96. Unique Binary Search Trees

Medium | Tree | Binary Search Tree | Math | Dynamic Programming | Binary Tree

Leetcode Link

Problem Description

The problem asks for the number of structurally unique Binary Search Trees (BSTs) that can be formed using exactly `n` nodes, each node having a unique value from 1 to `n`. A Binary Search Tree is a tree where each node satisfies the following conditions:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Understanding this, we need to calculate the count of all possible unique tree structures without considering the actual node values, as the unique values will be distributed in the same structure in exactly one way due to BST property.

Intuition

To solve this problem, we can use a method known as dynamic programming. Dynamic programming breaks down the problem into smaller subproblems and uses the results of these subproblems to solve larger ones.

The intuition behind the solution can be understood in terms of the following points:

1. Consider that the number of unique BSTs with `0` nodes (an empty tree) is `1`. This serves as our base case.
2. The key to solving the problem is realizing that when we choose a number `i` from `1` to `n` to be the root of a BST, then `1` to `i-1` will necessarily form the left subtree and `i+1` to `n` will form the right subtree.
3. The number of unique BSTs that can be formed with `i` as the root is the product of the number of unique BSTs that can be formed with `1` to `i-1` and the number of unique BSTs that can be formed with `i+1` to `n`.
4. Thus, we iterate through all numbers from `1` to `n`, treating each as the root, and multiply the number of ways to form left and right subtrees, summing these products up to get the total number of unique BSTs for `n` nodes.

The dynamic programming array `dp` will hold the solution to our subproblems where `dp[i]` represents the number of unique BSTs that can be formed with `i` distinct nodes. We iteratively fill up this array by using the results of smaller subproblems to build the solutions to larger subproblems.

Solution Approach

The solution uses dynamic programming to build up the number of unique BSTs that can be formed for each number of nodes from `0` to `n`. The following steps are executed in the solution:

1. A list `dp` is initialized with `n + 1` elements, all set to `0`. This list holds the solutions to our subproblems where `dp[i]` gives the number of unique BSTs that can be formed with `i` nodes. Since we start counting nodes from `1`, we make our list one element larger than `n` to make it easier to refer to `dp[n]`.

2. The first element `dp[0]` is set to `1`. This represents the fact that there is exactly one BST that can be formed with `0` nodes, the empty tree.

3. We then use a nested loop where we consider every number `i` from `1` to `n` to represent the number of nodes we are currently interested in. For each `i`, we consider every possible root `j` from `1` to `i`. Hence, we calculate the number of unique trees with `j` as the root by multiplying the number of unique trees that can be formed with `j-1` nodes (left subtree) and `i-j` nodes (right subtree).

4. For each `i` and `j`, we update the `dp` array by adding the product `dp[j] * dp[i - j - 1]` to `dp[i]`. This is done because when `j` is the root, there are `dp[j]` ways to construct the left subtree from `0` to `j-1` and `dp[i - j - 1]` ways to construct the right subtree from `j+1` to `i`. Since the root is fixed, the left and right subtrees form independently of each other; thus the total number is the product of possibilities for each side.

5. After the loops complete, `dp[n]` contains the final result, which is the number of unique BSTs that can be made with `n` distinct nodes.

In our case, `dp[-1]` is equivalent to `dp[n]` since Python supports negative indexing and refers to the last element of the list. This value is then returned by the function.

This implementation uses a bottom-up approach, and each subproblem is solved only once and then reused, leading to an efficient algorithm with a time complexity of O(n^2). The space complexity is O(n), as we need to store the number of unique BSTs for each number from `0` to `n`.

Example Walkthrough

To illustrate the solution approach, let's walk through a small example where `n = 3`. We are seeking to find the number of structurally unique BSTs that can be formed with nodes valued from `1` to `3`.

According to the solution approach, these are the steps we would follow:

1. Initialize a dynamic programming list `dp` of size `n+1` and set all its elements to `0`. This list will store the number of unique BSTs for each number of nodes, so in our case, `dp = [0, 0, 0, 0]`.
2. Set the base case `dp[0]` to `1`, which represents the count of unique BSTs with `0` nodes (empty tree). It means that our `dp` list now looks like this: `dp = [1, 0, 0, 0]`.
3. Now, we need to compute `dp[i]` for each `i` from `1` to `n`. Let's start by finding `dp[1]`.
   - For `i = 1` (with only one node), regardless of the value, there's only one way to create a BST. So, `dp[1] = 1`.
4. Let's move on to compute `dp[2]` (two nodes). We have two scenarios here:
   - We pick `1` as the root. There is `dp[0]` ways to arrange the left subtree since there are zero nodes to the left of `1`, and `dp[1]` ways to arrange the right subtree, with just one node `2`.
   - We pick `2` as the root. There is `dp[1]` ways to arrange the left subtree with just one node `1`, and `dp[0]` ways for the right subtree since there are no nodes to the right of `2`.
   - The total, combining both scenarios, gives us `dp[2] = dp[0]*dp[1] + dp[1]*dp[0] = 1*1 + 1*1 = 2`.
5. Finally, we have to compute `dp[3]`.
   - Picking `1` as the root: No nodes on the left, so we have `dp[0]` ways; and two nodes on the right (`2` and `3`), giving us `dp[2]` ways.
   - Picking `2` as the root: One node on the left (`1`), so `dp[1]` ways; and one node on the right (`3`), also `dp[1]` ways.
   - Picking `3` as the root: Two nodes on the left (`1` and `2`), so `dp[2]` ways; and no nodes on the right, giving `dp[0]` ways.
   - We sum up all these products to get: `dp[3] = dp[0]*dp[2] + dp[1]*dp[1] + dp[2]*dp[0] = 1*2 + 1*1 + 2*1 = 5`.

