1373. Maximum Sum BST in Binary Tree Binary Tree Depth-First Search | Binary Search Tree | Dynamic Programming Hard

### Leetcode Link

## The problem is about finding the maximum sum of all keys within any subtree of a binary tree, with the constraint that the subtree

Problem Description

must also be a Binary Search Tree (BST). The definition of a BST here is: The left subtree of a node contains only nodes with keys less than the node's key.

Both the left and right subtrees themselves must also be BSTs.

The right subtree of a node contains only nodes with keys greater than the node's key.

- Our goal is to identify such subtrees that are BSTs within the larger binary tree and calculate their sums, finally returning the maximum one.

Intuition

# behind the solution:

Using tuple as return value: Each recursive call returns a tuple with details about the subtree:

are BSTs and to calculate their respective sums, along with the minimum and maximum values within those subtrees.

To find the solution, we adopt a recursive approach that performs a depth-first search (DFS) on the binary tree. Here's the intuition

• Recursive DFS with BST validation: At each node, we perform a DFS to recursively validate whether the left and right subtrees

- 1. A boolean indicating whether the subtree is a BST. 2. The minimum value within the subtree (for BST validation). 3. The maximum value within the subtree (for BST validation).
- Local and global results: Within each recursion, we maintain a local sum of the subtree if it's a valid BST. We also maintain a global variable ans to track the maximum sum encountered among all BST subtrees.

4. The sum of the keys of nodes within the subtree.

- Validating and updating BST properties: If the left and right subtrees of a given node are BSTs and the maximum key in the left
  - is less than the current node's key while the current node's key is less than the minimum key in the right, the tree rooted at the current node is a BST. We then update the sum and the boundary values (minimum and maximum keys) for this subtree.

updated in the global variable ans. Finally, we return ans as the result.

step-by-step explanation of the critical components of the reference solution:

a valid empty BST with a sum of zero and safe boundary values. If a subtree is not a valid BST, we return a tuple that will fail the BST check for the parent call.

Throughout this process, we bubble up valid BST information while capturing their sum, and ensure the maximum sum found is

• Base case and invalid BST handling: When we reach a null child (a leaf node's child), we return a tuple that inherently suggests

The provided implementation effectively traverses the tree while checking and validating the BST properties and computes the sum of the subtree keys until the maximum sum of any BST subtree within the overall tree is identified.

Solution Approach The solution provided uses a Depth-First Search (DFS) approach to navigate through each node in the tree and applies the logic to identify valid BST subtrees and calculate their sums. The approach relies significantly on recursion and tuple unpacking. Below is a

 Data Structures: The primary data structure used here is the binary tree itself, traversed using recursion. The tuple is also used to return multiple values from the recursive calls. Base Case: When we encounter a None node, which is the case for a leaf node's child, we return a tuple (1, inf, -inf, 0)

lbst, lmi, lmx, ls for the left subtree and rbst, rmi, rmx, rs for the right subtree. Here:

The left and right subtrees must both be BSTs (lbst and rbst must be True).

meet BST requirements and providing non-useful boundary values and sum.

We start by calling the recursive function dfs() at the root node with the value 5.

lbst and rbst store whether the left and right subtrees are BSTs (1 for True, 0 for False).

• lmi and rmi are the minimum values found in the left and right subtrees, respectively.

representing a valid BST with infinite boundaries (since a leaf node's child can be considered an empty BST) and a sum of 0.

## Recursive Case: For any non-null node, we perform the following: Recursively call dfs(root.left) and dfs(root.right) to check the left and right subtrees, unpacking the returned tuple into

this are:

the current subtree.

proportional to the number of nodes in the tree.

Recursively call dfs() on the left child node 3.

2. Next, we call dfs() on the right child node 8.

-inf, 7, 15) for node 8.

Hence, dfs(5) would return (1, 2, 7, 29).

# Standardized and commented Python3 code for the problem

def \_\_init\_\_(self, val=0, left=None, right=None):

# Helper function to perform depth-first search.

max\_sum = max(max\_sum, subtree\_sum)

dfs(root) # Start the DFS from the root of the tree.

return max\_sum # Return the maximum sum.

TreeNode(int val) { this.val = val; }

public int maxSumBST(TreeNode root) {

private int[] dfs(TreeNode node) {

if (node == null) {

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

private int maxSum; // Holds the maximum sum of all BSTs found so far

private static final int INFINITY = 1 << 30; // Representing infinity as large number

// Recursive DFS function that returns an array with information about the BST

return True, float('inf'), float('-inf'), 0

left\_is\_bst, left\_min, left\_max, left\_sum = dfs(node.left)

max\_sum = 0 # Initialize the maximum sum of all BSTs in the tree.

right\_is\_bst, right\_min, right\_max, right\_sum = dfs(node.right)

def maxSumBST(self, root: TreeNode) -> int:

def dfs(node: TreeNode) -> tuple:

return False, 0, 0, 0

\* Definition for a binary tree node.

this.val = val;

\* public class TreeNode {

TreeNode left;

TreeNode right;

TreeNode() {}

dfs(root);

return maxSum;

int val;

if node is None:

# Represents a node in a binary tree.

self.val = val

self.left = left

self.right = right

For node 3, we call dfs() on its left child 2:

Next, we call dfs() on the right child 4 of node 3:

Node 4 is also a leaf, so dfs(4) returns (1, 4, 4, 4).

