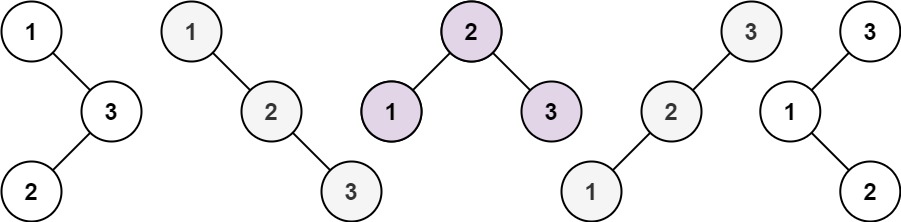
<https://leetcode.com/problems/unique-binary-search-trees/description/>

Given an integer n, return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.

**Example 1:**



**Input:** n = 3

**Output:** 5

**Example 2:**

**Input:** n = 1

**Output:** 1

**Constraints:**

1 <= n <= 19

**Attempt 1: 2024-07-10**

**Solution 1: Follow same style as L95 (30 min, Memory Limit Exceeded)**

class Solution {

    public int numTrees(int n) {

        List<String> result = helper(1, n);

        return result.size();

    }

    private List<String> helper(int start, int end) {

        List<String> result = new ArrayList<>();

        if(start > end) {

            result.add("null");

            return result;

        }

        for(int i = start; i <= end; i++) {

            List<String> left = helper(start, i - 1);

            List<String> right = helper(i + 1, end);

            for(String l : left) {

                for(String r : right) {

                    String tree = "(" + i + " " + l + " " + r + ")";

                    result.add(tree);

                }

            }

        }

        return result;

    }

}

**Refer to chatGPT**

**Q: I know Leetcode 96 only focus on count the number, but how can we calculate number by printing out all actual trees first then find its count, which is similar like what Leetcode 95 doing ?**

If you want to solve Leetcode 96 by actually generating all the possible unique BSTs (like in Leetcode 95) and then counting them, you can do so by adapting the solution from Leetcode 95. Here's how you can modify the code to generate all BSTs and then count them:

Generate All BSTs: Use a recursive function similar to the one used in Leetcode 95 to generate all unique BSTs.

Count the Trees: Simply count the number of trees generated.

Here's the complete solution in Java:

import java.util.ArrayList;

import java.util.List;

public class Solution {

public static class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public int numTrees(int n) {

List<TreeNode> trees = generateTrees(1, n);

return trees.size();

}

private List<TreeNode> generateTrees(int start, int end) {

List<TreeNode> ans = new ArrayList<>();

// Base case: if start > end, there are no nodes, so return a list with null

if (start > end) {

ans.add(null);

return ans;

}

// Try each number as a root

for (int i = start; i <= end; i++) {

// Generate all possible left subtrees

List<TreeNode> leftTrees = generateTrees(start, i - 1);

// Generate all possible right subtrees

List<TreeNode> rightTrees = generateTrees(i + 1, end);

// Combine each left and right subtree with the current root `i`

for (TreeNode left : leftTrees) {

for (TreeNode right : rightTrees) {

TreeNode root = new TreeNode(i);

root.left = left;

root.right = right;

ans.add(root);

}

}

}

return ans;

}

public static void main(String[] args) {

Solution solution = new Solution();

int n = 3; // You can change this value to test other cases

int count = solution.numTrees(n);

System.out.println("Number of unique BSTs for n = " + n + " is: " + count);

}

}

**Q: For Leetcode 96, is there a way we can print all unique BSTs but no need define TreeNode class ?**

**For Leetcode 96, we need to generate and count all unique Binary Search Trees (BSTs). However, if you want to print all unique BSTs without defining a TreeNode class, you can represent the trees in a different format, such as nested lists or strings.**

**Here is how you can achieve this using a string representation of trees:**

**Approach**

Generate All BSTs: Use a recursive function to generate all unique BSTs, but represent each tree as a string.

Print and Count the Trees: Store and print each tree representation and then count them.

