333. Largest BST Subtree **Binary Search Tree** Tree Medium **Depth-First Search Dynamic Programming**

Leetcode Link

Binary Tree

Problem Description

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

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

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

constraints of a BST within the original binary tree. The solution should return the number of nodes contained in this largest BST.

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.

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

It keeps track of the maximum and minimum value within the subtree to easily compare with its parent.

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.

function stops early when it finds that the current subtree cannot possibly be a BST. This reduces the number of unnecessary

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

A recursion base case is set. When the function encounters a None node (indicating an empty subtree), it returns (inf, -inf,

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

traverses the tree while performing checks and computations that ultimately reveal the largest BST subtree.

0). This ensures that an empty subtree will not affect the validity of its parent.

• The maximum value in the left subtree (lmx) must be less than root.val.

updating the count of the largest BST found during the recursive traversal.

• The minimum value in the right subtree (rmi) must be greater than root.val.

• 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.

applied here.

• These values are used to determine if the current root with its subtrees can form a valid BST:

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).

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.

• The function returns a tuple containing the new minimum value, maximum value, and the total node count of the valid BST. The

• The global variable ans is updated with the maximum value between its current value and the newly found BST's node count.

When the current node's value respects the BST property in relation to its children's values, the current subtree rooted at root is

 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

- 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
- 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.

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

By systematically exploring each node and determining locally whether a valid BST exists at that node, then combining those local

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

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

○ 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.

Python Solution

2 # class TreeNode:

class Solution:

3 #

9

11

12

13

14

15

16

17

18

19

20

21

22

23

1 # Definition for a binary tree node.

self.val = val

def dfs(node):

else:

dfs(root)

Java Solution

class TreeNode {

int val;

TreeNode left;

TreeNode() {}

TreeNode right;

TreeNode(int val) {

this.val = val;

this.val = val;

1 /**

8

9

10

11

12

13

14

15

16

*/

largest_bst_size = 0

return largest_bst_size

* Definition for a binary tree node.

if node is None:

return inf, -inf, 0

left and right subtrees.

if left_max < node.val < right_min:</pre>

return -inf, inf, 0

Initialize the largest BST size to 0.

TreeNode(int val, TreeNode left, TreeNode right) {

node count of the valid BST subtree.

2. We call dfs recursively on the left child (5) and right child (15).

Example Walkthrough

 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.

Node 7 returns (7, 7, 1) because it's a leaf node and hence a valid BST.

3. For node 5, we call dfs recursively on node 1 (left child) and node 8 (right child).

with its children is a valid BST. We return (1, 8, 3) for node 5.

Since 15 has no left child, we receive (inf, -inf, 0) from the left.

4. For node 15, we don't have a left child, and the right child is node 7.

the largest BST is the subtree rooted at node 5 with 3 nodes.

def __init__(self, val=0, left=None, right=None):

def largestBSTSubtree(self, root: Optional[TreeNode]) -> int:

Helper function to perform a DFS on the binary tree.

left_min, left_max, left_size = dfs(node.left)

in the BST property to parent nodes.

Perform a DFS on the binary tree starting from the root.

After the DFS is completed, return the largest BST size found.

right_min, right_max, right_size = dfs(node.right)

negative infinity, and 0, indicating an empty subtree.

Base case: If the node is None, return a tuple containing infinity,

Recursively obtain minimum value, maximum value, and size of the

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.

If the current subtree is not a BST, return values that convey a violation

return min(left_min, node.val), max(right_max, node.val), left_size + right_size + 1

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.

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,

7. Once we have completed the recursive DFS traversal, the accumulated value of ans will hold the size of the largest BST found

○ So, node 10 and its entire tree cannot form a valid BST. We return (-inf, inf, 0) for node 10.

within the binary tree, which is 3 in this example. We return this value as the solution to the problem.

- self.left = left self.right = right 6 # from math import inf
- 24 # Use a nonlocal variable to keep track of the largest BST size found so far. 25 nonlocal largest_bst_size 26 # 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.

```
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
```

