95. Unique Binary Search Trees II

Medium · Tree · Binary Search Tree · Dynamic Programming · Backtracking · Binary Tree · Leetcode Link

Problem Description

The problem asks us to generate all structurally unique binary search trees (BSTs) that can be constructed with `n` uniquely valued nodes, where each node's value is a unique integer from 1 to `n`. Structurally unique means that the trees' shape or structure must be different; simply rearranging nodes without changing the parents and children does not count as unique. The goal is not only to count these trees but to actually construct each distinct tree and return a collection of them.

Intuition

The intuition behind the solution for generating all structurally unique BSTs for `n` unique values involves understanding the properties of BSTs and using recursion to build all possible trees. The main property of a BST is that the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree only contains nodes with keys greater than the node's key.

Here's how we can arrive at the solution approach:

1. We know that the root can be any node with a value from 1 to `n`.
2. Once we select a value `i` for the root, all values from 1 to `i-1` must be in the left subtree, and all values from `i+1` to `n` must be in the right subtree, due to the properties of BSTs.
3. To generate all unique left and right subtrees, we can use recursion on the respective ranges. The recursive call will generate all possible left subtrees for the values smaller than `i` and all possible right subtrees for the values larger than `i`.
4. We combine each left subtree with each right subtree to form a unique BST with `i` as the root.

This approach utilizes the power of recursion to break down the problem into smaller subproblems, namely constructing all possible subtrees for a given range and then combining them to form larger and complete BSTs.

By defining a recursive function `gen(left, right)`, which takes a range of values and returns all possible BSTs within that range, we can use this function to explore all possible combinations that satisfy BST properties. This recursion forms the backbone of our solution. It starts by generating all trees for `1-n`, and for each of those, it recursively generates left and right subtrees for the remaining ranges until the ranges are empty, at which point it will use `None` to represent the absence of a node, indicating that the recursion has bottomed out.

This careful assembly of left and right subtrees around each possible root, following BST conditions, will yield all structurally unique BSTs containing `n` nodes.

Solution Approach

The solution uses a recursive helper function `gen(left, right)` which is designed to generate all possible unique BSTs composed of node values ranging from `left` to `right`, inclusive. Here's the step-by-step explanation of the `gen` function and the overall solution approach:

1. **Base Case**: If the `left` limit is greater than the `right` limit, it means there are no values to create a node, so we append `None` to the list of answers. This corresponds to the base case of the recursion which effectively handles the creation of empty trees (required at the leaf nodes of the final BSTs).

2. **Recursive Case**: The recursive case is when `left <= right`, meaning we have at least one value to create a node with.

3. **Generating Roots**: A loop runs from `left` to `right` including both ends. Each iteration of the loop represents choosing a different value as the root of the BST from the given range.

4. **Divide and Conquer**: For each potential root value `i`, we recursively call `gen` to create all possible left subtrees with values from `left` to `i-1` and right subtrees with values from `i+1` to `right`. This is where the divide-and-conquer algorithm comes into play, breaking the original problem into smaller subproblems, each capable of being solved independently.

5. **Constructing Trees**: After the recursive calls, we iterate over all combinations of left and right subtrees. For each combination, a new tree node is created with the value `i` (which we selected as a root earlier) and the left and right pointers set to the current left and right subtree from the combinations. These new trees are then added to the `ans` list.

6. **List of Tree Nodes**: The helper function `gen` eventually returns a list of `TreeNode` objects. Each `TreeNode` is the root of a structurally unique BST.

7. **Final Result**: The `generateTrees` function starts the recursive process by calling `gen(1, n)` and returns the list it receives from this call. The final return value is a list of all unique BSTs with node values from 1 to `n`.

The algorithm makes use of the fundamental properties of BSTs for the divide-and-conquer strategy and employs recursion efficiently to construct all possible outcomes. The data structure used is the `TreeNode` class, which represents the nodes in a BST.

In summary, the solution leverages recursion to explore every single potential node value as a root and attaches to it every combination of left and right subtrees that the sub-ranges can produce, ensuring that all unique BST structures are explored.

Example Walkthrough

Let's go through a small example to illustrate the solution approach using `n = 3`. This means we want to generate all structurally unique BSTs using node values {1, 2, 3}.

Step 1: Call the Main Function

We start by calling `gen(1, 3)` because our range of values is from 1 to 3.

Step 2: Loop Through Each Value as Root

We then establish a loop for selecting the root value from the candidates 1 to 3.

When `i = 1`, `1` is the root, we have no left subtree (`gen(1, 0)` returns `None`) and we need to generate the right subtree with values {2, 3}. The function `gen(2, 3)` is called for the right subtree.

When `i = 2`, `2` is the root, we generate the left subtree with value {1} only (`gen(1, 1)`) and the right subtree with value {3} only (`gen(3, 3)`).

