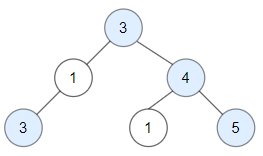
<https://leetcode.com/problems/count-good-nodes-in-binary-tree/description/>

Given a binary tree root, a node **X** in the tree is named **good** if in the path from root to **X** there are no nodes with a value **greater than** X.

Return the number of **good** nodes in the binary tree.

**Example 1:**



**Input:** root = [3,1,4,3,null,1,5]

**Output:** 4

**Explanation:** Nodes in blue are **good**.

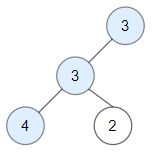
Root Node (3) is always a good node.

Node 4 -> (3,4) is the maximum value in the path starting from the root.

Node 5 -> (3,4,5) is the maximum value in the path

Node 3 -> (3,1,3) is the maximum value in the path.

**Example 2:**



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

**Output:** 3

**Explanation:** Node 2 -> (3, 3, 2) is not good, because "3" is higher than it.

**Example 3:**

**Input:** root = [1]

**Output:** 1

**Explanation:** Root is considered as **good**.

**Constraints:**

The number of nodes in the binary tree is in the range [1, 10^5].

Each node's value is between [-10^4, 10^4].

**Attempt 1: 2024-04-01**

**Solution 1: DFS (10 min, recursive)**

**Style 1: return int**

/\*\*

 \* 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 goodNodes(TreeNode root) {

        return helper(root, root.val);

    }

    private int helper(TreeNode root, int maxVal) {

        if(root == null) {

            return 0;

        }

        int result = 0;

        if(root.val >= maxVal) {

            result++;

        }

        int max = Math.max(maxVal, root.val);

        return result + helper(root.left, max) + helper(root.right, max);

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/count-good-nodes-in-binary-tree/solutions/635258/java-python-3-simple-recursive-and-iterative-dfs-codes-w-brief-explanation-and-analysis/>

**Method 1: recursive version**

Update the maximum value found while recurse down to the paths from root to leaves;

If node value >= current maximum, count it in.

return the total number after the completion of all recursions.

public int goodNodes(TreeNode root) {

return preorder(root, root.val);

}

private int preorder(TreeNode n, int v) {

if (n == null) // base cases.

return 0;

int max = Math.max(n.val, v); // maximum so far on the path.

return (n.val >= v ? 1 : 0) + preorder(n.left, max) + preorder(n.right, max); // recurse to children.

}

**Style 2: void return + global variable**

/\*\*

 \* 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 {

    int count = 0;

    public int goodNodes(TreeNode root) {

        helper(root, root.val);

        return count;

    }

    private void helper(TreeNode root, int maxVal) {

        if(root == null) {

            return;

        }

        if(root.val >= maxVal) {

            count++;

        }

        int max = Math.max(maxVal, root.val);

        helper(root.left, max);

        helper(root.right, max);

