333. Largest BST Subtree

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

Problem Description

The problem asks to find the largest Binary Search Tree (BST) within a given binary tree. A largest BST here refers to a BST with the maximum number of nodes. It's important to remember that a binary tree is not necessarily a BST. A BST is a specific type of binary tree where the left subtree contains only nodes with values less than the root's value, and the right subtree only nodes with values greater than the root's value. Each subtree must also follow these rules.

In this context, a "subtree" is considered any node and all its descendants, and we need to identify the largest one that fulfills the constraints of a BST within the original binary tree. The solution should return the number of nodes contained in this largest BST.

Intuition

This problem can be efficiently solved using a recursive depth-first search (DFS). The intuition is that for a subtree to be a BST, both its left and right children must also be BSTs, and the value of all nodes in the left subtree must be less than the root's value, and those in the right subtree must be greater.

The solution leverages a helper function, typically `dfs(root)`, which performs the following tasks:

- It checks the validity of BST for each node.
- It keeps track of the number of nodes in the valid subtree.
- It keeps track of the maximum and minimum value within the subtree to easily compare with its parent.

This recursive function returns a tuple `(min_value, max_value, number_of_nodes)` for each subtree analyzed. If the current subtree does not form a valid BST when considering its parent, it returns values that will cause the parent to also be invalid as a BST. This way, one does not need to further check subtrees of an already invalid BST, optimizing the process.

A key point for optimizing this algorithm is the early return when an invalid BST is found. Instead of checking all subtrees, the function stops early when it finds that the current subtree cannot possibly be a BST. This reduces the number of unnecessary checks and, therefore, the overall time complexity of the algorithm.

By maintaining a global or nonlocal variable that keeps track of the size of the largest BST found so far (`ans` in this case), we can keep updating the size whenever we find a larger valid BST.

At the end, the variable `ans` will hold the size of the largest found BST subtree, which we can return as the solution.

Solution Approach

The solution approach to finding the largest BST in a binary tree involves implementing a recursive Depth-First Search (DFS) that traverses the tree while performing checks and computations that ultimately reveal the largest BST subtree.

Here's a step-by-step breakdown of the `dfs(root)` function and how it integrates within the solution:

- A recursion base case is set. When the function encounters a `None` node (indicating an empty subtree), it returns `(inf, -inf, 0)`. This ensures that an empty subtree will not affect the validity of its parent.
- The `dfs` function is called recursively on the left child `dfs(root.left)` and the right child `dfs(root.right)` of the current `root`. Each call will return the minimum value, maximum value, and node count of the valid BST subtree rooted at that child.
- These values are used to determine if the current `root` with its subtrees can form a valid BST:
  - The maximum value in the left subtree (`lmx`) must be less than `root.val`.
  - The minimum value in the right subtree (`rmi`) must be greater than `root.val`.
- When the current node's value respects the BST property in relation to its children's values, the current subtree rooted at `root` is a valid BST. The node count is the sum of the node counts from the left and right subtrees plus one (the current `root` node).
- The global variable `ans` is updated with the maximum value between its current value and the newly found BST's node count.
- The function returns a tuple containing the new minimum value, maximum value, and the total node count of the valid BST. The minimum and maximum values are updated to ensure the parent node will be able to verify its BST properties correctly.
- If the current subtree does not form a valid BST, the function returns `(-inf, inf, 0)` to indicate an invalid subtree.
- Once the entire tree has been traversed, the accumulated value of `ans` will represent the largest number of nodes found in a BST within the tree.

The code uses Python's `nonlocal` keyword to modify the `ans` variable defined outside of the nested `dfs` function, which simplifies updating the count of the largest BST found during the recursive traversal.

In terms of data structures, the algorithm primarily operates on the binary tree itself and uses tuples to store and pass around multiple values compactly. The recursion stack implicitly created by the recursive calls is a critical part of the DFS traversal pattern applied here.

By systematically exploring each node and determining locally whether a valid BST exists at that node, then combining those local decisions to inform the parent node's validity, this approach effectively breaks down the problem into manageable subproblems adhering to the "divide and conquer" strategy, which is common in recursive algorithms.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose you have the following binary tree:

```
      10
     /  \
    5   15
   / \    \
  1   8    7
```

Each node contains a value, and we aim to find the largest BST within this binary tree.

