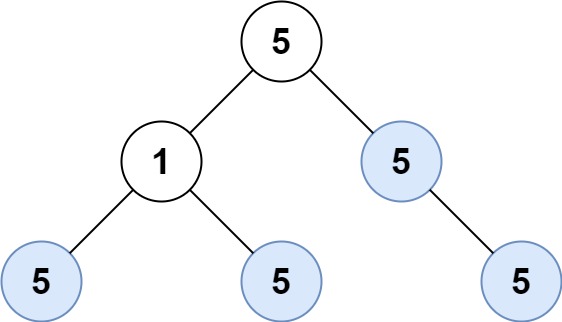
# Question

Given the root of a binary tree, return the number of **uni-value** subtrees.

A **uni-value subtree** means all nodes of the subtree have the same value.

**Example 1:**



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

**Output:** 4

**Example 2:**

**Input:** root = []

**Output:** 0

**Example 3:**

**Input:** root = [5,5,5,5,5,null,5]

**Output:** 6

**Constraints:**

* The numbrt of the node in the tree will be in the range [0, 1000].
* -1000 <= Node.val <= 1000

# Solution

#### **Approach 1: Depth First Search**

**Intuition**

Given a node in our tree, we know that it is a univalue subtree if it meets one of the following criteria:

1. The node has no children (base case)
2. All of the node's children are univalue subtrees, and the node and its children all have the same value

With this in mind we can perform a depth-first-search on our tree, and test if each subtree is uni-value in a bottom-up manner.

1 / 7

**Algorithm**

|  |
| --- |
| public class Solution {  int count = 0;  boolean is\_uni(TreeNode node) {  //base case - if the node has no children this is a univalue subtree  if (node.left == null && node.right == null) {  // found a univalue subtree - increment  count++;  return true;  }  boolean is\_unival = true;  // check if all of the node's children are univalue subtrees and if they have the same value  // also recursively call is\_uni for children  if (node.left != null) {  is\_unival = is\_uni(node.left) && is\_unival && node.left.val == node.val;  }  if (node.right != null) {  is\_unival = is\_uni(node.right) && is\_unival && node.right.val == node.val;  }  // return if a univalue tree exists here and increment if it does  if (!is\_unival) return false;  count++;  return true;  }  public int countUnivalSubtrees(TreeNode root) {  if (root == null) return 0;  is\_uni(root);  return count;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*).

Due to the algorithm's depth-first nature, the is\_uni status of each node is computed from bottom up. When given the is\_uni status of its children, computing the is\_uni status of a node occurs in O(1)*O*(1)

This gives us O(1)*O*(1) time for each node in the tree with O(N)*O*(*N*) total nodes for a time complexity of O(N)*O*(*N*)

* Space complexity : O(H)*O*(*H*), with H being the height of the tree. Each recursive call of is\_uni requires stack space. Since we fully process is\_uni(node.left) before calling is\_uni(node.right), the recursive stack is bound by the longest path from the root to a leaf - in other words the height of the tree.

#### **Approach 2: Depth First Search - Pass Parent Values**

**Algorithm**

We can use the intuition from approach one to further simplify our algorithm. Instead of checking if a node has no children, we treat null values as univalue subtrees that we don't add to the count.

In this manner, if a node has a null child, that child is automatically considered to a valid subtree, which results in the algorithm only checking if other children are invalid.

Finally, the helper function checks if the current node is a valid subtree but returns a boolean indicating if it is a valid component for its parent. This is done by passing in the value of the parent node.

|  |
| --- |
| public class Solution {  int count = 0;  boolean is\_valid\_part(TreeNode node, int val) {  // considered a valid subtree  if (node == null) return true;  // check if node.left and node.right are univalue subtrees of value node.val  // note that || short circuits but | does not - both sides of the or get evaluated with | so we explore all possible routes  if (!is\_valid\_part(node.left, node.val) | !is\_valid\_part(node.right, node.val)) return false;  // if it passed the last step then this a valid subtree - increment  count++;  // at this point we know that this node is a univalue subtree of value node.val  // pass a boolean indicating if this is a valid subtree for the parent node  return node.val == val;  }  public int countUnivalSubtrees(TreeNode root) {  is\_valid\_part(root, 0);  return count;  }  } |

The above code is a commented version of the code [here](https://leetcode.com/problems/count-univalue-subtrees/discuss/67602/Java-11-lines-added), originally written by [Stefan Pochmann](https://leetcode.com/stefanpochmann/).

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*). Same as the previous approach.
* Space complexity : O(H)*O*(*H*), with H being the height of the tree. Same as the previous approach.