95. Unique Binary Search Trees II Medium **Binary Search Tree** Backtracking **Dynamic Programming**

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

Binary Tree

Problem Description

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

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.

nodes with keys less than the node's key, and the right subtree only contains nodes with keys greater than the node's key.

- 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

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

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

(required at the leaf nodes of the final BSTs).

value as the root of the BST from the given range.

recursion has bottomed out.

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.

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.

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

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

TreeNode with the corresponding value. These are leaf nodes.

of all possible left and right subtrees for each candidate root from values 1 to 3.

Recursive function to generate all unique BSTs in the range [start, end]

Base case: if start is greater than end, there is no tree to add

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

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.

The algorithm makes use of the fundamental properties of BSTs for the divide-and-conquer strategy and employs recursion

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}.

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

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

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

(gen(3, 3)).

None).

Example Walkthrough

Step 1: Call the Main Function

Step 2: Loop Through Each Value as Root

Step 3: Recursively Generate Subtrees

Step 4: Construct Trees and Combine Subtrees

3). The function gen(2, 3) is called for the right subtree.

 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

• 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).

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

With i = 3 as root, set the left child to each TreeNode returned from gen(1, 2) and the right child to None.

 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.

Each complete tree represents one of the possible structurally unique BSTs that the function is designed to generate. Once all trees

In summary, for n = 3, we will end up with 5 unique BSTs, which are constructed by the function through recursion and combination

have been constructed for all i, the gen function will return the list of root nodes, which forms the output of the generateTrees

function.

Definition for a binary tree node.

self.right = right

def generate(start, end):

if start > end:

return trees

return generate(1, n)

// Definition for a binary tree node.

trees.append(None)

def generateTrees(self, n: int) -> List[TreeNode]:

trees = [] # List to store the trees

trees.append(root)

Return all possible unique BSTs generated from values 1 to n

Python Solution

class Solution:

10

12

13

14

15

28

29

30

31

32

33

34

56

58

57 }

8

9

10

11

12

13

14

15

16

18

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

46

47

48

49

50

51

53

52 };

17 };

C++ Solution

1 #include <vector>

4 struct TreeNode {

19 class Solution {

public:

private:

int val;

TreeNode *left;

TreeNode *right;

// Definition for a binary tree node.

// Constructor to initialize a tree node with default values

// Constructor to initialize a tree node with a specific value

std::vector<TreeNode*> generateTreesInRange(int left, int right) {

// 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) {}

// Main function to generate all structurally unique BSTs (binary search trees) that store values 1 to n

// Helper function that generates all unique BSTs within a specific value range (left to right inclusive)

// Base case: if there are no numbers to construct the tree with, add a nullptr to represent empty

std::vector<TreeNode*> leftSubtrees = generateTreesInRange(left, rootValue - 1);

// Add the newly formed BST to the list of BSTs for this range

std::vector<TreeNode*> rightSubtrees = generateTreesInRange(rootValue + 1, right);

// Combine the left and right subtrees with the current root node to form unique BSTs

TreeNode* rootNode = new TreeNode(rootValue, leftSubtree, rightSubtree);

std::vector<TreeNode*> trees; // Stores the list of unique BSTs for the current range

// Loop through each number within the range to make it the root of BSTs

for (int rootValue = left; rootValue <= right; ++rootValue) {</pre>

// Recursively generate all possible left subtrees

// Recursively generate all possible right subtrees

for (TreeNode* rightSubtree : rightSubtrees) {

for (TreeNode* leftSubtree : leftSubtrees) {

trees.push_back(rootNode);

return trees; // Return the list of BSTs generated for this range

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

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

std::vector<TreeNode*> generateTrees(int n) {

return generateTreesInRange(1, n);

trees.push_back(nullptr);

if (left > right) {

} else {

Java Solution

class TreeNode {

int val;

import java.util.ArrayList;

import java.util.List;

TreeNode left;

Step 5: Collect and Return the Results

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

23 # Nested loops to combine each left and right subtree with the current root node 24 25 for left_subtree in left_trees: for right_subtree in right_trees: 26 27

