543. Diameter of Binary Tree

Easy · Tree · Depth-First Search · Binary Tree

Leetcode Link

Problem Description

The task is to find the diameter of a binary tree. The diameter is the longest path between any two nodes in the tree, which does not necessarily have to involve the root node. Here, the length of the path is given by the number of edges between the nodes. Therefore, if a path goes through the sequence of nodes A → B → C, the length of this path is 2 since there are two edges involved (A to B and B to C).

Intuition

To solve this problem, we use a depth-first search (DFS) approach. The intuition behind using DFS is to explore as far as possible down each branch before backtracking, which helps in calculating the depth (or height) of each subtree rooted at `node`.

For every node, we compute the depth of its left and right subtrees, which are the lengths of the longest paths from this node to a leaf node down the left and right subtrees, respectively. The diameter through this node is the sum of these depths, as it represents a path from the deepest leaf in the left subtree, up to the current node, and then down to the deepest leaf in the right subtree.

The key is to recognize that the diameter at each node is the sum of the left and right subtree depths, and we are aiming to find the maximum diameter over all nodes in the tree. Hence, during the DFS, we keep updating a global or external variable with the maximum diameter found so far.

One crucial aspect of the solution is that for every DFS call, we return the depth of the current subtree to the previous caller (parent node) in order to compute the diameter at the parent level. The current node depth is calculated as `1 + max(left, right)`, accounting for the current node itself plus the deeper path among its left or right subtree.

This approach allows us to travel only once through the tree nodes, maintaining a time complexity of O(N), where N is the number of nodes in the tree.

Solution Approach

To implement the diameter calculation for a binary tree, we use DFS to traverse the tree. Here's a step-by-step breakdown of the algorithm used in the provided code:

1. **Define the `dfs` Function:** This recursive function takes a node of the tree as an argument. Its purpose is to compute the depth of the subtree rooted at that node and update the `ans` variable with the diameter passing through that node.
2. **Base Case:** If the current node is `None`, which means we've reached beyond a leaf node, we return a depth of `0`.
3. **Recursive Search:** We calculate the depth of the left subtree (`left`) and the depth of the right subtree (`right`) by making recursive calls to `dfs(root.left)` and `dfs(root.right)`.
4. **Update Diameter:** We update the `ans` variable (which is declared with the `nonlocal` keyword to refer to the `ans` variable in the outer scope) with the maximum of its current value and the sum of `left` and `right`. This represents the largest diameter found at the current node because it is the sum of the path through the left child plus the path through the right child.
5. **Returning Depth:** Each call to `dfs` returns the depth of the subtree it searched. The depth is the maximum between the depths of its left and right subtrees, increased by 1 to account for the current node.
6. **Start DFS:** We call `dfs` starting at the `root` node to initiate the depth-first search of the tree.
7. **Return the Result:** After traversing the whole tree, `ans` holds the length of the longest path, which is the diameter of the binary tree.

The overall complexity of the solution is `O(N)`, where `N` is the number of nodes in the binary tree since each node is visited exactly once.

This solution uses a DFS pattern to explore the depth of each node's subtrees and a global or "helper" scope variable to track the cumulative maximum diameter found during the entire traversal. The use of recursion and tracking a global maximum is a common strategy for tree-based problems where computations in child nodes need to influence a result variable at a higher scope.

The use of a nonlocal keyword in Python is essential here as it allows the nested `dfs` helper function to modify the `ans` variable defined in the outer `diameterOfBinaryTree` function's scope.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the depth-first search (DFS) algorithm to find the diameter of a binary tree.

Consider a binary tree:

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

In this example, the longest path (diameter) is between node 4 and node 3, which goes through node 2 and node 1. Here's how we can apply the solution approach to this tree:

1. We start DFS with the root node. Let's call `dfs(1)`:
   - It's not `None`, so we continue with the recursion.
2. We encounter node 1 and make recursive calls to its left and right children:
   - `dfs(2)` is called for the left subtree.
   - `dfs(3)` is called for the right subtree.
3. Inside `dfs(2)`:
   - We call `dfs(4)` for the left child and return 1 (since 4 is a leaf node).
   - We call `dfs(5)` for the right child and return 1 (since 5 is a leaf node).
   - We update `ans` to be `max(ans, left + right)` which at this point is `max(0, 1 + 1)` = 2.
   - We return `1 + max(left, right)` to indicate the depth from node 2 to the deepest leaf which equals `1 + 1` = 2.
4. Inside `dfs(3)`:
   - Since node 3 is a leaf node, we return 1.
5. Back to the `dfs(1)`, with the returned values:
   - We received 2 from the left subtree (`dfs(2)`).
   - We received 1 from the right subtree (`dfs(3)`).
   - We update `ans` with the sum of the left and right which is `max(2, 2 + 1)` = 3. This is the diameter passing through the root.
6. DFS is called throughout the entire tree, and the maximum value of `ans` is updated accordingly.
7. Since we've traversed the whole tree, the final `ans` is 3, which is the length of the longest path (diameter) in our binary tree, passing from node 4 to 3 via nodes 2 and 1.

The key steps in the example above show the recursive nature of the solution, updating a global maximum diameter as the recursion unfolds, and the use of depth calculations to facilitate this process. The time complexity is `O(N)` as we visit each node exactly once in the process.