1. We start with a DFS on the root node (10). We use the `dfs` function, which will return the minimum value, maximum value, and node count of the valid BST subtree.
2. We call `dfs` recursively on the left child (5) and right child (15).
3. For node 5, we call `dfs` recursively on node 1 (left child) and node 8 (right child).
   - Node 1 returns `(1, 1, 1)` because it has no children and is a valid BST on its own.
   - Node 8 returns `(8, 8, 1)` for the same reason.
   - Since the maximum value from the left `(1)` is less than 5, and the minimum value from the right `(8)` is greater than 5, node 5 with its children is a valid BST. We return `(1, 8, 3)` for node 5.
4. For node 15, we don't have a left child, and the right child is node 7.
   - Since 15 has no left child, we receive `(inf, -inf, 0)` from the left.
   - Node 7 returns `(7, 7, 1)` because it's a leaf node and hence a valid BST.
   - However, the minimum value from the right (7) is not greater than 15, so node 15 with its children cannot form a BST. Therefore, we return `(-inf, inf, 0)` for node 15, indicating it's an invalid subtree.
5. Now, back at the root node 10, we use the results from its children to determine whether it can form a BST. The left child returned `(1, 8, 3)` and the right child `(-inf, inf, 0)`.
   - The maximum value from the left (`8`) is less than 10, but we cannot use the results from the right child because it's invalid.
   - So, node 10 and its entire tree cannot form a valid BST. We return `(-inf, inf, 0)` for node 10.
6. With these recursive calls, we maintain a global variable `ans` that is updated every time a valid subtree BST is found. In our case, the largest BST is the subtree rooted at node 5 with 3 nodes.
7. Once we have completed the recursive DFS traversal, the accumulated value of `ans` will hold the size of the largest BST found within the binary tree, which is 3 in this example. We return this value as the solution to the problem.

Python Solution

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

from math import inf

class Solution:
    def largestBSTSubtree(self, root: Optional[TreeNode]) -> int:
        # Helper function to perform a DFS on the binary tree.
        def dfs(node):
            # Base case: If the node is None, return a tuple containing infinity,
            # negative infinity, and 0, indicating an empty subtree.
            if node is None:
                return inf, -inf, 0

            # Recursively obtain minimum value, maximum value, and size of the
            # left and right subtrees.
            left_min, left_max, left_size = dfs(node.left)
            right_min, right_max, right_size = dfs(node.right)

            # Use a nonlocal variable to keep track of the largest BST size found so far.
            nonlocal largest_bst_size

            # Check if the current tree rooted at 'node' is a BST by comparing its value
            # with the maximum of the left subtree and the minimum of the right subtree.
            if left_max < node.val < right_min:
                # Update the largest BST size if the current subtree is larger.
                largest_bst_size = max(largest_bst_size, left_size + right_size + 1)
                # Return a tuple with the new minimum, maximum, and size of the current subtree.
                return min(left_min, node.val), max(right_max, node.val), left_size + right_size + 1
            else:
                # If the current subtree is not a BST, return values that convey a violation
                # in the BST property to parent nodes.
                return -inf, inf, 0

        # Initialize the largest BST size to 0.
        largest_bst_size = 0
        # Perform a DFS on the binary tree starting from the root.
        dfs(root)
        # After the DFS is completed, return the largest BST size found.
        return largest_bst_size
```

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 {
    private int maxBSTSize;  // Variable to track the size of the largest BST.

    /**
     * Returns the size of the largest BST in a binary tree.
     *
     * @param root the root of the binary tree
     * @return the size of the largest BST
     */
    public int largestBSTSubtree(TreeNode root) {
        maxBSTSize = 0;
        dfs(root);
        return maxBSTSize;
    }

    /**
     * Performs a depth-first search to find the largest BST.
     *
     * @param node the current node
     * @return an array containing the minimum value, the maximum value,
     *         and the size of the largest BST in the subtree rooted at the current node
     */
    private int[] dfs(TreeNode node) {
        // Base case: if node is null, return "empty" values that won't affect parent node's calculation.
        if (node == null) {
            return new int[]{Integer.MAX_VALUE, Integer.MIN_VALUE, 0};
        }

        // Recursive DFS calls for left and right subtrees.
        int[] leftSubtree = dfs(node.left);
        int[] rightSubtree = dfs(node.right);

        // Check if current node's subtree is a BST.
        if (leftSubtree[1] < node.val && node.val < rightSubtree[0]) {
            // Calculate the size of this subtree (if it's a valid BST).
            int sizeOfCurrentBST = leftSubtree[2] + rightSubtree[2] + 1;

            // Update the answer if this is the largest BST so far.
            maxBSTSize = Math.max(maxBSTSize, sizeOfCurrentBST);

            // Return the updated information for this valid BST.
            return new int[]{
                Math.min(node.val, leftSubtree[0]), // Minimum value in the subtree.
                Math.max(node.val, rightSubtree[1]), // Maximum value in the subtree.
                sizeOfCurrentBST // Size of the subtree.
            };
        }
        // If the subtree with this node is not a BST, return "invalid" values.
        return new int[]{Integer.MIN_VALUE, Integer.MAX_VALUE, 0};
    }
}
```

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 largestSize; // Using "largestSize" to store largest BST subtree size.

