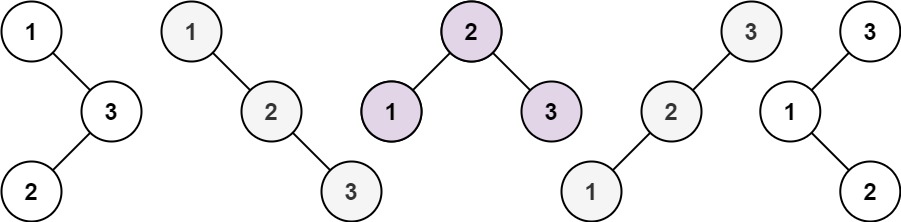
<https://leetcode.com/problems/unique-binary-search-trees-ii/description/>

Given an integer n, return all the structurally unique BST's (binary search trees), which has exactly n nodes of unique values from 1 to n. Return the answer in any order.

**Example 1:**



**Input:** n = 3

**Output:** [[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]

**Example 2:**

**Input:** n = 1

**Output:** [[1]]

**Constraints:**

1 <= n <= 8

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

**Solution 1: DFS + Divide and Conquer (360 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public List<TreeNode> generateTrees(int n) {

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

        if (n == 0) {

            return ans;

        }

        return getAns(1, n);

    }

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

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

        //此时没有数字，将 null 加入结果中

        if (start > end) {

            ans.add(null);

            return ans;

        }

        //只有一个数字，当前数字作为一棵树加入结果中(实际测试后发现并不需要这个条件)

        //if (start == end) {

        //    TreeNode tree = new TreeNode(start);

        //    ans.add(tree);

        //    return ans;

        //}

        //尝试每个数字作为根节点

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

            //得到所有可能的左子树

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

            //得到所有可能的右子树

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

            //左子树右子树两两组合

            for (TreeNode leftTree : leftTrees) {

                for (TreeNode rightTree : rightTrees) {

                    TreeNode root = new TreeNode(i);

                    root.left = leftTree;

                    root.right = rightTree;

                    //加入到最终结果中

                    ans.add(root);

                }

            }

        }

        return ans;

    }

}

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 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 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 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.

**Refer to**

<https://algo.monster/liteproblems/95>

**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:

We know that the root can be any node with a value from 1 to n.

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.

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.

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:

**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).

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

**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.

**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.

**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.

**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.

**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.

**Solution Implementation**

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;

}

}

**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 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 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 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.

**In the solution for Leetcode 95 "Unique Binary Search Trees II", the base condition if (start > end) followed by ans.add(null) is crucial for correctly constructing the binary search trees (BSTs).**

**Let's break down the reasoning behind this condition and its role in the overall solution.**

**Refer to chatGPT**

**Problem Context**

The problem requires generating all unique BSTs that store values from 1 to n. Each BST must satisfy the properties of a binary search tree, where:

The left subtree of a node contains only nodes with values less than the node's value.

The right subtree of a node contains only nodes with values greater than the node's value.

**Explanation of start > end Base Condition**

The recursive function getAns(int start, int end) generates all unique BSTs for the range [start, end]. Here's the breakdown:

Base Condition **start > end**:

When start > end, it means the current range is invalid or there's no number to include in the subtree. In this case, the only possible "tree" is an empty tree, which is represented by null.

Adding null to the list of trees (ans.add(null)) ensures that the recursion properly handles cases where a subtree should be empty.

Recursive Construction of Trees:

For each number i in the range [start, end], i is considered as the root.

All possible left subtrees are generated by recursively calling getAns(start, i - 1).

All possible right subtrees are generated by recursively calling getAns(i + 1, end).

Each combination of left and right subtrees is paired with the root i to form a complete tree, which is then added to the result list ans.

**Detailed Explanation with Example**

Consider n = 3 and we are generating trees for the range [1, 3]:

Initial Call:

generateTrees(3) calls getAns(1, 3).

Base Condition Handling:

getAns(1, 0), getAns(2, 1), etc., will eventually reach the base condition start > end, and will add null to ans.

Recursive Tree Construction:

For i = 1 as the root:

Left subtrees from getAns(1, 0) -> [null].

