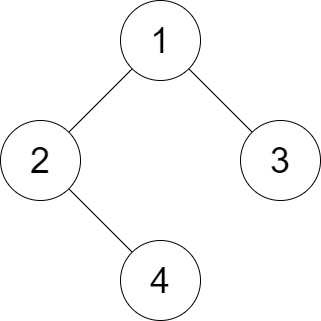
<https://leetcode.com/problems/number-of-good-leaf-nodes-pairs/description/>

You are given the root of a binary tree and an integer distance. A pair of two different **leaf** nodes of a binary tree is said to be good if the length of **the shortest path** between them is less than or equal to distance.

Return **the number of good leaf node pairs** in the tree.

**Example 1:**

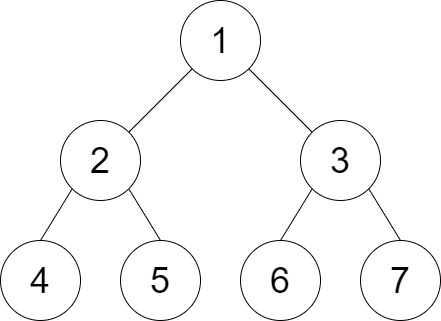


**Input:** root = [1,2,3,null,4], distance = 3

**Output:** 1

**Explanation:** The leaf nodes of the tree are 3 and 4 and the length of the shortest path between them is 3. This is the only good pair.

**Example 2:**



**Input:** root = [1,2,3,4,5,6,7], distance = 3

**Output:** 2

**Explanation:** The good pairs are [4,5] and [6,7] with shortest path = 2. The pair [4,6] is not good because the length of ther shortest path between them is 4.

**Example 3:**

**Input:** root = [7,1,4,6,null,5,3,null,null,null,null,null,2], distance = 3

**Output:** 1

**Explanation:** The only good pair is [2,5].

**Constraints:**

The number of nodes in the tree is in the range [1, 210].

1 <= Node.val <= 100

1 <= distance <= 10

**Attempt 1: 2024-01-15**

**Solution 1: Divide and Conquer + Nested Recursion (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 int countPairs(TreeNode root, int distance) {

        if(root == null) {

            return 0;

        }

        int curRootResult = 0;

        // Divide

        int leftResult = countPairs(root.left, distance);

        int rightResult = countPairs(root.right, distance);

        // Process

        // For each node we have to find its left and right counts

        int[] leftCountsOfCurRoot = new int[distance];

        int[] rightCountsOfCurRoot = new int[distance];

        // Theoretically, 'depth' start from 0, but here is 1, why?

        // because that's the distance between current node and its

        // direct left or right child, if current node as root and

        // treat its 'depth = 0', then its direct left or right child

        // has 'depth = 1', the helper() method start a DFS traversal

        // treat current node as root to update its 'leftCountsOfCurRoot'

        // and 'rightCountsOfCurRoot' count by traverse its left and

        // right subtree (left subtree start from root.left, right

        // subtree start from root.right)

        helper(root.left, leftCountsOfCurRoot, 1);

        helper(root.right, rightCountsOfCurRoot, 1);

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

            for(int j = 0; j < distance; j++) {

                if(i + j <= distance) {

                    curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

                }

            }

        }

        // Conquer

        return curRootResult + leftResult + rightResult;

    }

    private void helper(TreeNode root, int[] counts, int depth) {

        // If we've reached a null node, or exceeded the array length,

        // there's nothing to do

        if(root == null || depth >= counts.length) {

            return;

        }

        // If it's a leaf node, increment the count for this depth

        if(root.left == null && root.right == null) {

            counts[depth]++;

            return;

        }

        // Otherwise, recursively call for the left and right children,

        // increasing the depth.

        helper(root.left, counts, depth + 1);

        helper(root.right, counts, depth + 1);

    }

}

Time Complexity: O(n \* distance^2)

Space ComplexityL O()

**Test code**

public class TreeSolution {

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;

}

}

public int countPairs(TreeNode root, int distance) {

if(root == null) {

return 0;

}

int curRootResult = 0;

// Divide

int leftResult = countPairs(root.left, distance);

int rightResult = countPairs(root.right, distance);

// Process

// For each node we have to find its left and right counts

int[] leftCountsOfCurRoot = new int[distance];

int[] rightCountsOfCurRoot = new int[distance];

// Theoretically, 'depth' start from 0, but here is 1, why?