Create a new tree with the current number as the root

root = TreeNode(root_value, left_subtree, right_subtree)

```
16
               else:
                   # Iterate through each number in the current range as the root
17
                    for root_value in range(start, end + 1):
18
                        # Recursively generate all possible left subtrees with values less than the root
19
20
                        left_trees = generate(start, root_value - 1)
21
                        # Recursively generate all possible right subtrees with values greater than the root
                        right_trees = generate(root_value + 1, end)
22
```

```
TreeNode right;
9
       // Constructor to create a node without any children.
10
       TreeNode() {}
11
12
       // Constructor to create a node with a specific value.
13
       TreeNode(int val) { this.val = val; }
14
15
       // Constructor to create a node with a specific value and given left and right children.
16
       TreeNode(int val, TreeNode left, TreeNode right) {
17
           this.val = val;
           this.left = left;
19
           this.right = right;
20
21
22 }
23
   class Solution {
25
       // Generates all structurally unique BSTs (binary search trees) that store values 1...n.
       public List<TreeNode> generateTrees(int n) {
26
27
           return generateTreesInRange(1, n);
28
29
30
       // Helper function that generates all structurally unique BSTs for values in the range [start, end].
       private List<TreeNode> generateTreesInRange(int start, int end) {
32
           List<TreeNode> trees = new ArrayList<>();
33
           // If start is greater than end, there is no tree to add (empty tree).
34
           if (start > end) {
35
               trees.add(null);
           } else {
36
37
               // Try each number as root, and recursively generate left and right subtrees.
38
               for (int rootValue = start; rootValue <= end; ++rootValue) {</pre>
39
                   // Generate all left subtrees by considering numbers before the root value.
                    List<TreeNode> leftSubtrees = generateTreesInRange(start, rootValue - 1);
40
                   // Generate all right subtrees by considering numbers after the root value.
41
                   List<TreeNode> rightSubtrees = generateTreesInRange(rootValue + 1, end);
42
43
                   // Combine left and right subtrees with the current rootValue to form unique BSTs.
                    for (TreeNode leftSubtree : leftSubtrees) {
44
45
                        for (TreeNode rightSubtree : rightSubtrees) {
                            // Create a new tree node as the root with leftSubtree and rightSubtree as children.
46
                            TreeNode root = new TreeNode(rootValue, leftSubtree, rightSubtree);
47
48
                            // Add the newly formed tree to the list of trees.
49
                            trees.add(root);
50
51
52
53
54
           // Return all possible BSTs formed from values in the range [start, end].
55
           return trees;
```

Typescript Solution 1 // TreeNode definition representing a node in the binary tree.

```
2 interface TreeNode {
         val: number;
         left: TreeNode | null;
         right: TreeNode | null;
  6
    // 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 [];
 10
        // Call the recursive helper to construct the trees.
 11
         return constructTrees(1, n);
 12
 13 }
 14
    // 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> = [];
 17
         if (start > end) {
 18
             trees.push(null); // Base case: if there are no numbers to add, add null for the tree node.
 19
 20
             return trees;
 21
 22
 23
         // Iterate all numbers from start to end, considering each number as the root.
         for (let i: number = start; i <= end; i++) {</pre>
 24
 25
             // Recursively construct all possible left and right subtrees.
 26
             let leftSubtrees: Array<TreeNode | null> = constructTrees(start, i - 1);
             let rightSubtrees: Array<TreeNode | null> = constructTrees(i + 1, end);
 27
 28
             // Combine each left and right subtree with the root node 'i'.
 29
 30
             for (let left of leftSubtrees) -
                 for (let right of rightSubtrees) {
 31
 32
                     // Create the root node with the current value 'i'.
                     let rootNode: TreeNode = { val: i, left: left, right: right };
 33
 34
                     trees.push(rootNode); // Add the formed tree to the list.
 35
 36
 37
 38
         return trees; // Return all possible trees.
 39
 40
    // 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.
 43
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 nth 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
```

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.

Space Complexity

* C_n).

Catalan number, times n, giving us a time complexity of $O(n * C_n)$.

Considering the above factors, the time complexity is O(n * C_n), where C_n is the nth 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

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

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 0(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

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

smaller compared to the output space, it is often considered secondary.