Here's how you can do it in Java:

import java.util.ArrayList;

import java.util.List;

public class Solution {

public int numTrees(int n) {

List<String> trees = generateTrees(1, n);

return trees.size();

}

private List<String> generateTrees(int start, int end) {

List<String> ans = new ArrayList<>();

// Base case: if start > end, there are no nodes, so return a list with "null"

if (start > end) {

ans.add("null");

return ans;

}

// Try each number as a root

for (int i = start; i <= end; i++) {

// Generate all possible left subtrees

List<String> leftTrees = generateTrees(start, i - 1);

// Generate all possible right subtrees

List<String> rightTrees = generateTrees(i + 1, end);

// Combine each left and right subtree with the current root `i`

for (String left : leftTrees) {

for (String right : rightTrees) {

String tree = "(" + i + " " + left + " " + right + ")";

ans.add(tree);

}

}

}

return ans;

}

public static void main(String[] args) {

Solution solution = new Solution();

int n = 3; // You can change this value to test other cases

int count = solution.numTrees(n);

System.out.println("Number of unique BSTs for n = " + n + " is: " + count);

}

}

**Explanation**

numTrees Method: This is the main method that generates all unique BSTs and prints their string representations, then returns their count.

generateTrees Method: Recursively generates all unique BSTs for a given range [start, end], but represents each tree as a string:

If start > end, it returns a list containing "null", representing an empty subtree.

For each possible root i from start to end, it generates all possible left subtrees and right subtrees.

It combines each left subtree with each right subtree and constructs a string representation of the tree with root i.

Tree Representation: Each tree is represented as a string in the format "(root left right)", where root is the root value, and left and right are the string representations of the left and right subtrees, respectively.

**Complexity**

Time Complexity: The time complexity is high due to the combinatorial nature of generating all unique BSTs. It is exponential in the number of nodes.

Space Complexity: The space complexity is also high due to the storage of all generated tree strings. The number of unique BSTs for n nodes is given by the nth Catalan number, which grows exponentially.

**Conclusion**

This approach solves Leetcode 96 by generating all unique BSTs using string representations instead of defining a TreeNode class. It is an educational method to visualize the problem and understand the generation of unique BSTs. However, it is not as efficient as the dynamic programming approach for counting BSTs in terms of time and space complexity.

**Solution 2: Native DFS (30 min)**

class Solution {

    public int numTrees(int n) {

        return helper(n);

    }

    private int helper(int n) {

        // There is one unique BST for a tree with zero nodes, which is an empty tree

        // so n = 0 create empty tree and return 1

        if(n <= 1) {

            return 1;

        }

        int result = 0;

        for(int i = 1; i <= n; i++) {

// For each root i, left subtree has i - 1 nodes(i from 1 to i - 1, total i - 1),

// right subtree has n - i nodes(i from i + 1 to n, total n - i)

            result += helper(i - 1) \* helper(n - i);

        }

        return result;

    }

}

Time Complexity : O(3^N), where N is the given number of nodes in BST. Read here for proof.

Space Complexity : O(N), the maximum recursive stack depth.

**The base condition could also change from n <= 1 to n < 1, since we only need empty tree = 1 as base condition**

class Solution {

    public int numTrees(int n) {

        return helper(n);

    }

    private int helper(int n) {

        // There is one unique BST for a tree with zero nodes, which is an empty tree

        // so n = 0 create empty tree and return 1

        if(n < 1) {

            return 1;

        }

        int result = 0;

        for(int i = 1; i <= n; i++) {

// For each root i, left subtree has i - 1 nodes(i from 1 to i - 1, total i - 1),

// right subtree has n - i nodes(i from i + 1 to n, total n - i)

            result += helper(i - 1) \* helper(n - i);

        }

        return result;

    }

}

Time Complexity : O(3^N), where N is the given number of nodes in BST. Read here for proof.

Space Complexity : O(N), the maximum recursive stack depth.

**Solution 3: DFS + Memoization (10 min)**

class Solution {

    public int numTrees(int n) {

        Integer[] memo = new Integer[n + 1];

        return helper(n, memo);

    }

    private int helper(int n, Integer[] memo) {

        // There is one unique BST for a tree with zero nodes, which is an empty tree

        // so n = 0 create empty tree and return 1

        if(n <= 1) {

            return 1;

        }

        if(memo[n] != null) {

            return memo[n];

        }

        int result = 0;

        for(int i = 1; i <= n; i++) {

            // For each root i, left subtree has i - 1 nodes(i from 1 to i - 1, total i - 1),

            // right subtree has n - i nodes(i from i + 1 to n, total n - i)

            result += helper(i - 1, memo) \* helper(n - i, memo);

        }

        return memo[n] = result;

    }

}

Time Complexity : O(N^2)

Here we calculate numTrees(i) (for 1<=i<=N) only once and memoize it which will take O(N).

For calculating each of numTrees(i), we need N iterations to calculate numTrees(0) \* numTrees(i)

+ numTrees(1) \* numTrees(i-1) + numTrees(2) \* numTrees(i-2)+ ... + numTrees(i) \* numTrees(0).

Thus, the overall time complexity becomes O(N \* N).