// because that's the distance between current node and its

// direct left or right child, if current node as root and

// treat its 'depth = 0', then its direct left or right child

// has 'depth = 1', the helper() method start a DFS traversal

// treat current node as root to update its 'leftCountsOfCurRoot'

// and 'rightCountsOfCurRoot' count by traverse its left and

// right subtree (left subtree start from root.left, right

// subtree start from root.right)

helper(root.left, leftCountsOfCurRoot, 1);

helper(root.right, rightCountsOfCurRoot, 1);

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

for(int j = 0; j < distance; j++) {

if(i + j <= distance) {

curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

}

}

}

// Conquer

return curRootResult + leftResult + rightResult;

}

private void helper(TreeNode root, int[] counts, int depth) {

// If we've reached a null node, or exceeded the array length,

// there's nothing to do

if(root == null || depth >= counts.length) {

return;

}

// If it's a leaf node, increment the count for this depth

if(root.left == null && root.right == null) {

counts[depth]++;

return;

}

// Otherwise, recursively call for the left and right children,

// increasing the depth.

helper(root.left, counts, depth + 1);

helper(root.right, counts, depth + 1);

}

public static void main(String[] args) {

/\*\*

\* 1

\* / \

\* 2 3

\* / \ / \

\* 4 5 6 7

\*/

TreeSolution so = new TreeSolution();

TreeNode one = so.new TreeNode(1);

TreeNode two = so.new TreeNode(2);

TreeNode three = so.new TreeNode(3);

TreeNode four = so.new TreeNode(4);

TreeNode five = so.new TreeNode(5);

TreeNode six = so.new TreeNode(6);

TreeNode seven = so.new TreeNode(7);

one.left = two;

one.right = three;

two.left = four;

two.right = five;

three.left = six;

three.right = seven;

int result = so.countPairs(one, 3);

System.out.println(result);

}

}

**Step by Step**

/\*\*

 \*                1

 \*           /         \

 \*        2              3

 \*      /   \          /   \

 \*     4     5        6     7

 \*/

1. root = 4

root.left = null

root.right = null

4's leftCountsOfCurRoot = [0,0,0]

4's rightCountsOfCurRoot = [0,0,0]

helper(root.left, leftCountsOfCurRoot, 1) -> [0,0,0]

helper(root.right, rightCountsOfCurRoot, 1) -> [0,0,0]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=0) + leftResult(=0) + rightResult(=0) = 0

---------------------------------------------------------------

2. root = 5

root.left = null

root.right = null

5's leftCountsOfCurRoot = [0,0,0]

5's rightCountsOfCurRoot = [0,0,0]

helper(root.left, leftCountsOfCurRoot, 1) -> [0,0,0]

helper(root.right, rightCountsOfCurRoot, 1) -> [0,0,0]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=0) + leftResult(=0) + rightResult(=0) = 0

---------------------------------------------------------------

3. root = 2

root.left = 4

root.right = 5

2's leftCountsOfCurRoot = [0,0,0]

2's rightCountsOfCurRoot = [0,0,0]

helper(root.left, leftCountsOfCurRoot, 1)

4 is leaf node, counts[depth]++ -> counts[1]++ -> [0,1,0]

helper(root.right, rightCountsOfCurRoot, 1)

5 is leaf node, counts[depth]++ -> counts[1]++ -> [0,1,0]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=1) + leftResult(=0) + rightResult(=0) = 1

---------------------------------------------------------------

4. root = 6

root.left = null

root.right = null

6's leftCountsOfCurRoot = [0,0,0]

6's rightCountsOfCurRoot = [0,0,0]

helper(root.left, leftCountsOfCurRoot, 1) -> [0,0,0]

helper(root.right, rightCountsOfCurRoot, 1) -> [0,0,0]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=0) + leftResult(=0) + rightResult(=0) = 0

---------------------------------------------------------------

5. root = 7

root.left = null

root.right = null

7's leftCountsOfCurRoot = [0,0,0]

7's rightCountsOfCurRoot = [0,0,0]

helper(root.left, leftCountsOfCurRoot, 1) -> [0,0,0]

helper(root.right, rightCountsOfCurRoot, 1) -> [0,0,0]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=0) + leftResult(=0) + rightResult(=0) = 0

---------------------------------------------------------------

6. root = 3

root.left = 6

root.right = 7

3's leftCountsOfCurRoot = [0,0,0]

3's rightCountsOfCurRoot = [0,0,0]

helper(root.left, leftCountsOfCurRoot, 1)

6 is leaf node, counts[depth]++ -> counts[1]++ -> [0,1,0]

helper(root.right, rightCountsOfCurRoot, 1)