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 diameterOfBinaryTree(self, root: TreeNode) -> int:
        # Helper function to perform depth-first search and calculate depth.
        def dfs(node):
            # If the node is None, then this is a leaf so we return 0.
            if node is None:
                return 0
            nonlocal max_diameter
            # Recursively find the depths of the left and right subtrees.
            left_depth = dfs(node.left)
            right_depth = dfs(node.right)
            # Update the maximum diameter found so far.
            # Diameter at this node will be the sum of depths of left & right subtrees.
            max_diameter = max(max_diameter, left_depth + right_depth)
            # Return the depth of this node which is max of left or right subtree depths plus 1.
            return 1 + max(left_depth, right_depth)

        # Initialize the maximum diameter as 0.
        max_diameter = 0
        # Start the DFS traversal from the root.
        dfs(root)
        # Finally, return the maximum diameter found.
        return max_diameter
```

Java Solution

```java
/**
 * Definition for a binary tree node.
 * This class represents a node in a binary tree, with a value and pointers to its 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;
    }
}

/**
 * This class contains methods to solve for the diameter of a binary tree.
 */
class Solution {
    private int maxDiameter; // Holds the maximum diameter found

    /**
     * Finds the diameter of a binary tree, which is the length of the longest path between any two nodes in a tree.
     * This path may or may not pass through the root.
     *
     * @param root the root node of the binary tree
     * @return the diameter of the binary tree
     */
    public int diameterOfBinaryTree(TreeNode root) {
        maxDiameter = 0;
        depthFirstSearch(root);
        return maxDiameter;
    }

    /**
     * A recursive method that calculates the depth of the tree and updates the maximum diameter.
     * The path length between the nodes is calculated as the sum of the heights of left and right subtrees.
     *
     * @param node the current node
     * @return the maximum height from the current node
     */
    private int depthFirstSearch(TreeNode node) {
        if (node == null) {
            // Base case: if the current node is null, return a height of 0
            return 0;
        }
        // Recursively find the height of the left and right subtrees
        int leftHeight = depthFirstSearch(node.left);
        int rightHeight = depthFirstSearch(node.right);

        // Update the maximum diameter if the sum of heights of the current node's subtrees is greater
        maxDiameter = Math.max(maxDiameter, leftHeight + rightHeight);

        // Return the max height seen up to the current node, including the current node's height (which is 1)
        return 1 + Math.max(leftHeight, rightHeight);
    }
}
```

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:
    // Class member to keep track of the maximum diameter found.
    int maxDiameter;

    // This function initializes the maxDiameter to zero and starts the DFS traversal.
    int diameterOfBinaryTree(TreeNode* root) {
        maxDiameter = 0;
        depthFirstSearch(root);
        return maxDiameter;
    }

    // Helper function for DFS traversal of the tree.
    // Calculates the depth of the tree and updates the maximum diameter.
    int depthFirstSearch(TreeNode* node) {
        if (node == nullptr) return 0; // Base case: return zero for null nodes.

        // Recursive DFS on the left child.
        int leftDepth = depthFirstSearch(node->left);
        // Recursive DFS on the right child.
        int rightDepth = depthFirstSearch(node->right);

        // Update the maximum diameter if the current node's diameter is greater.
        maxDiameter = max(maxDiameter, leftDepth + rightDepth);

        // Return the maximum depth from this node down to the leaf.
        return 1 + max(leftDepth, rightDepth);
    }
};
```

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);
  }
}

// Variable to hold the final result - the diameter of the binary tree.
let diameter: number = 0;

/**
 * A depth-first search function to traverse the tree and calculate
 * the diameter of the binary tree.
 *
 * @param {TreeNode | null} node - The current node being visited.
 * @returns {number} - The height of the current node.
 */
function depthFirstSearch(node: TreeNode | null): number {
  // Base case: if the node is null, return a height of 0.
  if (node === null) {
    return 0;
  }

  // Calculate the height of the left and right subtrees.
  const leftHeight = depthFirstSearch(node.left);
  const rightHeight = depthFirstSearch(node.right);

  // Update the diameter if the sum of left and right heights is larger than the current diameter.
  diameter = Math.max(diameter, leftHeight + rightHeight);

  // Return the height of the current node, which is max of left or right subtree height plus 1.
  return Math.max(leftHeight, rightHeight) + 1;
}

/**
 * Calculates the diameter of a binary tree - the length of the longest
 * path between any two nodes in a tree. This path may or may not pass through the root.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {number} - The diameter of the binary tree.
 */
function diameterOfBinaryTree(root: TreeNode | null): number {
  // Initialize diameter to 0 before starting DFS.
  diameter = 0;

  // Start DFS traversal from the root to calculate the diameter.
  depthFirstSearch(root);

  return diameter;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the `diameterOfBinaryTree` method is `O(n)`, where `n` is the number of nodes in the binary tree. This is because the auxiliary function `dfs` is called exactly once for each node in the tree. In the `dfs` function, the work done at each node is constant time, primarily consisting of calculating the maximum of the left and right heights and updating the `ans` variable if necessary.

Space Complexity

The space complexity of the `diameterOfBinaryTree` method is `O(h)`, where `h` is the height of the binary tree. This accounts for the maximum number of recursive calls that stack up on the call stack at any point during the execution of the `dfs` function. In the worst case, where the tree is skewed (forms a straight line), the height of the tree can be `n`, leading to a space complexity of `O(n)`. However, in a balanced tree, the height `h` would be `O(log n)`, leading to a more efficient space utilization.