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

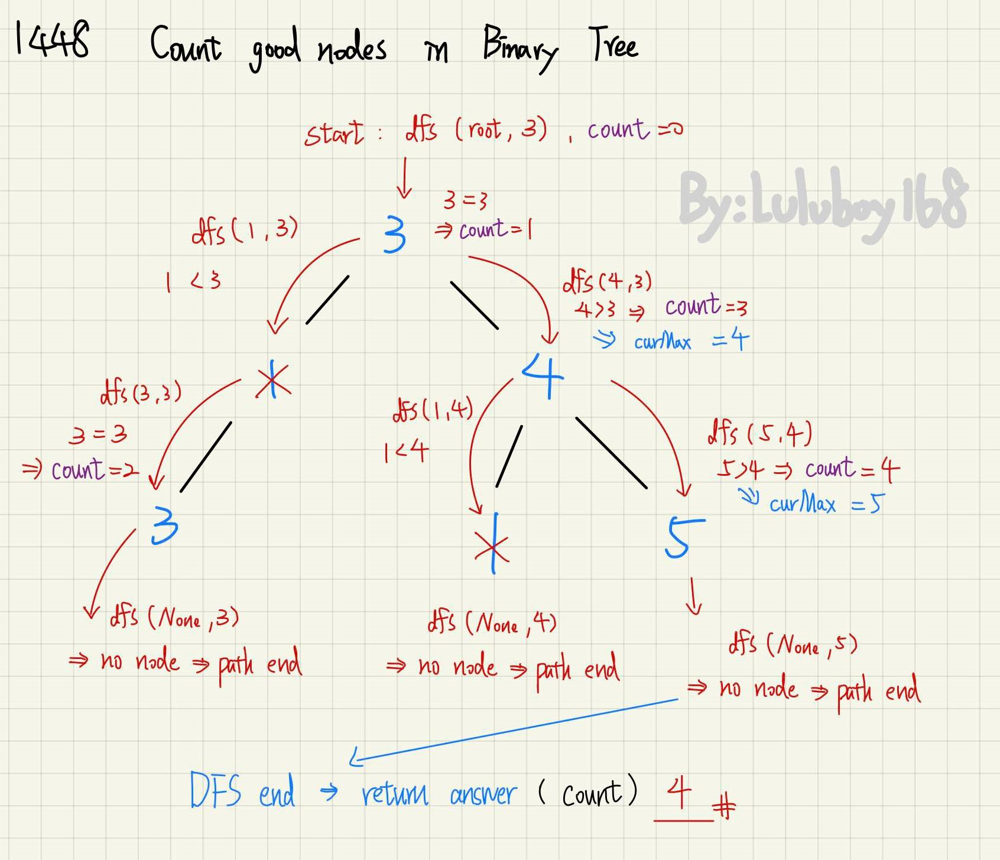
<https://leetcode.com/problems/count-good-nodes-in-binary-tree/solutions/2512547/c-python-c-97-dfs-detailed-graph-explantion-beginner-friendly-easy-to-understand/>

**Idea:**

DFS through every path, and keep tracking of biggest value(curMax) in the path.

If current node is >= the biggest value in the path, we add the answer by one.

Here's an example to show how the code works:



class Solution {

public:

int count = 0;

void dfs(TreeNode\* node,int curMax){

if (!node) return;

if (node->val >= curMax){

count++;

curMax = node->val;

}

dfs(node->left, curMax);

dfs(node->right, curMax);

}

int goodNodes(TreeNode\* root) {

dfs(root, root->val);

return count;

}

};

**Refer to**

<https://algo.monster/liteproblems/1448>

**Problem Description**

The problem requires counting the "good" nodes in a binary tree. A "good" node is defined as a node where, along the path from the root of the tree to that node, there are no nodes with a value greater than the value of the node itself. To simplify, if you start at the root and walk towards the node, every ancestor's value you encounter must be less than or equal to the node's value for it to be considered good. We must traverse the tree and count how many such good nodes exist.

**Intuition**

The key to solving this problem is to maintain the max value encountered along the path from the root to the current node. We perform a depth-first search (DFS) traversal of the tree and carry the maximum value found so far to each node's children. At each node, we do two checks:

If the current node's value is greater than or equal to the max value we've seen so far on this path, it qualifies as a good node, and we increment our count of good nodes. We also update the max value to the current node's value, because it's now the highest value seen on the path for the subtree rooted at this node.

We continue to traverse the tree by going left and right, carrying forward this updated max value to be used for the node's children.

The code defines a recursive dfs function that takes the current node and the current max value as parameters. If the node is None, we've hit a leaf's child, and there's nothing more to do, so we return. If the node is good, we increment our global answer ans by 1 and update the max value if necessary. Then we call dfs on the left and right children, ensuring that we pass the potentially updated max value. The code starts with a max value initialized to a very small number to ensure that the root node is always considered good (since there's no value in the tree less than this initial value).

The global variable ans is used to retain the count of good nodes found during the DFS traversal. After the traversal is completed, ans will store the total number of good nodes, and it's returned by the goodNodes function as the final answer.

**Solution Approach**

The solution implements a Depth-First Search (DFS) algorithm to traverse the tree. DFS is a common tree traversal technique that explores as far as possible along each branch before backtracking. This allows the solution to keep track of the current path's maximum value and check for "good" nodes.

