250. Count Univalue Subtrees

Feb. 28, 2019 | 28.3K views

Average Rating: 4.19 (21 votes)

Given a binary tree, count the number of uni-value subtrees.

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

Example :

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

              5
             / \
            1   5
           / \   \
          5   5   5

Output: 4
```

Solution

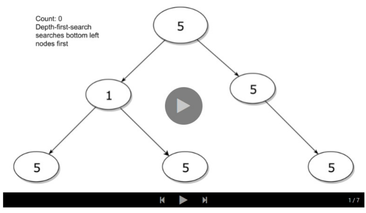
Approach 1: Depth First Search

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.



Algorithm

Java Python Copy

```java
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_univ1 = 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_univ1 = is_uni(node.left) && is_univ1 && node.left.val == node.val;
         }

        if (node.right != null) {
            is_univ1 = is_uni(node.right) && is_univ1 && node.right.val == node.val;
        }

        // return if a univalue tree exists here and increment if it does
        if (!is_univ1) return false;
        count++;
```

Complexity Analysis

- Time complexity : $O(n)$.

  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 $O(1)$

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

- Space complexity : $O(H)$, 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.

Java Python Copy

```java
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, originally written by Stefan Pochmann.

Complexity Analysis

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

Written by @alwinpeng.

Rate this article:

Previous Next

Comments: 11

Sort By

Type comment here... (Markdown is supported)

Preview Post

JAMESJI78 ★ 154 April 10, 2019 7:47 PM

This is really a hard one for me but I'm going to persist :) You only lose if you give up

86 Share Reply

SkandaB ★ 136 March 26, 2020 8:11 PM

Why do most of the solutions in LeetCode articles ***encourage*** use of global variables.
If you are really practicing for interview, you should not be in a habit of using global variables.

You'd rarely be encouraged in real-life software development to use global variables.

2 Share Reply

SHOW 2 REPLIES

socialguy ★ 144 January 1, 2020 12:09 PM

```
def num_univals(node: BinaryTree[T]) -> int:
    if node is None:
        return 0
    if node.left is None and node.right is None:
```

Read More

1 Share Reply

yh32 ★ 31 November 4, 2019 11:56 PM

We can get the uni value from the child node. I think this is easier to understand for me.
if the value is null, null, the child is not a univalue binary tree.
if the value is not null and equals to the current node value, we increment the value and return the current node value.

Read More

1 Share Reply

Gypsophila ★ 62 October 7, 2019 7:05 AM

In solution 1, checking leaf node is redundant.

1 Share Reply

powerrc ★ 12 May 31, 2019 3:21 AM

Is the time complexity really o(n)?

0 Share Reply

SHOW 1 REPLY

flyseeksky ★ 9 February 25, 2020 9:57 AM

There is a small problem with Solution 2: the call of `self.is_valid_part(root, 0)` will yield the right `self.count`: but the return value can be wrong if the `root.val!=0`. For example, a tree `[5,5,5]` will yield a `self.count` of 3, but the return value is `False`, meaning this is not a uni-value tree. Although it does not affect the final answer, this result is inconsistent.

Read More

0 Share Reply

NeosDeus ★ 150 January 13, 2020 8:58 AM

Whoever wrote this article is a pro. It's so much better than most of the articles on here.

0 Share Reply

MrKickass ★ 12 January 6, 2020 6:09 AM

Easy to understand solution, just use a flag to check if left subtree and right subtree are equal. Flag comes in handy when a subtree's values do not match with root but root's parent and root's values match.
Please provide your opinions.

Read More

0 Share Reply

ajay24 ★ 16 October 5, 2019 3:12 PM

Recursive Solution.
Without the global count variable.

```
pair<int, bool> countUnivalTrees(TreeNode* root) {
    if (!root ||(!root->left && !root->right)) return make_pair(0, true);
```

Read More

0 Share Reply