Space Complexity : O(N), required for recursion and memoization

**The base condition could also change from n <= 1 to n < 1, since we only need empty tree = 1 as base condition**

class Solution {

    public int numTrees(int n) {

        Integer[] memo = new Integer[n + 1];

        return helper(n, memo);

    }

    private int helper(int n, Integer[] memo) {

        // There is one unique BST for a tree with zero nodes, which is an empty tree

        // so n = 0 create empty tree and return 1

        if(n < 1) {

            return 1;

        }

        if(memo[n] != null) {

            return memo[n];

        }

        int result = 0;

        for(int i = 1; i <= n; i++) {

            // For each root i, left subtree has i - 1 nodes(i from 1 to i - 1, total i - 1),

            // right subtree has n - i nodes(i from i + 1 to n, total n - i)

            result += helper(i - 1, memo) \* helper(n - i, memo);

        }

        return memo[n] = result;

    }

}

Time Complexity : O(N^2)

Here we calculate numTrees(i) (for 1<=i<=N) only once and memoize it which will take O(N).

For calculating each of numTrees(i), we need N iterations to calculate numTrees(0) \* numTrees(i)

+ numTrees(1) \* numTrees(i-1) + numTrees(2) \* numTrees(i-2)+ ... + numTrees(i) \* numTrees(0).

Thus, the overall time complexity becomes O(N \* N).

Space Complexity : O(N), required for recursion and memoization

**Solution 4: DP (60 min)**

class Solution {

    public int numTrees(int n) {

        int[] dp = new int[n + 1];

        dp[0] = dp[1] = 1;

        for(int i = 2; i <= n; i++) {

            for(int j = 1; j <= i; j++) {

                dp[i] += dp[j - 1] \* dp[i - j];

            }

        }

        return dp[n];

    }

}

Time Complexity : O(N^2), we iterate over the range i=[2, n] and iteratively calculate dp[i].

The total number of operations performed equals 2+3+4+5..n = (n\*(n+1)/2)-1 ≈ O(N^2)

Space Complexity : O(N), required to store the results in dp

**Actually if only define dp[0] = 1, NO need dp[1] = 1 is also fine, which exactly only require mapping to empty tree condition**

**Then we calculate result for each number of nodes i from 1...n one after another instead from 2 ...n.**

class Solution {

    public int numTrees(int n) {

        int[] dp = new int[n + 1];

        dp[0] = 1;

        for(int i = 1; i <= n; i++) {

            for(int j = 1; j <= i; j++) {

                dp[i] += dp[j - 1] \* dp[i - j];

            }

        }

        return dp[n];

    }

}

Time Complexity : O(N^2), we iterate over the range i=[2, n] and iteratively calculate dp[i].

The total number of operations performed equals 2+3+4+5..n = (n\*(n+1)/2)-1 ≈ O(N^2)

Space Complexity : O(N), required to store the results in dp

**[C++/Python] 5 Easy Solutions w/ Explanation | Optimization from Brute-Force to DP to Catalan O(N)**

**Refer to**

<https://leetcode.com/problems/unique-binary-search-trees/solutions/1565543/c-python-5-easy-solutions-w-explanation-optimization-from-brute-force-to-dp-to-catalan-o-n/>

❌ **Solution - I (Brute-Force) [TLE]**

Let's start by trying to solve the problem in Brute-Force manner. To form structurally unique BST consisting of n nodes, we can start by taking any of the node 1...n as the root node. Let the chosen root node be i. Then, we have a BST where the root node is i, the left child consist of all nodes from 1...i-1 (since left sub-tree must have only less than root's value) and right child consist of all nodes from i+1...n.

1 1 2 3 3

\ \ / \ / /

3 2 1 3 2 1

/ \ / \

2 3 1 2

i = 1 i = 2 i = 3

(i = root node)

**Now, we need to realize that the number of structurally unique BST formable with nodes having value i+1...n is equal to the number of structurally unique BST formable with nodes having value i+1-i...n-i = 1...n-i. Why? Because we only need to find BST which are structurally unique irrespective of their values and we can form an equal number of them with nodes from 1...n or 2...n+1 or n...2n-1 and so on. So, the number only depends on number of nodes using which BST is to be formed.**

**Now, when we choose i as root node, we will have nodes from 1...i-1 (i-1 nodes in total) in left sub-tree and nodes from i+1...n (n-i nodes in total) in the right side. We can then form numTrees(i-1) BSTs for left sub-tree and numTrees(n-i) BSTs for the right sub-tree. The total number of structurally unique BSTs formed having root i will be equal to product of these two, i.e, numTrees(i-1) \* numTrees(n-i). The same can be followed recursively till we reach base case - numTrees(0) = numTrees(1) = 1 because we can form only a single empty BST and single node BST in these cases respectively.**

The final answer will be summation of answers considering all 1...n as root nodes.

