a value of 2, at depth 2 and position 2.
- `131`: Left child of node `121` with a value of 1, at depth 3 and position 1 (this is a leaf node).
- `132`: Right child of node `121` with a value of 2, at depth 3 and position 2 (this is a leaf node).

Now let's map this representation to one suitable for depth-first search:

1. **Dictionary Representation:**

   - We create a dictionary `mp` like so:

     ```
     {
       11: 1,  // Root node
       12: 1,  // Left child of root
       13: 2,  // Right child of root
       13: 1,  // Left child of node 12
       14: 2   // Right child of node 12
     }
     ```

   - This maps the depth-position identifier to the node's value.
2. **Implementing DFS:**

   - We define a recursive function `dfs(node, t)`.
   - Initially, we call `dfs(11, 0)` because our root is `111`, and our initial total path sum is `0`.
3. **Traversing the Tree:**

   - Starting with node `11`, we check for its children, which are, in theory, `121` (left child) and `122` (right child).
   - For the left child `121`, we calculate its identifier as `12` (by stripping away the value digit), which is present in `mp`. Therefore, we call `dfs(12, 1)` (`1` being the current total sum plus root's value).
   - For the right child `122`, we also calculate its identifier as `13` and since it's in `mp`, we call `dfs(13, 1)`.
4. **Summing Path Values:**

   - We then traverse to node `121` (`12` in `mp`). We check for children `131` and `132` similarly, ending up calling `dfs(13, 2)` and `dfs(14, 2)` respectively since `12 + 1 = 2`.
   - Both are leaves, so we add their sums `(2 + 1)` and `(2 + 2)` to a global variable `ans`.
5. **Calculating Results:**

   - By recursing through both left and right children for all nodes, we add all the root-to-leaf paths to `ans`.
   - The paths we have taken are: `111 -> 121 -> 131` and `111 -> 121 -> 132`.
   - This results in sums of `1 + 1 + 1 = 3` and `1 + 1 + 2 = 4` respectively.
   - Our `ans` turns out to be `3 + 4 = 7`.

Finally, `ans` contains the sum of the values of all root-to-leaf paths which in our case is `7`. This is returned as the result of our problem.

Python Solution

```python
from typing import List

class Solution:
    def pathSum(self, nums: List[int]) -> int:
        # Helper function for depth-first search from a given node with cumulative total `total`
        def dfs(node, total):
            # If the current node is not in the tree, stop the recursion
            if node not in tree_map:
                return
            # Add the current node's value to the running total
            total += tree_map[node]
            depth, pos = divmod(node, 10) # Split the node code into depth and positional information
            # Calculate the node code for the left and right children
            left_child = (depth + 1) * 10 + (pos * 2) - 1
            right_child = left_child + 1
            # If both children are absent, add the current total to the global 'answer'
            if left_child not in tree_map and right_child not in tree_map:
                nonlocal answer
                answer += total
                return
            # Recurse on the left and right children
            dfs(left_child, total)
            dfs(right_child, total)

        # Initialize the answer variable to accumulate the total path sums
        answer = 0
        # Create a mapping from node codes (depth and position) to values using list comprehension
        tree_map = {num // 10: num % 10 for num in nums}
        # Start the DFS traversal from the root node (code 11) with an initial total of 0
        dfs(11, 0)
        # Return the accumulated total path sums
        return answer

```

Java Solution

```java
class Solution {
    private int totalPathSum; // Use a more descriptive name for the variable 'ans'.
    private Map<Integer, Integer> valueMap; // Rename 'mp' to 'valueMap' for clarity.

    // Computes the sum of all paths in the tree.
    public int pathSum(int[] nums) {
        totalPathSum = 0;
        valueMap = new HashMap<>(nums.length);

        // Store each value in a map with its key as {node level}{node position}
        for (int num : nums) {
            // Dividing by 10 gives us the {node level}{node position}, modulo 10 gives us the value at that node
            valueMap.put(num / 10, num % 10);
        }

        // Start DFS from the root node, which is always 11
        dfs(11, 0);
        return totalPathSum;
    }

    // Performs a depth-first search to calculate all path sums.
    private void dfs(int node, int currentSum) {
        // If the node does not exist, return
        if (!valueMap.containsKey(node)) {
            return;
        }

        // Add the value of the current node to the running sum
        currentSum += valueMap.get(node);

        // Calculate level and position for the current node
        int level = node / 10;
        int position = node % 10;

        // Calculate the node values for the left and right children
        int leftChild = (level + 1) * 10 + (position * 2) - 1;
        int rightChild = leftChild + 1;

        // If the node is a leaf node, add the running sum to the total path sum
        if (!valueMap.containsKey(leftChild) && !valueMap.containsKey(rightChild)) {
            totalPathSum += currentSum;
            return;
        }

        // Continue DFS on the left and right children
        dfs(leftChild, currentSum);
        dfs(rightChild, currentSum);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int totalPathSum; // Holds the total sum of path values
    unordered_map<int, int> nodesMap; // Maps the node id to its value

    // Main function to calculate the sum of the path values.
    int pathSum(vector<int>& nums) {
        totalPathSum = 0; // Initialize total path sum
        nodesMap.clear(); // Clear the map before computation

        // Populate the map with node values. Node id as key and node value as value.
        for (int num : nums) {
            nodesMap[num / 10] = num % 10;
        }

        // Start depth-first search at the root node, which is id 11.
        dfs(11, 0);

        // Return the computed total path sum.
        return totalPathSum;
    }

    // Recursive depth-first search function to compute path sums.
    void dfs(int nodeId, int sum) {
        // Check if node id is not present, and return if it's not.
        if (!nodesMap.count(nodeId)) {
            return;
        }

        sum += nodesMap[nodeId]; // Add the node's value to the running sum.

        // Compute the current node's depth and position.
        int depth = nodeId / 10;
        int position = nodeId % 10;

        // Calculate left and right children's ids.
        int leftChildId = (depth + 1) * 10 + (position * 2) - 1;
        int rightChildId = leftChildId + 1;

        // Check if the node is a leaf node (i.e., it doesn't have any children)
        if (!nodesMap.count(leftChildId) && !nodesMap.count(rightChildId)) {
            totalPathSum += sum; // Add the sum to the total path sum.
            return;
        }

        // Recursive calls for left and right children.
        dfs(leftChildId, sum);
        dfs(rightChildId, sum);
    }
};