When `i = 3`, `3` is the root, we generate the left subtree with values {1, 2} (`gen(1, 2)`) and there is no right subtree (`gen(4, 3)` returns `None`).

Step 3: Recursively Generate Subtrees

For each of these scenarios, we recursively generate the possible left and right subtrees:

- For `i = 1`, `gen(2, 3)` needs to consider `2` and `3` as potential roots for the right subtree.
- For `i = 2`, since we are considering single elements (`1` for left and `3` for right), `gen(1, 1)` and `gen(3, 3)` directly return the TreeNode with the corresponding value. These are leaf nodes.
- For `i = 3`, `gen(1, 2)` considers both `1` and `2` as roots. If `1` is the root, `2` would be its right child (`gen(2, 2)` returns TreeNode with value 2). If `2` is the root, `1` would be its left child (`gen(1, 1)` returns TreeNode with value 1).

Step 4: Construct Trees and Combine Subtrees

Now we construct the trees by combining the returned TreeNode(s) for the left and right subtrees with the root. For instance:

- With `i = 1` as root, combine `None` as left subtree (since there are no values less than 1) and each TreeNode returned from `gen(2, 3)` as right subtrees individually.
- With `i = 2` as root, set the left child to TreeNode with value `1` and right child to TreeNode with value `3`.
- With `i = 3` as root, set the left child to each TreeNode returned from `gen(1, 2)` and the right child to `None`.

Step 5: Collect and Return the Results

Each complete tree represents one of the possible structurally unique BSTs that the function is designed to generate. Once all trees have been constructed for all `i`, the `gen` function will return the list of root nodes, which forms the output of the `generateTrees` function.

In summary, for `n = 3`, we will end up with 5 unique BSTs, which are constructed by the function through recursion and combination of all possible left and right subtrees for each candidate root from values 1 to 3.

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 generateTrees(self, n: int) -> List[TreeNode]:
        # Recursive function to generate all unique BSTs in the range [start, end]
        def generate(start, end):
            trees = []  # List to store the trees
            # Base case: if start is greater than end, there is no tree to add
            if start > end:
                trees.append(None)
            else:
                # Iterate through each number in the current range as the root
                for root_value in range(start, end + 1):
                    # Recursively generate all possible left subtrees with values less than the root
                    left_trees = generate(start, root_value - 1)
                    # Recursively generate all possible right subtrees with values greater than the root
                    right_trees = generate(root_value + 1, end)

                    # Nested loops to combine each left and right subtree with the current root node
                    for left_subtree in left_trees:
                        for right_subtree in right_trees:
                            # Create a new tree with the current number as the root
                            root = TreeNode(root_value, left_subtree, right_subtree)
                            trees.append(root)
            return trees

        # Return all possible unique BSTs generated from values 1 to n
        return generate(1, n)

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    // Constructor to create a node without any children.
    TreeNode() {}

    // Constructor to create a node with a specific value.
    TreeNode(int val) { this.val = val; }

    // Constructor to create a node with a specific value and given left and right children.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // Generates all structurally unique BSTs (binary search trees) that store values 1...n.
    public List<TreeNode> generateTrees(int n) {
        return generateTreesInRange(1, n);
    }

    // Helper function that generates all structurally unique BSTs for values in the range [start, end].
    private List<TreeNode> generateTreesInRange(int start, int end) {
        List<TreeNode> trees = new ArrayList<>();
        // If start is greater than end, there is no tree to add (empty tree).
        if (start > end) {
            trees.add(null);
        } else {
            // Try each number as root, and recursively generate left and right subtrees.
            for (int rootValue = start; rootValue <= end; ++rootValue) {
                // Generate all left subtrees by considering numbers before the root value.
                List<TreeNode> leftSubtrees = generateTreesInRange(start, rootValue - 1);
                // Generate all right subtrees by considering numbers after the root value.
                List<TreeNode> rightSubtrees = generateTreesInRange(rootValue + 1, end);
                // Combine left and right subtrees with the current rootValue to form unique BSTs.
                for (TreeNode leftSubtree : leftSubtrees) {
                    for (TreeNode rightSubtree : rightSubtrees) {
                        // Create a new tree node as the root with leftSubtree and rightSubtree as children.
                        TreeNode root = new TreeNode(rootValue, leftSubtree, rightSubtree);
                        // Add the newly formed tree to the list of trees.
                        trees.add(root);
                    }
                }
            }
        }
        // Return all possible BSTs formed from values in the range [start, end].
        return trees;
    }
}

```

C++ Solution