7 is leaf node, counts[depth]++ -> counts[1]++ -> [0,1,0]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=1) + leftResult(=0) + rightResult(=0) = 1

---------------------------------------------------------------

7. root = 1

root.left = 2

    root.left = 4

    root.left = 5

root.right = 3

    root.left = 6

    root.left = 7

1's leftCountsOfCurRoot = [0,0,0]

1's rightCountsOfCurRoot = [0,0,0]

helper(root.left(=2), leftCountsOfCurRoot, 1)

    helper(root.left(=4), leftCountsOfCurRoot, 2)

    4 is leaf node, counts[depth]++ -> counts[2]++ -> [0,0,1]

    helper(root.right(=5), leftCountsOfCurRoot, 2)

    5 is leaf node, counts[depth]++ -> counts[2]++ -> [0,0,2]

helper(root.right(=3), leftCountsOfCurRoot, 1)

    helper(root.left(=6), leftCountsOfCurRoot, 2)

    6 is leaf node, counts[depth]++ -> counts[2]++ -> [0,0,1]

    helper(root.right(=7), leftCountsOfCurRoot, 2)

    7 is leaf node, counts[depth]++ -> counts[2]++ -> [0,0,2]

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

    for(int j = 0; j < distance; j++) {

        if(i + j + 1 < distance) {

            curRootResult += leftCountsOfCurRoot[i] \* rightCountsOfCurRoot[j];

        }

    }

}

curRootResult(=0) + leftResult(=1) + rightResult(=1) = 2

**Refer to**

<https://algo.monster/liteproblems/1530>

**Problem Description**

The problem presents a binary tree and an integer distance. The task is to find the number of 'good' leaf node pairs in the tree. A leaf node pair is considered 'good' if the shortest path between the two leaves is less than or equal to the specified distance. The path length is the number of edges in the shortest path connecting the two leaves.

In simpler terms, you need to:

Traverse the tree to find all leaf nodes.

Determine the shortest path between every pair of leaf nodes.

Count the pairs where the path's length is within the given distance.

**Intuition**

To solve this problem, we apply the depth-first search (DFS) strategy. The main intuition is to search the entire tree starting from the root while keeping track of the depth to reach each leaf node. Once we reach leaf nodes, we can create pairs and check if the sum of depths is less than or equal to the given distance.

Key points that lead to this approach:

DFS is a natural way to explore all paths in a tree.

Since we're interested in leaf nodes, we can ignore any node that is not a leaf once we reach a certain depth greater than distance

We use a Counter to keep track of the number of leaves encountered at each depth. This allows us to efficiently calculate the number of good leaf node pairs.

The process involves the following steps:

If the current node is None, or we've exceeded the distance, we stop the search (base case for DFS).

If the current node is a leaf, we record its depth in the Counter

We perform DFS on both the left and right children of the current node.

**After performing DFS on both subtrees of the root, we have two Counters that contain the depth distribution of leaf nodes for each subtree. 注意是在每个节点都需要2个Counters，而不是仅对root节点这一个节点，这样才保证了当最短路线不需要经过root节点的时候也能在最短路线对应的节点的2个Counters中找到需要的距离**

We iterate over the products of the pairs of counts where the sum of their respective depths is less than or equal to distance

The final answer is the sum of good leaf node pairs between the left and right subtrees of the root and the number of good leaf node pairs found recursively within the left and right subtrees themselves.

**Solution Approach**

The solution employs a Depth-First Search (DFS) algorithm combined with a Counter data structure from Python's collections module. Here's how the code addresses the problem:

The dfs function is a recursive helper function designed to perform DFS on the tree starting from the given node. It takes three parameters: root, cnt, and i, where root is the current node being traversed, cnt is a Counter object keeping track of the number of times a leaf node is encountered at each depth, and i represents the current depth.

def dfs(root, cnt, i):

if root is None or i >= distance:

return

if root.left is None and root.right is None:

cnt[i] += 1

return

dfs(root.left, cnt, i + 1)

dfs(root.right, cnt, i + 1)

The base countPairs function initializes a result variable ans and recursively calls itself on the left and right subtrees of the given root, summing up the number of good leaf pairs found in the subtrees.

ans = self.countPairs(root.left, distance) + self.countPairs(root.right, distance)

Two Counter instances, cnt1 and cnt2, are then initialized to keep track of the depth of leaf nodes in the left and right subtrees, respectively.

cnt1 = Counter()

cnt2 = Counter()

The dfs function is called for both subtrees.

dfs(root.left, cnt1, 1)

dfs(root.right, cnt2, 1)

Two nested loops iterate over the cnt1 and cnt2 Counters, where for each depth k1 in cnt1 and k2 in cnt2, we check if k1 + k2 is less than or equal to distance.

