# 938. Range Sum of BST

Easy Array Depth-First Search Binary Search Tree Binary Tree

Leetcode Link

## Problem Description

The problem provides us with the root node of a binary search tree (BST) and two integer values, `low` and `high`. It asks us to calculate the sum of all the node values in the BST that fall within the inclusive range `[low, high]`. This means that if a node's value is equal to or greater than `low` and equal to or less than `high`, it should be included in our sum. The binary search property of the tree, where left child nodes are less than the parent node and right child nodes are greater, can be used to optimize our search for the range.

## Intuition

The intuition behind the solution approach is to perform a depth-first traversal of the BST while leveraging the BST property to avoid unnecessary traversal. The steps of the approach are:

1. If the current node is null (we've reached a leaf's child), we can stop traversing this path.
2. If the current node's value falls within `[low, high]`, we add it to the sum and continue to search both the left and right subtrees as there may be more nodes within the range.
3. If the current node's value is less than `low`, we only need to traverse the right subtree because all values in the left subtree will also be less than `low`.
4. If the current node's value is greater than `high`, we only need to traverse the left subtree, as all values in the right subtree will be greater than `high`.

Using these rules, the search efficiently skips parts of the tree that don't contribute to the range sum, which optimizes our algorithm significantly compared to a naive approach that checks every node.

## Solution Approach

The implementation of the solution involves the use of a recursive helper function called `search`, nested within the main function `rangeSumBST`. This is a common design pattern in recursive solutions, allowing the use of helper functions with additional parameters without changing the main function's signature. The algorithm uses this recursive function to perform a modified in-order traversal of the binary search tree. Here's a breakdown of how the algorithm works, with references to the data structures and patterns used:

1. **Initialization**: A variable `self.ans` is initialized to `0`. This will accumulate the sum of the node values that are within the range `[low, high]`.
2. **Recursive Function (`search`)**: A nested function `search` is defined, which takes the current node as its parameter. The use of recursion is key here, enabling the function to traverse the tree depth-first.
3. **Base Case**: At the beginning of the `search` function, there is a check to see if the current node is `None`. If it is, the function returns immediately. This check acts as the base case of the recursion, terminating the traversal at leaf nodes.
4. **Range Check**:
   - If the current node's value is within the range `[low, high]` (inclusive), the value is added to `self.ans`. As this node is within the range, there might be other nodes within the range both to the left and right, so the function recursively calls itself on both `node.left` and `node.right`.
5. **BST Property Utilization**:
   - If the current node's value is less than `low`, the function calls itself recursively on `node.right` as all left child nodes of the current node are guaranteed to be out of the range.
   - If the current node's value is greater than `high`, the function calls itself recursively on `node.left` as all right child nodes of the current node are guaranteed to be out of the range.
6. **Starting the Traversal**: The recursive `search` function is first called on the root node to start the depth-first traversal.
7. **Returning the Answer**: Finally, after the recursive calls have completed, `self.ans` contains the sum of the values within the range, and this value is returned by the `rangeSumBST` function.

The use of a BST's intrinsic properties allows the algorithm to avoid inspecting every single node in the tree, thereby significantly improving efficiency, especially in cases where the range bounds are far from the minimum and maximum values in the tree. The space complexity is proportional to the height of the tree (O(H)) due to the recursion stack, and the time complexity is O(N) in the worst case when the tree is unbalanced, but it could be much less if the tree is balanced and the search prunes branches effectively.

### Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the following binary search tree (BST) and the range [low, high] = [10, 22]:

```
      15
     /  \
    10   20
   /  \
  8   13   25
 /
7
```

Now let's walk through how the solution would process this tree:

1. The `rangeSumBST` function initializes `self.ans` to `0`. This will hold the sum of values within the given range.
2. The `search` function is called with the root node (15). Since 15 is within the range [10, 22], it is added to `self.ans` (0 + 15 = 15), and the `search` function will be called on both its left child (10) and right child (20).
3. The `search` function is now called with node 10. Since 10 is also within the range, we add it to `self.ans` (15 + 10 = 25), and call `search` on both of its children (5 and 13).
4. When called with node 5, the function does not add its value to `self.ans` since 5 is outside the range. As 5 is less than `low` (10), we do not call `search` on its left subtree, and there is no right subtree to explore.
5. The `search` function is next called with node 13, which falls within the range. Therefore, 13 is added to `self.ans` (25 + 13 = 38), and since it has a right child (11) that could be within the range, `search` is called on node 11.
6. Node 11 is within the range, so its value is added to `self.ans` (38 + 11 = 49). Node 11 has no children, so no further calls are made here.
7. Back to the root, the `search` function is called with the node 20. This node's value is within the range and therefore is sum-up to `self.ans` (49 + 20 = 69). Since there is a right child (25) and it is potentially within the range, we call `search` on it.
8. Node 25 is greater than `high` (22), so we don't add it to `self.ans`. Also, since its value is greater than `high`, we do not explore its right subtree (not existing in this case), and because it has no left child, the traversal of this path ends.

With all paths explored, the final sum calculated is `self.ans` = 69, and the `rangeSumBST` returns this sum as the answer.

This example demonstrates how the algorithm effectively uses the BST properties to skip over irrelevant parts of the tree (for instance, not exploring left subtrees of nodes with values less than `low`, and right subtrees of nodes with values greater than `high`), thus optimizing the search for efficiency.