Imx and rmx are the maximum values found in the left and right subtrees, respectively. • 1s and rs are the sum of all nodes in the left and right subtrees, respectively. With the above information, we check whether the current node with its subtrees can form a valid BST. The conditions for

 Calculating Sum and Update Maximum Sum: If the current subtree rooted at root is a valid BST, we calculate the sum s by adding the node's value root.val to the sums of the left and right subtrees (ls + rs). We then use a global variable ans to track

the maximum sum encountered, updating it with s if s is greater than ans. We also update the range of values (min and max) for

• Invalid BST Handling: If the current subtree rooted at root is not a valid BST, we return (0, 0, 0, 0) indicating the failure to

Finally, we initiate the recursion by calling dfs (root) and return the global maximum sum ans found among all valid BST subtrees in

The maximum value in the left subtree (lmx) must be less than the current node's value (root.val).

The current node's value must be less than the minimum value in the right subtree (rmi).

the binary tree. The implementation efficiently ensures that each subtree is only traversed once and all necessary checks and calculations are

performed during this single traversal. As a result, the algorithm is relatively efficient with a time complexity that is linearly

Example Walkthrough Let's walk through a small example to illustrate the solution approach using the recursive Depth-First Search (DFS) with BST validation explained in the problem description. Suppose we have the following binary tree:

We need to find the maximum sum of keys of any subtree that is also a Binary Search Tree (BST).

■ Node 2 is a leaf, so dfs(2) returns (1, 2, 2, 2) (it's a valid BST with itself being the only node, min = 2, max = 2, sum =

The sum for the subtree rooted at 3 is 3 (node value) + 2 (left sum) + 4 (right sum) = 9, so the function returns (1, 2,

Since the left "empty subtree" and the right subtree rooted at 7 are valid BSTs and their max/min values respectively satisfy

The sum for the subtree rooted at 8 is 8 (node value) + 0 (left sum) + 7 (right sum) = 15, and the function returns (1,

Node 8 has no left child, so the left call returns (1, inf, -inf, 0), indicating an empty tree which is trivially a BST.

 Since both children of node 3 are valid BSTs and their max/min values respectively satisfy the BST properties, node 3 with its children is also a valid BST.

class TreeNode:

class Solution:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

10

17

18

19

20

21

22

23

24

25

26

27

\*

4, 9) for node 3.

2).

 We see that the values satisfy the BST properties because 4 < 5 < inf.</li> ○ The sum of keys of this whole subtree is 5 (node value) + 9 (left sum) + 15 (right sum) = 29.

Node 8 has a right child 7. Since 7 is a leaf node, dfs(7) returns (1, 7, 7, 7).

the BST properties with respect to node 8, the whole subtree rooted at 8 is a BST.

3. With the results from dfs(3) and dfs(8), we now decide if the whole tree rooted at 5 is a BST.

find the maximum BST subtree sum. Python Solution

Among all the subtrees we checked, the one with the maximum sum is the whole tree itself with a sum of 29. Therefore, the function

This example illustrates the process of recursively exploring the tree, validating BST properties at each node, and combining sums to

dfs() initiated on the root 5 ultimately returns a maximum sum ans of 29, which is the result that will be returned by our algorithm.

- # Check if the current subtree is a binary search tree (BST). if left\_is\_bst and right\_is\_bst and left\_max < node.val < right\_min:</pre> # Update the maximum sum if the current subtree is a BST. subtree\_sum = left\_sum + right\_sum + node.val nonlocal max\_sum
- Java Solution