3 2 1

/ \ / \ / \

numTrees(2) numTrees(0) numTrees(1) numTrees(1) numTrees(0) numTrees(2)

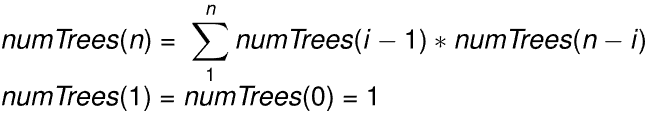
i = 3 i = 2 i = 1

i

=> / \

numTrees(i-1)numTrees(n-i)

With that in mind, we have the following -



class Solution {

public:

int numTrees(int n) {

if(n <= 1) return 1;

int ans = 0;

for(int i = 1; i <= n; i++)

ans += numTrees(i-1) \* numTrees(n-i);

return ans;

}

};

**The base condition could also change from n <= 1 to n < 1, since we only need empty tree = 1 as base condition**

class Solution {

public:

int numTrees(int n) {

if(n < 1) return 1;

int ans = 0;

for(int i = 1; i <= n; i++)

ans += numTrees(i-1) \* numTrees(n-i);

return ans;

}

};

Time Complexity : O(3^N), where N is the given number of nodes in BST. Read here for proof.

Space Complexity : O(N), the maximum recursive stack depth.

✔️ **Solution - II (Dynamic Programming - Memoization)**

The above approach times out due to lots of unnecessary repeated calculation.

f(i) = numTrees(i)

f(5)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

↙ ↓ ↓ ↓ ↘

(f(0)\* f(4)) f(1)\*f(3) f(2)\*f(2) f(3)\*f(1) f(4)\*f(0)

\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_ ⬆️ ⬆️ ⬆️ ⬆️ ⬆️

↙ ↓ ↓ ↘

f(0)f(3) f(1)f(2) f(2)f(1) f(3)f(0)

\_\_\_\_\_\_|\_\_\_\_\_ ⬆️ ⬆️ ⬆️

↙ ↓ ↘

f(0)f(2) f(1)f(1) f(2)f(1)

\_\_|\_\_ ⬆️

↙ ↘

f(0)f(1) f(1)f(0)

In the above diagram, drawing out even the partial recursion tree for the above approach, we can find that there are many redundant repeated calculations. We can instead store or memoize these result and later avoid repeated calculations over and over again.

The approach and code will be very similar. The only change is every time we calculate the result for numTrees(i), we store the result in dp[i] and only then return it. After that, each time we encounter dp[i] that's already calculated, we can directly return the result. This way, we won't solve for the same numTrees(i) multiple times.

class Solution {

public:

int dp[20]{};

int numTrees(int n) {

if(n <= 1) return 1;

if(dp[n]) return dp[n];

for(int i = 1; i <= n; i++)

dp[n] += numTrees(i-1) \* numTrees(n-i);

return dp[n];

}

};

**The base condition could also change from n <= 1 to n < 1, since we only need empty tree = 1 as base condition**

class Solution {

public:

int dp[20]{};

int numTrees(int n) {

if(n < 1) return 1;

if(dp[n]) return dp[n];

for(int i = 1; i <= n; i++)

dp[n] += numTrees(i-1) \* numTrees(n-i);

return dp[n];

}

};

Time Complexity : O(N^2)

Here we calculate numTrees(i) (for 1<=i<=N) only once and memoize it which will take O(N). For calculating each of numTrees(i), we need N iterations to calculate numTrees(0) \* numTrees(i) + numTrees(1) \* numTrees(i-1) + numTrees(2) \* numTrees(i-2)+ ... + numTrees(i) \* numTrees(0)**.** Thus, the overall time complexity becomes O(N \* N).

Space Complexity : O(N), required for recursion and memoization

✔️ **Solution - III (Dynamic Programming - Tabulation)**

We can also solve it using iterative dynamic programming. Again, the logic is similar to above with slight change in approach that we start from base conditions instead of other way around.

We have base conditions of dp[0] = dp[1] = 1.

Then we calculate result for each number of nodes i from 2...n one after another.

For i nodes. we can consider each of the node j from 1...i as the root node of BST.