```

Typescript Solution

```typescript
// Global variable to hold the total sum of path values.
let totalPathSum: number = 0;

// Global map to associate node ids with their values.
let nodesMap = new Map<number, number>();

/**
 * Function to calculate the sum of the path values for a binary tree represented by an array.
 * @param nums An array representing a binary tree where each element is composed of 3 digits.
 */
function pathSum(nums: number[]): number {
    totalPathSum = 0;   // Initialize the total path sum.
    nodesMap.clear();   // Clear the map before starting the computation.

    // Populate the map with node values where key is node id and value is node value.
    nums.forEach(num => {
        nodesMap.set(Math.floor(num / 10), num % 10);
    });

    // Start depth-first search at the root node which has id 11.
    dfs(11, 0);

    // Return the total sum of path values computed.
    return totalPathSum;
}

/**
 * Recursive depth-first search function to compute path sums.
 * @param nodeId The id of the current node.
 * @param sum The sum of values along the current path.
 */
function dfs(nodeId: number, sum: number): void {
    // Check if the node id exists in the map. If not, return immediately.
    if (!nodesMap.has(nodeId)) {
        return;
    }

    // Add the current node's value to the running sum.
    sum += nodesMap.get(nodeId) || 0;

    // Calculate the depth and position for the current node.
    const depth: number = Math.floor(nodeId / 10);
    const position: number = nodeId % 10;

    // Calculate the ids for the left and right children of the current node.
    const leftChildId: number = (depth + 1) * 10 + (position * 2) - 1;
    const rightChildId: number = leftChildId + 1;

    // Check if the current node is a leaf node (does not have any children).
    if (!nodesMap.has(leftChildId) && !nodesMap.has(rightChildId)) {
        // Since it's a leaf, add the path sum to the total.
        totalPathSum += sum;
        return;
    }

    // Recursively call dfs for the left and right children if they exist.
    dfs(leftChildId, sum);
    dfs(rightChildId, sum);
}

```

Time and Space Complexity

The provided code defines a method `pathSum` which computes the sum of all paths in a binary tree represented in a compact array form where an integer `xyz` represents a node at depth `x` (1-indexed), position `y` (1-indexed) within its level, with value `z`. To calculate the sum of all root-to-leaf paths, the code uses a Depth-First Search (DFS) algorithm.

Time Complexity

The time complexity of the function is `O(N)`, where `N` is the number of elements in the `nums` array. This is because:

- Each element in the array is visited exactly once to populate the `mp` dictionary.
- The `dfs` function is called recursively to explore all possible paths from the root to the leaves. In the worst case, the tree is a full binary tree, and `dfs` will be called exactly `2^h - 1` times, where `h` is the height of the tree. Since the `nums` array can at most represent a binary tree with a height where `2^h - 1` is equal to the length of `nums`, the `dfs` call does not increase the complexity beyond `O(N)`.

Space Complexity

The space complexity of the function is `O(H)`, where `H` is the height of the binary tree represented by the `nums` array. This accounts for:

- The space used by the `mp` dictionary, which is equal to the number of elements in `nums`, i.e., `O(N)`.
- The maximum depth of the recursion stack, which is equal to the height of the tree `H`, i.e., `O(H)`.
- Since `N` can represent a full binary tree, in the worst case, `H` could be `O(logN)` because `N = 2^H - 1` in a full binary tree.

Considering the most constraining factor, the overall space complexity is `O(H)`. In the context of this problem, since the depth of the tree is limited (in practical scenarios by the binary tree representation), the space complexity due to the call stack may also be considered `O(1)` as a constant factor, depending on the constraints of the problem. However, without specific constraints given, we stick with `O(H)`.