## 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

class Solution:
    def rangeSumBST(self, root: TreeNode, low: int, high: int) -> int:
        # Helper function to perform DFS and accumulate node values within the range [low, high]
        def dfs(node):
            # If the current node is None, return and do nothing
            if not node:
                return

            # If the current node's value is within the range [low, high], add it to the total sum
            if low <= node.val <= high:
                self.total_sum += node.val
                # Continue searching to the left and right as there may still be values
                # within the range [low, high]
                dfs(node.left)
                dfs(node.right)
            elif node.val < low:
                # If the current node's value is less than the low value of the range
                # then only search to the right as all left values will also be less
                dfs(node.right)
            elif node.val > high:
                # If the current node's value is greater than the high value of the range
                # then only search to the left as all right values will also be greater
                dfs(node.left)

        # Initialize the total_sum as 0 before starting the DFS
        self.total_sum = 0
        # Start the depth-first search from the root node
        dfs(root)
        # After DFS, return the total sum of values within [low, high]
        return self.total_sum
```

## Java Solution

```java
// Class containing the solution method to calculate the sum of values of all nodes within a range of a BST
class Solution {
    // This method computes the sum of values falling within the [low, high] range in a BST
    public int rangeSumBST(TreeNode root, int low, int high) {
        // If the current node is null, return 0 since there are no values to add
        if (root == null) {
            return 0;
        }

        // If the value of the current node is within the [low, high] range,
        // include its value in the sum, and continue searching in both left and right subtrees
        if (low <= root.val && root.val <= high) {
            return root.val + rangeSumBST(root.left, low, high) + rangeSumBST(root.right, low, high);
        }
        // If the value of the current node is less than the lower bound,
        // skip the left subtree because all values there will be out of range,
        // and only continue searching in the right subtree
        else if (root.val < low) {
            return rangeSumBST(root.right, low, high);
        }
        // If the value of the current node is greater than the upper bound,
        // skip the right subtree because all values there will be out of range,
        // and only continue searching in the left subtree
        else {
            return rangeSumBST(root.left, low, high);
        }
    }
}

// Definition for a binary tree node.
class TreeNode {
    int val;    // The integer value of the node
    TreeNode left;  // Reference to the left child
    TreeNode right;  // Reference to the right child

    // Constructor for creating a new node without children
    TreeNode() {}

    // Constructor for creating a new node with a given value
    TreeNode(int val) {
        this.val = val;
    }

    // Constructor for creating a new node with a given value and references to left and right children
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

## 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:
    /**
     * Computes the sum of values of all nodes with a value in the range [low, high].
     * Utilizes the BST property to prune branches that cannot contain values in the range.
     *
     * @param root Pointer to the root of the binary search tree (BST).
     * @param low The lower bound of the range (inclusive).
     * @param high The upper bound of the range (inclusive).
     * @return The sum of values in the specified range.
     */
    int rangeSumBST(TreeNode* root, int low, int high) {
        if (root == nullptr) {
            // Base case: if the current node is null, return sum of 0
            return 0;
        }

        // Check if the current node's value is within the range [low, high]
        if (low <= root->val && root->val <= high) {
            // If it is in the range, include the node's value in the sum,
            // and continue to check both left and right children
            return root->val + rangeSumBST(root->left, low, high) + rangeSumBST(root->right, low, high);
        } else if (root->val < low) {
            // If current node's value is less than low, no need to check
            // the left subtree as all values will be out of range; proceed
            // to check the right child
            return rangeSumBST(root->right, low, high);
        } else {
            // If current node's value is greater than high, no need to check
            // the right subtree as all values will be out of range; proceed
            // to check the left child
            return rangeSumBST(root->left, low, high);
        }
    }
};
```

## Typescript Solution

```typescript
// Typescript representation of a binary search tree node
class TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Computes the sum of values of all nodes with a value in the range [low, high].
 * Utilizes the Binary Search Tree (BST) property to prune branches that cannot contain values in the range.
 *
 * @param root The root of the binary search tree (BST).
 * @param low The lower bound of the range (inclusive).
 * @param high The upper bound of the range (inclusive).
 * @returns The sum of values within the specified range.
 */
function rangeSumBST(root: TreeNode | null, low: number, high: number): number {
  if (!root) {
    // Base case: if the current node is null, return a sum of 0.
    return 0;
  }

  // Check if the current node's value is within the range [low, high].
  if (root.val >= low && root.val <= high) {
    // If the current node's value is within the range, include the node's value in the sum,
    // and continue to check both the left and right subtrees.
    return root.val + rangeSumBST(root.left, low, high) + rangeSumBST(root.right, low, high);
  } else if (root.val < low) {
    // The current node's value is less than the low end of the range,
    // so there is no need to check the left subtree (all values will be too small).
    // Instead, traverse the right subtree only.
    return rangeSumBST(root.right, low, high);
  } else {
    // The current node's value is greater than the high end of the range,
    // so there is no need to check the right subtree (all values will be too large).
    // Instead, traverse the left subtree only.
    return rangeSumBST(root.left, low, high);
  }
}

export {};
```

## Time and Space Complexity

The given Python code defines a function `rangeSumBST` that calculates the sum of values of all nodes with a value in the range `[low, high]` within a binary search tree (BST). The function implements a recursive depth-first search strategy.

### Time Complexity

The time complexity of the function is `O(N)`, where `N` is the number of nodes in the binary search tree. This is because, in the worst-case scenario, the function may have to visit every node in the tree. However, due to the properties of a binary search tree and the pruning of branches that do not fall within the `[low, high]` range, the average case will likely be more efficient than this. Nonetheless, the upper bound worst-case remains `O(N)`.

### Space Complexity

The space complexity of the recursion is `O(H)`, where `H` is the height of the tree. This complexity arises from the call stack that holds the recursive calls during the search process. In the worst-case scenario of a skewed binary search tree where the tree behaves like a linked list, the height of the tree could be `N`, making the space complexity `O(N)`. For a balanced tree, the height of the tree would be `log(N)`, leading to a space complexity of `O(log(N))`. It should be noted that this does not take into account the space used to store the binary search tree itself, only the additional space used by the recursion call stack.