```cpp
#include <vector>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructor to initialize a tree node with default values
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor to initialize a tree node with a specific value
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor to initialize a tree node with a value and left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Main function to generate all structurally unique BSTs (binary search trees) that store values 1 to n
    std::vector<TreeNode*> generateTrees(int n) {
        return generateTreesInRange(1, n);
    }

private:
    // Helper function that generates all unique BSTs within a specific value range (left to right inclusive)
    std::vector<TreeNode*> generateTreesInRange(int left, int right) {
        std::vector<TreeNode*> trees; // Stores the list of unique BSTs for the current range
        // Base case: if there are no numbers to construct the tree with, add a nullptr to represent empty
        if (left > right) {
            trees.push_back(nullptr);
        } else {
            // Loop through each number within the range to make it the root of BSTs
            for (int rootValue = left; rootValue <= right; ++rootValue) {
                // Recursively generate all possible left subtrees
                std::vector<TreeNode*> leftSubtrees = generateTreesInRange(left, rootValue - 1);
                // Recursively generate all possible right subtrees
                std::vector<TreeNode*> rightSubtrees = generateTreesInRange(rootValue + 1, right);
                // Combine the left and right subtrees with the current root node to form unique BSTs
                for (TreeNode* leftSubtree : leftSubtrees) {
                    for (TreeNode* rightSubtree : rightSubtrees) {
                        TreeNode* rootNode = new TreeNode(rootValue, leftSubtree, rightSubtree);
                        // Add the newly formed BST to the list of BSTs for this range
                        trees.push_back(rootNode);
                    }
                }
            }
        }
        return trees; // Return the list of BSTs generated for this range
    }
};

```

Typescript Solution

```typescript
// TreeNode definition representing a node in the binary tree.
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

// Generates all structurally unique BSTs (binary search trees) that store values 1 to n.
function generateTrees(n: number): Array<TreeNode | null> {
    if (n == 0) return [];
    // Call the recursive helper to construct the trees.
    return constructTrees(1, n);
}

// Helper function that returns an array of all possible BSTs within a range of values.
function constructTrees(start: number, end: number): Array<TreeNode | null> {
    let trees: Array<TreeNode | null> = [];
    if (start > end) {
        trees.push(null); // Base case: if there are no numbers to add, add null for the tree node.
        return trees;
    }

    // Iterate all numbers from start to end, considering each number as the root.
    for (let i: number = start; i <= end; i++) {
        // Recursively construct all possible left and right subtrees.
        let leftSubtrees: Array<TreeNode | null> = constructTrees(start, i - 1);
        let rightSubtrees: Array<TreeNode | null> = constructTrees(i + 1, end);

        // Combine each left and right subtree with the root node 'i'.
        for (let left of leftSubtrees) {
            for (let right of rightSubtrees) {
                // Create the root node with the current value 'i'.
                let rootNode: TreeNode = { val: i, left: left, right: right };
                trees.push(rootNode); // Add the formed tree to the list.
            }
        }
    }
    return trees; // Return all possible trees.
}

// Here, the TreeNode definition should be present in the global scope to match the existing functions.
// The TreeNode constructor/class definition is omitted since it is not to be defined outside of a class as per the instructions.

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be tricky to analyze due to its recursive nature and because it generates all unique binary search trees (BSTs) with `n` distinct nodes. The number of such trees is given by the `n`th Catalan number, which is `C_n = (2n)! / ((n+1)!n!)`.

In the worst case, the recursion tree would have a branching factor equal to the maximum number of nodes `n`, at each node there is a loop that runs from `left` to `right`. As we recurse down the tree and back, we create new trees for each possible root node.

For each value of `i` (from `left` to `right` inclusive), we generate all possible left subtrees with `gen(left, i - 1)` and all possible right subtrees with `gen(i + 1, right)`. Then, for each combination of left and right subtrees, we connect them to a root node with value `i`. Since the number of combinations for the left and right subtrees is the product of the count of left subtrees and the count of right subtrees, we have a multiplication of possibilities each time we perform this operation.

Considering the above factors, the time complexity is O(n * C_n), where `C_n` is the `n`th Catalan number, since we have `n` possibilities at each level of the recursive stack.

The time taken for all recursive calls can thus be approximated as proportional to the number of unique BSTs generated, which is the Catalan number, times `n`, giving us a time complexity of `O(n * C_n)`.

Space Complexity

The space complexity consists of the space needed for the output, which stores all unique BSTs, and the space needed for the execution stack during recursion.

The output itself will hold C_n trees, and each tree has `n` nodes, so the space needed to store these trees is also proportional to `O(n * C_n)`.

The recursion stack space will be proportional to the height of the recursion tree, which is `O(n)` since in the worst case we will make `n` nested recursive calls before hitting the base case. However, since the space required for the recursion stack is significantly smaller compared to the output space, it is often considered secondary.

Thus, the space complexity of the code is `O(n * C_n)` for storing the unique BSTs generated.