# Return a tuple representing (is BST, minimum value, maximum value, sum).

# Return a tuple (is BST, new min value, new max value, subtree sum).

return True, min(left\_min, node.val), max(right\_max, node.val), subtree\_sum

# Current subtree is not a BST, return tuple with flags set to False/zero values.

#### this.left = left; 11 12 this.right = right; 13 14 \* } 15 \*/ class Solution { 16

```
28
                 // Returns an array with the structure [isBST, minVal, maxVal, sum]
                 // For a null node, isBST=1 (true), minVal=INFINITY, maxVal=-INFINITY, sum=0
 29
                 return new int[] {1, INFINITY, -INFINITY, 0};
 30
 31
 32
 33
             // Explore left subtree
 34
             int[] leftSubtree = dfs(node.left);
 35
             // Explore right subtree
             int[] rightSubtree = dfs(node.right);
 36
 37
 38
             // Value of current node
 39
             int value = node.val;
 40
 41
             // Checks if both left and right subtrees are BST and values fall in the correct range
 42
             if (leftSubtree[0] == 1 && rightSubtree[0] == 1 && leftSubtree[2] < value && rightSubtree[1] > value) {
 43
                 // Calculate sum of the BST rooted at current node
 44
                 int sum = value + leftSubtree[3] + rightSubtree[3];
 45
 46
                 // Update the global maxSum if the current sum is greater
 47
                 maxSum = Math.max(maxSum, sum);
 48
 49
                 // Return [isBST, minVal, maxVal, sum] for the current subtree
                 return new int[] {1, Math.min(leftSubtree[1], value), Math.max(rightSubtree[2], value), sum};
 50
 51
 52
 53
             // Return array indicating current subtree is not BST
 54
             // By default, isBST=0 indicating it's not a BST
 55
             return new int[4];
 56
 57 }
 58
C++ Solution
  1 // Definition for a binary tree node.
  2 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) {}
  9 };
 10
 11 class Solution {
 12 public:
         int maxSumBST(TreeNode* root) {
 13
             int maxSum = 0; // This variable will hold the maximum sum of any BST found in the tree.
 14
 15
             const int INF = 1 << 30; // A large enough value to represent infinity in this context.
 16
 17
             // This is a post-order traversal DFS lambda function that will explore each node
 18
             // to check if the subtree rooted at that node is a BST and then calculate the
 19
             // sum of all nodes in that BST.
             function<vector<int>(TreeNode*)> dfs = [&](TreeNode* node) -> vector<int> {
 20
 21
                 if (!node) {
```

// Base case: if the node is empty, return a vector indicating

return vector<int>{1, INF, -INF, 0};

// the subtree rooted at the current node is also a BST.

// If it is not a valid BST, return vector with default values.

constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {

auto left = dfs(node->left);

auto right = dfs(node->right);

// and the sum of the BST.

return vector<int>(4);

// Return the maximum sum found.

this.val = (val === undefined ? 0 : val);

this.left = (left === undefined ? null : left);

// that this is a valid BST with min value INF, max value —INF, and sum 0.

// value in the left subtree and less than the min value in the right subtree,

if (left[0] && right[0] && left[2] < node->val && node->val < right[1]) {</pre>

// Perform DFS starting from the root to find the max sum of any BST in the tree.

// If the left and right subtrees are BSTs, and the current node's value is greater than the max

maxSum = std::max(maxSum, sum); // Update max sum if this BST is the largest found so far.

// Return a vector indicating that this is a valid BST, with the new min and max values,

return vector<int>{1, std::min(left[1], node->val), std::max(right[2], node->val), sum};

int sum = left[3] + right[3] + node->val; // Calculate the sum of nodes in this BST.

#### Typescript Solution 1 // Definition for a binary tree node. 2 class TreeNode {

val: number

left: TreeNode | null

right: TreeNode | null

**}**;

dfs(root);

return maxSum;

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

53

8

9

52 };

```
this.right = (right === undefined ? null : right);
 10
 11 }
 12
    // Holds the maximum sum of all BSTs in the tree.
 14 let maxSum = 0;
 15
    // Helper function which performs a depth-first search to find the maximum sum BST.
    function dfs(node: TreeNode | null): [boolean, number, number, number] {
 18
         const infinity = 1 << 30; // Using a large number to represent infinity.</pre>
 19
 20
         // Base case: if the current node is null, return a tuple representing an empty BST.
 21
         if (!node) {
 22
             return [true, infinity, -infinity, 0];
 23
 24
 25
         // Recursive calls for left and right subtrees.
 26
         const [isLeftBST, leftMin, leftMax, leftSum] = dfs(node.left);
 27
         const [isRightBST, rightMin, rightMax, rightSum] = dfs(node.right);
 28
 29
         // A valid BST must satisfy: the max of left child < current node's value < min of right child.
 30
         if (isLeftBST && isRightBST && leftMax < node.val && node.val < rightMin) {</pre>
 31
             const sum = leftSum + rightSum + node.val;
 32
             maxSum = Math.max(maxSum, sum); // Update maxSum if a new max is found.
 33
             // Return a tuple representing a valid BST with updated minimum, maximum, and sum.
 34
             return [true, Math.min(leftMin, node.val), Math.max(rightMax, node.val), sum];
 35
 36
 37
         // If the subtree is not a BST, return a tuple indicating false without valid min, max, and sum.
 38
         return [false, 0, 0, 0];
 39 }
 40
 41 // Main function to call dfs and return the maxSum found.
 42 // This function checks the entire tree to find the maximum sum of any BST within it.
    function maxSumBST(root: TreeNode | null): number {
         dfs(root);
 44
         return maxSum;
Time and Space Complexity
The time complexity of the given code is O(n), where n is the number of nodes in the tree. The code performs a single post-order
traversal of the tree, visiting each node exactly once. During each visit, the function calculates and returns a tuple containing
information about whether the subtree is a Binary Search Tree, the minimum and maximum values in the subtree, and the sum of
```

the worst case, we present the space complexity as O(n).

values in the subtree if it is a BST. Since these operations are all done in constant time for each node, the time complexity of the entire algorithm is linear in the number of nodes. The space complexity of the solution is also 0(n) in the worst case. The worst-case space complexity occurs when the tree is skewed (having only left children or only right children), which results in a recursion depth equal to the number of nodes in the tree. Thus, the system call stack will at most have n activation records corresponding to the recursive calls. In a balanced tree, the space complexity would be O(log n) due to the height of the call stack corresponding to the tree height, but since we have to account for