The given Python code defines a nested dfs function within the goodNodes method. The dfs function is responsible for traversing the tree. It is called recursively for the root node initially, with the lowest possible integer value mx set as the initial maximum value (-1000000) encountered along the path. This is to ensure that the root node is always considered as a “good” node, since its value will certainly be higher than this minimum value.

The data structure used here is the given binary tree structure with TreeNode objects. Each TreeNode object contains a value val, and two pointers, left and right, pointing to its child nodes.

Here's what happens in the recursive dfs function:

The function receives a TreeNode object and the current path's max value mx.

It first checks if the current node is None. If so, the recursion ends (base case).

If the current node is not None, it checks if the node's value is greater than or equal to mx. If it is, it increments the counter variable ans by 1 since this node is "good".

The maximum value mx may be updated to the current node's value if it is indeed higher.

The dfs function is then recursively called with the left child of the current node and the updated max value, followed by the recursive call with the right child and the updated max value.

This process continues recursively, visiting all the nodes in the tree, checking each node's value against the maximum value seen so far along that path, and updating the count of good nodes in the global ans variable. After the DFS is completed, ans will hold the count of all good nodes, which is then returned by the goodNodes method.

Here's an example of how the dfs function is defined within the goodNodes method, and how it gets called initially:

def goodNodes(self, root: TreeNode) -> int:

def dfs(root: TreeNode, mx: int):

# rest of the dfs implementation

ans = 0

dfs(root, -1000000) # Start the DFS with the root node and a min initial value

return ans

This DFS pattern, combined with the use of a recursive helper function and a global counter variable, encapsulates the desired logic in a clean and efficient manner to solve the task at hand.

**Example Walkthrough**

Let's run through a small example using a binary tree to illustrate the solution approach outlined. Suppose we have the following binary tree:

3

/ \

1 4

/ / \

3 1 5

We want to count the number of "good" nodes in this tree. A node is "good" if no value greater than that node's value is encountered from the root to that node itself.

We start the DFS with the root node (value 3) and the minimum initial value as mx = -1000000.

Since the root node's value (3) is greater than mx, we count it as a "good" node. The "good" nodes count ans is now 1.

Recursively call DFS on the left child (value 1) with mx = 3 (the value of the root node, since it was larger).

The left child's value (1) is not greater than the current mx = 3, so we do not increment ans. The "good" nodes count remains 1.

Recursively call DFS on the left child's left child (value 3) with mx = 3.

This left child's left child's value (3) is equal to mx, so it's a "good" node. Increment ans to 2.

Since the left child (value 1) has no right child, we backtrack and continue the DFS on the right child of the root (value 4) with mx = 3.

The right child's value (4) is greater than mx, so it's "good". Increment ans to 3 and update mx to 4.

Recursively call DFS on the right child's left child (value 1) with mx = 4.

The right child's left child's value (1) is not greater than mx = 4. We do not increment ans.

Recursively call DFS on the right child's right child (value 5) with mx = 4.

The right child's right child's value (5) is greater than mx, so it's "good". Increment ans to 4 and update mx to 5.

Now that the entire tree has been traversed, we can conclude that there are 4 "good" nodes in this tree.

Hence, the goodNodes method will return 4, which is the total count of good nodes for this example binary tree.

**Java Solution**

// Definition for a binary tree node.

class TreeNode {

    int val;         // Value of the node

    TreeNode left;   // Reference to the left child

    TreeNode right;  // Reference to the right child

    // Constructor for a tree node with no children

    TreeNode() {}

    // Constructor for a tree node with a specific value

    TreeNode(int value) { this.val = value; }

    // Constructor for a tree node with a value and references to left and right children

    TreeNode(int value, TreeNode leftChild, TreeNode rightChild) {

        this.val = value;

        this.left = leftChild;

        this.right = rightChild;

    }

}

public class Solution {

    private int numGoodNodes = 0; // Variable to keep count of good nodes

    // Public method that starts the depth-first search and returns the number of good nodes

    public int goodNodes(TreeNode root) {

        dfsHelper(root, Integer.MIN\_VALUE);

        return numGoodNodes;