If the condition is satisfied, it means we found good leaf node pairs. We then multiply the counts of the respective depths (v1 and v2) and add them to ans.

for k1, v1 in cnt1.items():

for k2, v2 in cnt2.items():

if k1 + k2 <= distance:

ans += v1 \* v2

Finally, ans is returned as the total number of good leaf node pairs in the tree.

**The use of Counter to keep track of the depths at which leaf nodes occur is a key factor in optimizing the solution. It allows the code to efficiently pair leaf nodes by depth without explicitly calculating the distance between every possible pair of leaf nodes.**

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach. Consider the following binary tree and the distance value of 3:

1

/ \

2 3

/ \ / \

4 5 6 7

All numbered nodes represent individual nodes in the tree, with 4, 5, 6, and 7 being the leaf nodes.

Now, let's apply the solution approach:

We perform a DFS starting from the root (node 1). Since the root is not a leaf, we continue on to its children.

In our dfs implementation, when visiting node 2, we find it is not a leaf so we continue the search to its children. We increment the depth by 1 and visit nodes 4 and 5. As these are leaf nodes, we record their depth in cnt1.

For node 4, at depth 2, cnt1[2] becomes 1.

For node 5, at depth 2, cnt1[2] becomes 2 since another leaf node is encountered at the same depth.

We do the same for the right subtree. In cnt2, we record the depths for leaf nodes 6 and 7.

For node 6, at depth 2, cnt2[2] becomes 1.

For node 7, at depth 2, cnt2[2] becomes 2.

After the DFS is done, we iterate over the counters. Now we check for each k1 in cnt1 and each k2 in cnt2 whether k1 + k2 <= distance. So we check for each pair of leaves from left and right subtrees if the ends meet the distance requirement.

Since cnt1 and cnt2 both have leaf nodes at depth 2, we check if 2 (from cnt1) + 2 (from cnt2) <= 3, which is not true. Therefore, no pair between leaf nodes 4, 5 and leaf nodes 6, 7 is counted.

From points 1 to 4, since the distance from any leaf node in the left subtree to any leaf node in the right subtree exceeds the given distance of 3, no "good" leaf node pair would be added to our result ans in this scenario.

Therefore, with the distance set to 3, there are no good leaf node pairs that satisfy the condition in this binary tree. If the distance were higher, say 4, then pairs like (4, 6), (4, 7), (5, 6), and (5, 7) would be considered good pairs as their path lengths would be equal to the distance.

This example demonstrates the efficiency of the Counter data structure combined with the dfs recursive function. We're able to find leaf node depths and efficiently conclude whether leaf node pairs across different subtrees are within the specified distance without having to assess the actual paths or distances directly.

**Java Solution**

class Solution {

// Method to count the number of good leaf node pairs within the given 'distance'.

public int countPairs(TreeNode root, int distance) {

// Base case: if the tree is empty, there are no pairs.

if (root == null) {

return 0;

}

// Recursively count pairs in the left and right subtrees.

int result = countPairs(root.left, distance) + countPairs(root.right, distance);

// Arrays to hold the count of leaf nodes at each level for the left and right subtrees.

int[] leftCounts = new int[distance];

int[] rightCounts = new int[distance];

// Depth-first traversals to populate counts for the left and right subtrees.

dfs(root.left, leftCounts, 1);

dfs(root.right, rightCounts, 1);

// Now, iterate over all pairs of counts from left and right subtrees.

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

for (int j = 0; j < distance; ++j) {

// If the sum of levels is within the 'distance', then these nodes can form a good pair.

if (i + j + 1 <= distance) {

result += leftCounts[i] \* rightCounts[j];

}

}

}

// Return the total count of good leaf node pairs.

return result;

}

// Helper method to perform a DFS on the tree, populating the 'counts' array with the number of leaves at each level.

void dfs(TreeNode node, int[] counts, int level) {

// If we've reached a null node, or exceeded the array length, there's nothing to do.

if (node == null || level >= counts.length) {

return;

}

// If it's a leaf node, increment the count for this level.

if (node.left == null && node.right == null) {

counts[level]++;

return;

}

// Otherwise, recursively call DFS for the left and right children, increasing the level.

dfs(node.left, counts, level + 1);

dfs(node.right, counts, level + 1);

}

}

/\*\*

\* Definition for a binary tree node.

\*/

class TreeNode {

int val; // the value of the node

TreeNode left; // reference to the left child node

TreeNode right; // reference to the right child node

TreeNode() {}

// Constructor to create a node with a specific value.