    // Public member function to initiate the computation of largest BST subtree size.
    int largestBSTSubtree(TreeNode* root) {
        largestSize = 0;
        dfsHelper(root);
        return largestSize;
    }

    // Helper function to perform depth-first search and return essential info of each subtree.
    vector<int> dfsHelper(TreeNode* root) {
        // Base case: If current node is null, return maximum, minimum and size as specified.
        if (!root) return {INT_MAX, INT_MIN, 0};

        // Recursive cases for left and right subtrees.
        auto leftInfo = dfsHelper(root->left);
        auto rightInfo = dfsHelper(root->right);

        // If current subtree is BST, calculate subtree size and update "largestSize".
        if (leftInfo[1] < root->val && root->val < rightInfo[0]) {
            int subtreeSize = leftInfo[2] + rightInfo[2] + 1;
            largestSize = max(largestSize, subtreeSize);
            // Return a vector containing the minimum value, maximum value, and size of this BST.
            return {min(root->val, leftInfo[0]), max(root->val, rightInfo[1]), subtreeSize};
        }

        // If current subtree is not BST, return values that invalidate the parent node's BST status.
        return {INT_MIN, INT_MAX, 0};
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node in TypeScript.
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

// Variables and functions declared globally as per instructions.

// Variable to store largest BST subtree size.
let largestSize: number = 0;

/**
 * Initiates the computation of the largest BST subtree size.
 * @param root - The root node of the binary tree.
 * @returns The size of the largest BST subtree.
 */
function largestBSTSubtree(root: TreeNode | null): number {
    largestSize = 0;
    dfsHelper(root);
    return largestSize;
}

/**
 * Helper function to perform depth-first search and returns essential info of each subtree.
 * @param root - The current node of the binary tree.
 * @returns An array containing the minimum value, maximum value, and size of the subtree.
 */
function dfsHelper(root: TreeNode | null): [number, number, number] {
    // Base case: If current node is null, return maximum, minimum and size as specified.
    if (!root) return [Number.MAX_SAFE_INTEGER, Number.MIN_SAFE_INTEGER, 0];

    // Recursive cases for left and right subtrees.
    let leftInfo = dfsHelper(root.left);
    let rightInfo = dfsHelper(root.right);

    // If current subtree is BST, calculate subtree size and update "largestSize".
    if (leftInfo[1] < root.val && root.val < rightInfo[0]) {
        let subtreeSize = leftInfo[2] + rightInfo[2] + 1;
        largestSize = Math.max(largestSize, subtreeSize);
        // Return an array containing the min value, max value, and size of this BST.
        return [Math.min(root.val, leftInfo[0]), Math.max(root.val, rightInfo[1]), subtreeSize];
    }

    // If current subtree is not BST, return values that invalidate the parent node's BST status.
    return [Number.MIN_SAFE_INTEGER, Number.MAX_SAFE_INTEGER, 0];
}
```

Time and Space Complexity

The given code defines a function `largestBSTSubtree` that finds the size of the largest BST (Binary Search Tree) within a binary tree. Let's analyze its time and space complexity.

Time Complexity

The main computation in the code is done by the `dfs` (Depth-First Search) function, which is a recursive function that is called on every node of the tree exactly once. At each node, `dfs` performs a constant amount of work. It checks if the current subtree is a BST, and updates the minimum and maximum value seen so far along with the count of nodes in the subtree.

Thus, for a binary tree with `n` nodes, every node is visited once, resulting in a time complexity of `O(n)`.

Space Complexity

The space complexity of the code comes from two parts: the recursion stack and the nonlocal variable `ans`.

1. **Recursion Stack**: In the worst case, the recursion goes as deep as the height of the tree. For a balanced binary tree, the height is `log(n)`, leading to a space complexity of `O(log(n))` because of the recursion stack. However, in the worst case (a degenerate tree where each node has only one child), the height of the tree becomes `n`, which would result in a space complexity of `O(n)` for the recursion stack.
2. **Nonlocal Variable**: The nonlocal variable `ans` is used to track the maximum size of the BST found at any node. This only uses a constant amount of space, `O(1)`.

The overall space complexity is then the maximum space required by any part of the algorithm. Thus, the space complexity of the algorithm is `O(h)` where `h` is the height of the tree. In the worst case, this simplifies to `O(n)`.