So, the array `dp` at the end of our computation is `[1, 1, 2, 5]`. The number of unique BSTs that can be formed with `3` nodes is `dp[3]`, which is `5`. Hence, the function would return `5` for `n = 3`. This method can be extended for any larger `n`, using the same dynamic programming approach.

Python Solution

```python
class Solution:
    def numTrees(self, n: int) -> int:
        # Initialize a list for dynamic programming with a size of n + 1.
        # Set the first element to 1 because there's exactly one structure for an empty tree.
        num_trees = [0] * (n + 1)
        num_trees[0] = 1

        # Build the number of unique BSTs (Binary Search Trees) for each count of nodes
        # from 1 up to and including n. The outer loop goes through numbers of nodes.
        for nodes in range(1, n + 1):
            # The inner loop goes through the number of nodes by considering all possible left subtrees
            # (which are represented by j) and right subtrees (which are represented by nodes - j - 1).
            for left_tree_nodes in range(nodes):
                # The number of unique BSTs with 'nodes' nodes is the sum of
                # the number of unique BSTs for each left subtree (j) multiplied by
                # the number of unique BSTs for each right subtree (nodes - j - 1).
                num_trees[nodes] += num_trees[left_tree_nodes] * num_trees[nodes - left_tree_nodes - 1]

        # The last element of the list will hold the result for n nodes.
        return num_trees[-1]
```

Java Solution

```java
class Solution {
    // Function to compute number of unique BSTs with n nodes
    public int numTrees(int n) {
        // Initialize the dp array to store the number of unique BSTs for each count of nodes
        int[] dp = new int[n + 1];

        // There is one unique BST for a tree with zero nodes, which is an empty tree
        dp[0] = 1;

        // Iterate over each count of nodes from 1 to n
        for (int nodes = 1; nodes <= n; ++nodes) {
            // Calculate the number of unique BSTs for 'nodes' number of nodes
            for (int root = 0; root < nodes; ++root) {
                // For each position 'root', the number of trees is the product of the
                // number of unique trees in the left subtree (dp[root])
                // and the number of unique trees in the right subtree (dp[nodes - root - 1]).
                dp[nodes] += dp[root] * dp[nodes - root - 1];
            }
        }

        // Return the result for n nodes
        return dp[n];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to compute the number of unique BSTs (Binary Search Trees) that can be constructed with 'n' distinct nodes
    int numTrees(int n) {
        // Create a dp (Dynamic Programming) vector with 'n+1' elements initialized to 0
        // dp[i] will store the number of unique BSTs that can be constructed with 'i' nodes
        vector<int> dp(n + 1, 0);

        // The number of BSTs with 0 nodes (empty tree) is 1
        dp[0] = 1;

        // Build up the number of BSTs for each number of nodes from 1 to 'n'
        for (int nodes = 1; nodes <= n; ++nodes) {
            // Iterate through all possible left subtrees with 'left' nodes
            for (int left = 0; left < nodes; ++left) {
                // The number of right nodes is 'nodes - left - 1' (because one node is the root)
                int right = nodes - left - 1;
                // Multiply the number of ways to construct the left subtree with 'left' nodes
                // by the number of ways to construct the right subtree with 'right' nodes
                dp[nodes] += dp[left] * dp[right];
            }
        }

        // The result is the number of unique BSTs that can be constructed with 'n' nodes
        return dp[n];
    }
};
```

Typescript Solution

```typescript
// Function to compute the number of unique BSTs (Binary Search Trees)
// that can be constructed with 'n' distinct nodes
function numTrees(n: number): number {
    // Create a dp (Dynamic Programming) array with 'n+1' elements initialized to 0
    // dp[i] will store the number of unique BSTs that can be constructed with 'i' nodes
    let dp: number[] = new Array(n + 1).fill(0);

    // The number of BSTs with 0 nodes (empty tree) is 1
    dp[0] = 1;

    // Build up the number of BSTs for each number of nodes from 1 to 'n'
    for (let nodes = 1; nodes <= n; nodes++) {
        // Iterate through all possible left subtrees with 'left' nodes
        for (let left = 0; left < nodes; left++) {
            // The number of right nodes is 'nodes - left - 1' (because one node is the root)
            let right = nodes - left - 1;
            // Multiply the number of ways to construct the left subtree with 'left' nodes
            // by the number of ways to construct the right subtree with 'right' nodes
            dp[nodes] += dp[left] * dp[right];
        }
    }

    // The result is the number of unique BSTs that can be constructed with 'n' nodes
    return dp[n];
}

```

Time and Space Complexity

The given code is designed to calculate the number of structurally unique BSTs (binary search trees) that store values 1 through `n` using dynamic programming. To analyze the time and space complexity, we will consider the main operations and how they scale with respect to the input `n`.

Time Complexity

- We initialize a list `dp` of size `n+1` with zero, which takes `O(n)` time.
- There is a nested loop where the outer loop runs from `1` to `n` and the inner loop runs `i` times (which is at most `n` times when `i==n`).
- Within the inner loop, the corresponding element in `dp` is updated, and this operation is constant time, `O(1)`.
- Thus, the time complexity for the nested loops can be approximated as the sum of the first `n` natural numbers, which is `O(n^2)`.

Therefore, the overall time complexity of the code is `O(n^2)`.

Space Complexity

- We have a list `dp` that contains `n+1` elements, which represents the space complexity of `O(n)`.
- No other data structures are used that grow with `n`, so `O(n)` represents the total space complexity.

In conclusion, the time complexity of the code is `O(n^2)` and the space complexity is `O(n)`.