Right subtrees from getAns(2, 3):

For i = 2 as the root:

Left subtrees from getAns(2, 1) -> [null].

Right subtrees from getAns(3, 3) -> [TreeNode(3)].

Combine to form trees with root 2 and right child 3.

For i = 3 as the root:

Left subtrees from getAns(2, 2) -> [TreeNode(2)].

Right subtrees from getAns(4, 3) -> [null].

Combine to form trees with root 3 and left child 2.

**Why ans.add(null) is Important**

Without ans.add(null), the recursion wouldn't correctly represent cases where a subtree should be empty. For example, when constructing trees with root i where i is at the boundary of the range, the left or right subtree might be empty. This would break the construction of the BSTs, as it wouldn't account for the possibility of nodes having null children.

**Conclusion**

The base condition if (start > end) { ans.add(null); return ans; } ensures that the recursive function can handle cases where a subtree should be empty, allowing the correct construction of all possible unique BSTs within the specified range. This approach ensures that all combinations of left and right subtrees are correctly paired with each possible root, resulting in the complete set of unique BSTs

**What's the time complexity ?**

**Refer to chatGPT**

The time complexity of the given solution to generate all unique Binary Search Trees (BSTs) for values from 1 to n is quite complex to analyze due to the recursive nature of the problem and the combinatorial explosion of the number of trees as n increases.

However, we can provide an explanation and an estimate of the complexity based on the structure of the algorithm.

**Overview**

The algorithm involves recursively generating all possible BSTs by considering each value in the range [start, end] as a root and recursively generating all possible left and right subtrees. This results in a large number of recursive calls and combinations.

**Detailed Complexity Analysis**

Number of Trees:

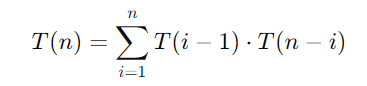
The number of unique BSTs that can be formed with n nodes is given by the nth Catalan number, which is approximately C(n) ~ 4^n / (n^(3/2) \* sqrt(π)).

The nth Catalan number grows exponentially, meaning the number of unique BSTs increases very rapidly with n.

Recursive Calls:

For each node considered as a root, the algorithm recursively generates all possible left and right subtrees.

If T(n) is the number of trees generated for n nodes, we can express it as:



This represents the combinatorial nature of the problem, where each combination of left and right subtrees needs to be considered for each possible root.

Time Complexity:

The overall time complexity is dominated by the number of unique BSTs, which is given by the Catalan number.

Additionally, for each tree, constructing and combining the left and right subtrees involves further recursive calls and combinations.

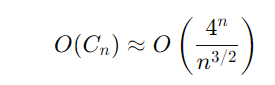
Construction and Combining:

Constructing a single tree involves selecting a root, and then recursively generating and combining all possible left and right subtrees.

For each combination, the algorithm creates a new tree node and sets its left and right children, which adds additional overhead.

**Overall Time Complexity**

Given the combinatorial explosion and the recursive nature of the problem, the time complexity of this algorithm is:



where C\_n is the nth Catalan number.

**Space Complexity**

The space complexity of the algorithm is also high due to the following reasons:

Recursive Call Stack:

The depth of the recursion can go up to n, resulting in a space complexity of O(n) for the call stack.

Storing Results:

The algorithm stores all unique BSTs in a list, and each tree can have up to n nodes.

Storing all trees requires O(C\_n \* n) space, where C\_n is the nth Catalan number and n is the maximum number of nodes per tree.

**Conclusion**

Time Complexity: O(C\_n), where C\_n is the nth Catalan number, approximately O(4^n / n^(3/2)).

Space Complexity: O(C\_n \* n), considering both the recursive stack and the space needed to store the trees.

The complexity reflects the highly combinatorial nature of the problem and the exponential growth in the number of unique BSTs as n increases.

