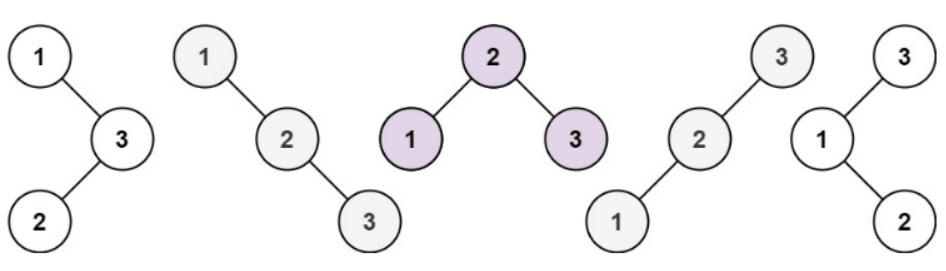
<https://leetcode.com/problems/unique-binary-search-trees/>

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: 2023-01-20**

**Solution 1: Native DFS (10 min, TLE)**

class Solution {

public int numTrees(int n) {

if(n == 0 || n == 1) {

return 1;

}

int result = 0;

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

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

}

return result;

}

}

Time Complexity : O(3^N)

Space Complexity: O(N)

**Solution 2: Top Down DP Memoization (10 min)**

class Solution {

public int numTrees(int n) {

Integer[] memo = new Integer[20];

return helper(n, memo);

}

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

if(n == 0 || n == 1) {

return 1;

}

if(memo[n] != null) {

return memo[n];

}

int result = 0;

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

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

}

memo[n] = result;

return result;

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Solution 3: Bottom Up DP (10 min)**

class Solution {

public int numTrees(int n) {

int[] dp = new int[20];

dp[0] = 1;

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)

Space Complexity: 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)

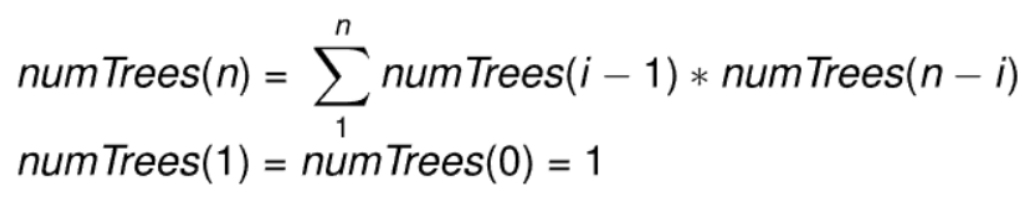
i = 3 i = 2 i = 1

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;

}

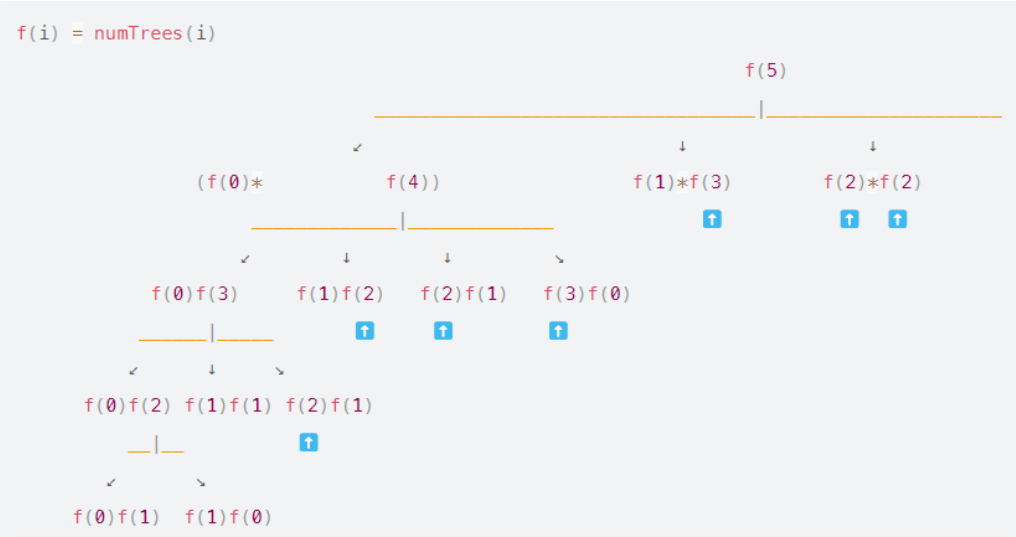
};

***Time Complexity :*** **O(3^N)**, where N is the given number of nodes in BST. Read **[here](https://stackoverflow.com/questions/27371612/catalan-numbers-recursive-function-time-complexity)** 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.



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];

}

};

***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];

}

};

***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(N2)

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

**Refer to**

<https://leetcode.com/problems/unique-binary-search-trees/solutions/31666/dp-solution-in-6-lines-with-explanation-f-i-n-g-i-1-g-n-i/>

The problem can be solved in a dynamic programming way. I’ll explain the intuition and formulas in the following.

Given a sequence 1…n, to construct a Binary Search Tree (BST) out of the sequence, we could enumerate each number i in the sequence, and use the number as the root, naturally, the subsequence 1…(i-1) on its left side would lay on the left branch of the root, and similarly the right subsequence (i+1)…n lay on the right branch of the root. We then can construct the subtree from the subsequence recursively. Through the above approach, we could ensure that the BST that we construct are all unique, since they have unique roots.

The problem is to calculate the number of unique BST. To do so, we need to define two functions:

G(n): the number of unique BST for a sequence of length n.

F(i, n), 1 <= i <= n: the number of unique BST, where the number i is the root of BST, and the sequence ranges from 1 to n.

As one can see, G(n) is the actual function we need to calculate in order to solve the problem. And G(n) can be derived from F(i, n), which at the end, would recursively refer to G(n).

First of all, given the above definitions, we can see that the total number of unique BST G(n), is the sum of BST F(i) using each number i as a root.

*i.e.*

G(n) = F(1, n) + F(2, n) + ... + F(n, n).

Particularly, the bottom cases, there is only one combination to construct a BST out of a sequence of length 1 (only a root) or 0 (empty tree).

*i.e.*

G(0)=1, G(1)=1.

Given a sequence 1…n, we pick a number i out of the sequence as the root, then the number of unique BST with the specified root F(i), is the cartesian product of the number of BST for its left and right subtrees. For example, F(3, 7): the number of unique BST tree with number 3 as its root. To construct an unique BST out of the entire sequence [1, 2, 3, 4, 5, 6, 7] with 3 as the root, which is to say, we need to construct an unique BST out of its left subsequence [1, 2] and another BST out of the right subsequence [4, 5, 6, 7], and then combine them together (*i.e.* cartesian product). The tricky part is that we could consider the number of unique BST out of sequence [1,2] as G(2), and the number of of unique BST out of sequence [4, 5, 6, 7] as G(4). Therefore, F(3,7) = G(2) \* G(4).

*i.e.*

F(i, n) = G(i-1) \* G(n-i) 1 <= i <= n

Combining the above two formulas, we obtain the recursive formula for G(n). *i.e.*

G(n) = G(0) \* G(n-1) + G(1) \* G(n-2) + … + G(n-1) \* G(0)

In terms of calculation, we need to start with the lower number, since the value of G(n) depends on the values of G(0) … G(n-1).

With the above explanation and formulas, here is the implementation in Java.

public int numTrees(int n) {

int [] G = new int[n+1];

G[0] = G[1] = 1;

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

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

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

}

}

return G[n];

}