this.left = left; 17 18 this.right = right; 19 20 } 21

```
class Solution {
 23
         private int maxBSTSize; // Variable to track the size of the largest BST.
 24
 25
         /**
 26
          * Returns the size of the largest BST in a binary tree.
 27
          * @param root the root of the binary tree
 28
          * @return the size of the largest BST
 29
 30
 31
         public int largestBSTSubtree(TreeNode root) {
 32
             maxBSTSize = 0;
 33
             dfs(root);
 34
             return maxBSTSize;
 35
 36
 37
         /**
 38
          * Performs a depth-first search to find the largest BST.
 39
 40
          * @param node the current node
          * @return an array containing the minimum value, the maximum value,
 41
 42
                    and the size of the largest BST in the subtree rooted at the current node
 43
         private int[] dfs(TreeNode node) {
 44
 45
             // Base case: if node is null, return "empty" values that won't affect parent node's calculation.
 46
             if (node == null) {
                 return new int[]{Integer.MAX_VALUE, Integer.MIN_VALUE, 0};
 47
 48
 49
 50
             // Recursive DFS calls for left and right subtrees.
 51
             int[] leftSubtree = dfs(node.left);
 52
             int[] rightSubtree = dfs(node.right);
 53
 54
             // Check if current node's subtree is a BST.
 55
             if (leftSubtree[1] < node.val && node.val < rightSubtree[0]) {</pre>
 56
                 // Calculate the size of this subtree (if it's a valid BST).
 57
                 int sizeOfCurrentBST = leftSubtree[2] + rightSubtree[2] + 1;
 58
 59
                 // Update the answer if this is the largest BST so far.
                 maxBSTSize = Math.max(maxBSTSize, sizeOfCurrentBST);
 60
 61
 62
                 // Return the updated information for this valid BST.
                 return new int[]{
 63
                     Math.min(node.val, leftSubtree[0]), // Minimum value in the subtree.
 64
                     Math.max(node.val, rightSubtree[1]), // Maximum value in the subtree.
 65
 66
                     sizeOfCurrentBST // Size of the subtree.
 67
                 };
 68
 69
             // If the subtree with this node is not a BST, return "invalid" values.
             return new int[]{Integer.MIN VALUE, Integer.MAX VALUE, 0};
 70
 71
 72 }
 73
C++ Solution
  1 /**
     * Definition for a binary tree node.
      */
     struct TreeNode {
         int val;
        TreeNode *left;
```

// Definition for a binary tree node in TypeScript. interface TreeNode { val: number;

Typescript Solution

left: TreeNode | null;

TreeNode *right;

8

9

11 };

14 public:

13 class Solution {

10

12

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

};

TreeNode() : val(0), left(nullptr), right(nullptr) {}

int largestBSTSubtree(TreeNode* root) {

vector<int> dfsHelper(TreeNode* root) {

return {INT_MIN, INT_MAX, 0};

if (!root) return {INT_MAX, INT_MIN, 0};

auto leftInfo = dfsHelper(root->left);

auto rightInfo = dfsHelper(root->right);

// Recursive cases for left and right subtrees.

if (leftInfo[1] < root->val && root->val < rightInfo[0]) {</pre>

int subtreeSize = leftInfo[2] + rightInfo[2] + 1;

largestSize = max(largestSize, subtreeSize);

largestSize = 0;

dfsHelper(root);

return largestSize;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

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

// Public member function to initiate the computation of largest BST subtree size.

// Helper function to perform depth-first search and return essential info of each subtree.

// If current subtree is BST, calculate subtree size and update "largestSize".

// Base case: If current node is null, return maximum, minimum and size as specified.

// Return a vector containing the minimum value, maximum value, and size of this BST.

// If current subtree is not BST, return values that invalidate the parent node's BST status.

return {min(root->val, leftInfo[0]), max(root->val, rightInfo[1]), subtreeSize};

```
right: TreeNode | null;
 6 }
    // Variables and functions declared globally as per instructions.
   // Variable to store largest BST subtree size.
   let largestSize: number = 0;
12
13
    * 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.
17
    */
   function largestBSTSubtree(root: TreeNode | null): number {
        largestSize = 0;
19
20
       dfsHelper(root);
       return largestSize;
21
22 }
23
24
   /**
    * 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.
28
    */
   function dfsHelper(root: TreeNode | null): [number, number, number] {
29
       // 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];
31
32
33
       // Recursive cases for left and right subtrees.
       let leftInfo = dfsHelper(root.left);
34
       let rightInfo = dfsHelper(root.right);
35
36
37
       // If current subtree is BST, calculate subtree size and update "largestSize".
38
       if (leftInfo[1] < root.val && root.val < rightInfo[0]) {</pre>
           let subtreeSize = leftInfo[2] + rightInfo[2] + 1;
39
           largestSize = Math.max(largestSize, subtreeSize);
40
           // Return an array containing the min value, max value, and size of this BST.
41
           return [Math.min(root.val, leftInfo[0]), Math.max(root.val, rightInfo[1]), subtreeSize];
43
44
       // If current subtree is not BST, return values that invalidate the parent node's BST status.
45
       return [Number.MIN_SAFE_INTEGER, Number.MAX_SAFE_INTEGER, 0];
46
47 }
48
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
```

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

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

The main computation in the code is done by the dfs (Depth-First Search) function, which is a recursive function that is called on

and updates the minimum and maximum value seen so far along with the count of nodes in the subtree.

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

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,

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.

constant amount of space, 0(1). The overall space complexity is then the maximum space required by any part of the algorithm. Thus, the space complexity of the

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

algorithm is O(h) where h is the height of the tree. In the worst case, this simplifies to O(n).