**Refer to**

<https://leetcode.wang/leetCode-95-Unique-Binary-Search-TreesII.html>

解法一完全没有用到查找二叉树的性质，暴力尝试了所有可能从而造成了重复。我们可以利用一下查找二叉树的性质。左子树的所有值小于根节点，右子树的所有值大于根节点。

所以如果求 1...n 的所有可能。

我们只需要把 1 作为根节点，[ ] 空作为左子树，[ 2 ... n ] 的所有可能作为右子树。

2 作为根节点，[ 1 ] 作为左子树，[ 3...n ] 的所有可能作为右子树。

3 作为根节点，[ 1 2 ] 的所有可能作为左子树，[ 4 ... n ] 的所有可能作为右子树，然后左子树和右子树两两组合。

4 作为根节点，[ 1 2 3 ] 的所有可能作为左子树，[ 5 ... n ] 的所有可能作为右子树，然后左子树和右子树两两组合。

...

n 作为根节点，[ 1... n ] 的所有可能作为左子树，[ ] 作为右子树。

至于，[ 2 ... n ] 的所有可能以及 [ 4 ... n ] 以及其他情况的所有可能，可以利用上边的方法，把每个数字作为根节点，然后把所有可能的左子树和右子树组合起来即可。

如果只有一个数字，那么所有可能就是一种情况，把该数字作为一棵树。而如果是 [ ]，那就返回 null。

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public List<TreeNode> generateTrees(int n) {

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

        if (n == 0) {

            return ans;

        }

        return getAns(1, n);

    }

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

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

        //此时没有数字，将 null 加入结果中

        if (start > end) {

            ans.add(null);

            return ans;

        }

        //只有一个数字，当前数字作为一棵树加入结果中(实际测试后发现并不需要这个条件)

        //if (start == end) {

        //    TreeNode tree = new TreeNode(start);

        //    ans.add(tree);

        //    return ans;

        //}

        //尝试每个数字作为根节点

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

            //得到所有可能的左子树

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

            //得到所有可能的右子树

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

            //左子树右子树两两组合

            for (TreeNode leftTree : leftTrees) {

                for (TreeNode rightTree : rightTrees) {

                    TreeNode root = new TreeNode(i);

                    root.left = leftTree;

                    root.right = rightTree;

                    //加入到最终结果中

                    ans.add(root);

                }

            }

        }

        return ans;

    }

}

**或者另一种写法**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public List<TreeNode> generateTrees(int n) {

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

        if(n == 0) {

            return result;

        }

        return helper(1, n);

    }

    private List<TreeNode> helper(int lo, int hi) {

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

        // Base case

        // Refer to

        // https://www.youtube.com/watch?v=GZ0qvkTAjmw

        /\*\*

          Why the base case is lo > hi ?

          Because when we reach to the leave node and you still call

          recursive helper() method, the boundary will become below:

          e.g i = 5 -> left = helper(5,4), right = helper(6,5)

          hence lo > hi always the terminate condition

        \*/

        if(lo > hi) {

            return list;

        }

        for(int i = lo; i <= hi; i++) {

            List<TreeNode> left = helper(lo, i - 1);

            List<TreeNode> right = helper(i + 1, hi);

            // Create root should in each block since for loop

            // in the block means create multiple root required

            // TreeNode root = new TreeNode(i);

            if(left.size() == 0 && right.size() == 0) {

                TreeNode root = new TreeNode(i);

                list.add(root);

            } else if(right.size() == 0) {

                for(TreeNode l : left) {

                    TreeNode root = new TreeNode(i);

                    root.left = l;

                    list.add(root);

                }

            } else if(left.size() == 0) {

                for(TreeNode r : right) {

                    TreeNode root = new TreeNode(i);

                    root.right = r;

                    list.add(root);

                }

            } else {

                for(TreeNode l : left) {

                    for(TreeNode r : right) {

                        TreeNode root = new TreeNode(i);

                        root.left = l;

                        root.right = r;

                        list.add(root);

                    }

                }

            }

        }

        return list;

    }

}

**Refer to**

[L96.Unique Binary Search Trees (Ref.L95)](note://WEB9cd2e136dd2ed5ec4ad122a8ef93d3d4)

[L241.Different Ways to Add Parentheses (Ref.L95)](note://WEBbaa2d082a1dad88f29dcca85766d9625)