Considering the jth node as the root node in BST having total of i nodes, the final result is summation of dp[j-1] \* dp[i-j], for all j from 1...i. (Comparing to above solution dp[j-1] = numTrees(j-1) and dp[i-j]=numTrees(i-j))

class Solution {

public:

int numTrees(int n) {

vector<int> dp(n+1);

dp[0] = dp[1] = 1;

for(int i = 2; i <= n; i++)

for(int j = 1; j <= i; j++)

dp[i] += dp[j-1] \* dp[i-j];

return dp[n];

}

};

**Actually if only define dp[0] = 1, NO need dp[1] = 1 is also fine, which exactly only require mapping to empty tree condition**

**Then we calculate result for each number of nodes i from 1...n one after another instead from 2 ...n.**

class Solution {

public:

    int numTrees(int n) {

        vector<int> dp(n+1);

        dp[0] = 1;

        for(int i = 1; i <= n; i++)

            for(int j = 1; j <= i; j++)

                dp[i] += dp[j-1] \* dp[i-j];

        return dp[n];

    }

};

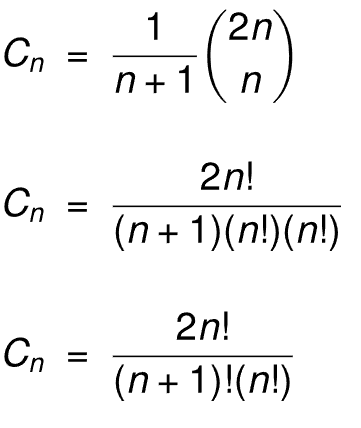
Time Complexity : O(N^2), we iterate over the range i=[2, n] and iteratively calculate dp[i]. The total number of operations performed equals 2+3+4+5..n = (n\*(n+1)/2)-1 ≈ O(N^2)

Space Complexity : O(N), required to store the results in dp

✔️ **Solution - IV (Catalan Numbers)**

Observing the series we get above for various numTrees(n), we see that it is infact a series of popular numbers known as Catalan Numbers. This approach is hard to get unless you are already familiar with **catalan numbers** and probably wont be expected in interview either. But I am mentioning this approach as another possible and more efficient solution to this question.

We can use the following formula for calculating catalan numbers Cn to calculate the result in O(N) time complexity -



We will use 1st equation of above image with binomial coefficient function - ncr in C++ to avoid overflow. In python, we can directly calculate factorial as it can handle long numbers

class Solution {

public:

long ncr(int n, int r) {

long ans = 1;

for(int i = 0; i < r; i++) {

ans \*= n-i;

ans /= i+1;

}

return ans;

}

int numTrees(int n) {

return ncr(2\*n, n) / (n + 1);

}

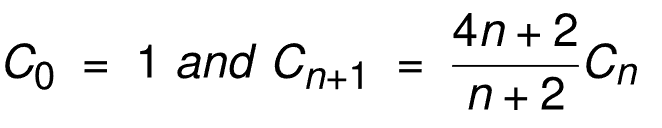
};

Time Complexity : O(N) The ncr function runs in O(N) time. In python, the factorial function takes O(N) time as well.

Space Complexity : O(1)

✔️ **Solution - V (Catalan Numbers - 2)**

The Catalan Numbers also follow the below recurrence relation -



This can be said to be a kind of dynamic programming approach where our next result depends only on previous one. This is slightly easier to implement in code than the 1st formula for catalan numbers.

class Solution {

public:

int numTrees(int n) {

long ans = 1;

for(int i = 0; i < n; i++)

ans \*= (4\*i+2) / (i+2.);

return ans;

}

};

Time Complexity : O(N)

Space Complexity : O(1)

**Refer to**

<https://algo.monster/liteproblems/96>

**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:

Consider that the number of unique BSTs with 0 nodes (an empty tree) is 1. This serves as our base case.

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.

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.

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:

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].

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.

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).

**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 (dp[j]'s coming from j-1 - 0 + 1 = j) and dp[i - j - 1] ways to construct the right subtree from j+1 to i - 1(dp[i - j - 1]'s i - j - 1 coming from (i - 1) - (j+1) + 1 = i - j - 1, and be careful, its not i but i - 1, the i - 1 is coming from the 'root' defined from 0 to '< i', the maximum 'root' is 'i - 1'). 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.**

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:

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].

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].

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.

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.

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.

**Solution Implementation**

class Solution {

// Function to compute number of unique BSTs with n nodes

public int numTrees(int n) {

// Initialise the dp array to store the number of unique BSTs for each count of nodes

// dp[i] gives the number of unique BSTs that can be formed with i 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];

}

}

**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).

**Refer to**

[L95.Unique Binary Search Trees II (Ref.L241,L96)](note://WEBf95fafbd7e21c2466d68ef518b0685d4)