TreeNode(int val) {

this.val = val;

}

// Constructor to create a node with specific value, left child, and right child.

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

this.val = val;

this.left = left;

this.right = right;

}

}

**Time and Space Complexity**

The given code consists of a recursive depth-first search (DFS) to traverse a binary tree while counting the number of good leaf node pairs. A 'good' pair is defined as a pair of leaf nodes such that the number of edges between them is less than or equal to the given distance.

**Time Complexity:**

To analyze the time complexity, we can observe that:

**The DFS function, which we'll call dfs, is called recursively for every node in the tree.** In the worst case, if the tree is balanced, there will be O(n) calls since it has to visit each node, where n is the total number of nodes in the tree.

Inside each call to dfs, we increment the count of leaf nodes at a particular depth, which takes O(1) time.

After the dfs calls, we have two nested loops that iterate over the counters cnt1 and cnt2. In the worst case, these counters can be as large as O(distance), because leaf nodes more than distance away from the root are not counted. This results in O(distance^2) time complexity for these loops.

Since these steps are performed for every node, the overall time complexity is O(n \* distance^2).

**Space Complexity:**

For space complexity:

The space used by the recursive call stack for dfs will be O(h), where h is the height of the tree. In the worst case of a skewed tree, this will be O(n).

Additional space is used for the cnt1 and cnt2 counters, which store at most distance elements each. Therefore, the space allocated for the counters is O(distance).

**As dfs is called on every node, and each call has its own counter which could theoretically store up to distance elements, the cumulative space for all dfs calls could be O(n \* distance) in the worse case scenario.** However, in the average balanced tree case, the space would be limited due to overlapping subtree nodes calling dfs. Owing to the fact that we are not keeping counters for all nodes in the tree, but only for leaf nodes, and they will overlap significantly, the worst-case space complexity is smaller than O(n\*distance), but it still scales with both n and distance, making O(n + distance) a conservative estimate.

In the given code, there's an assumption of the existence of a Counter class, which behaves similarly to the Counter class from Python's collections module. If custom logic had been used instead, it could potentially change the space complexity due to the data structure used to implement the counting logic.

**Refer to**

<https://leetcode.com/problems/number-of-good-leaf-nodes-pairs/solutions/756198/java-dfs-solution-with-a-twist-100-faster-explained/>

Steps -

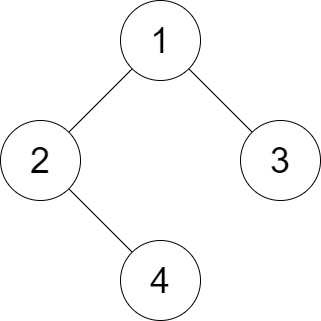
We have to do normal

post order tree traversal.

The trick is to keep track of number of leaf nodes with a particular distance. The arrays are used for this purpose.

For this we maintain an array of size = max distance.

Example -



In above example , assume maximum distance = 4. So we maintain an array of size 4.

For root node 1,

left = [ 0,0,**1**,0,0]

right = [0,**1**,0,0,0]

**Here, left[2] = 1, which denotes that there is one leaf node with distance 2 in left subtree of root node 1.**

**right[1] = 1, which denotes that there is one leaf node with distance 1 in right subtree of root node 1.**

**In this way, we have to recursively, calculate the left and right subtree of every root node.**

**Once we have both left and right arrays for a particular root, we have to just calculate total number of good node pairs formed using result += left[l]\*right[r];**

Before we bactrack to parent, we have to return the distance for parents by adding up left and right subtrees of current node. Note that we are doing - res[i+1] = left[i]+right[i];

The intution is that, if a leaf node is at distance i from current node, it would be at distance i+1 from its parent. Hence, will building the res array, we are adding sum in i+1 th position and return to parent.

class Solution {

int result = 0;

public int countPairs(TreeNode root, int distance) {

dfs(root,distance);

return result;

}

int[] dfs(TreeNode root,int distance){

if(root == null)

return new int[distance+1];

if(root.left == null && root.right == null){

int res[] = new int[distance+1];

res[1]++;

return res;

}

int[] left = dfs(root.left,distance);

int[] right = dfs(root.right,distance);

for(int l=1;l<left.length;l++){

for(int r = distance-1;r>=0;r--){

if(l+r <=distance)

result += left[l]\*right[r];

}

}

int res[] = new int[distance+1];

//shift by 1

for(int i=res.length-2;i>=1;i--){

res[i+1] = left[i]+right[i];

}

return res;

}

}