    }

    // Helper method that performs a depth-first search on the tree

    private void dfsHelper(TreeNode node, int maxSoFar) {

        if (node == null) {

            return; // Base case: if the node is null, return

        }

        if (maxSoFar <= node.val) {

            // If the current value is greater than or equal to the maximum value so far,

            // it is a good node, so increment the counter and update the maximum value

            numGoodNodes++;

            maxSoFar = node.val;

        }

        dfsHelper(node.left, maxSoFar); // Recursively call helper for left subtree

        dfsHelper(node.right, maxSoFar); // Recursively call helper for right subtree

    }

}

[Time and Space Complexity](https://algo.monster/problems/runtime_summary)

The provided Python code defines a function goodNodes that counts the number of "good" nodes in a binary tree. A "good" node is defined as a node whose value is greater than or equal to all the values in the nodes that lead to it from the root. The function implements a depth-first search (DFS) to traverse the tree and count these nodes.

**Time Complexity**

The time complexity of the code is determined by how many nodes are visited during the DFS traversal. The function visits every node exactly once. Therefore, the time complexity is O(N), where N is the number of nodes in the tree.

**Space Complexity**

The space complexity is primarily determined by the call stack due to the recursive nature of the DFS. In the worst-case scenario (a skewed tree), the depth of the recursive call stack can become O(N) if the tree is a linear chain of N nodes. In the average case of a balanced tree, the height would be O(logN), resulting in O(logN) recursive calls at any given time. However, since we need to consider the worst case, the space complexity of the code is O(N).

**Solution 2: BFS (10 min, iterative with queue)**

/\*\*

 \* 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 Node {

    TreeNode treeNode;

    int curMax;

    public Node(TreeNode treeNode, int curMax) {

        this.treeNode = treeNode;

        this.curMax = curMax;

    }

}

class Solution {

    public int goodNodes(TreeNode root) {

        Queue<Node> q = new LinkedList<>();

        q.offer(new Node(root, root.val));

        int count = 0;

        while(!q.isEmpty()) {

            Node cur = q.poll();

            if(cur.treeNode.val >= cur.curMax) {

                count++;

            }

            int max = Math.max(cur.treeNode.val, cur.curMax);

            if(cur.treeNode.left != null) {

                q.offer(new Node(cur.treeNode.left, max));

            }

            if(cur.treeNode.right != null) {

                q.offer(new Node(cur.treeNode.right, max));

            }

        }

        return count;

    }

}

**Refer to**

<https://leetcode.com/problems/count-good-nodes-in-binary-tree/solutions/635258/java-python-3-simple-recursive-and-iterative-dfs-codes-w-brief-explanation-and-analysis/>

**Method 2: Iterative version**

Combine TreeNode with current path maximum into a Node / tuple, put it into a stack and do iterative DFS to count good nodes.

class Node {

TreeNode tn;

int currentMax = Integer.MIN\_VALUE;

public Node(TreeNode n, int mx) {

tn = n;

currentMax = Math.max(mx, n.val);

}

}

class Solution {

public int goodNodes(TreeNode root) {

Deque<Node> stk = new ArrayDeque<>();

stk.push(new Node(root, root.val));

int cnt = 0;

while (!stk.isEmpty()) {

Node cur = stk.pop();

cnt += cur.tn.val >= cur.currentMax ? 1 : 0;

for (TreeNode kid : new TreeNode[]{cur.tn.left, cur.tn.right}) {

if (kid != null) {

stk.push(new Node(kid, cur.currentMax));

}

}

}

return cnt;

}

}

**Analysis:**

For both methods,

Time: O(n), space: O(h), where n and h are the number and height of the binary tree, respectively.

**Refer to**

<https://leetcode.com/problems/count-good-nodes-in-binary-tree/solutions/635258/java-python-3-simple-recursive-and-iterative-dfs-codes-w-brief-explanation-and-analysis/comments/1232446>

class Solution:

def goodNodes(self, root: TreeNode) -> int:

if root is None:

return 0

ans = 1

st = []

if root.left:

st.append((root.left,root.val))

if root.right:

st.append((root.right,root.val))

while st:

curr, max\_val = st.pop()

if max\_val<=curr.val:

ans+=1

if curr.left:

st.append((curr.left, max(max\_val, curr.val)))

if curr.right:

st.append((curr.right, max(max\_val, curr.val)))

return ans