<https://leetcode.ca/all/333.html>

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.

**Note:** A subtree must include all of its descendants.

Example:

Input: [10,5,15,1,8,null,7]

10

/ \

5 15

/ \ \

1 8 7

Output: 3

Explanation: The Largest BST Subtree in this case is the highlighted one.

The return value is the subtree's size, which is 3.

**Follow up:** Can you figure out ways to solve it with O(n) time complexity?

**Attempt 1: 2023-01-02**

**Solution 1: Native recursive traversal two pass DFS O(N^2) solution (30 min, for each node check if a BST start from it, if yes then find number of nodes in in this tree)**

**Style 1: Top Down DFS but with actual return (largestBSTSubtree) + Top Down DFS (isValidBST) + Bottom Up DFS (countNodes)**

public class TreeSolution {

private class TreeNode {

public int val;

public TreeNode left, right;

public TreeNode(int val) {

this.val = val;

this.left = this.right = null;

}

}

public static void main(String[] args) {

TreeSolution s = new TreeSolution();

TreeNode one = s.new TreeNode(1);

TreeNode two = s.new TreeNode(2);

TreeNode three = s.new TreeNode(3);

TreeNode four = s.new TreeNode(4);

TreeNode five = s.new TreeNode(5);

TreeNode six = s.new TreeNode(6);

TreeNode seven = s.new TreeNode(7);

TreeNode eight = s.new TreeNode(8);

/\*\*

\* 2

\* / \

\* 1 3

\* / \

\* 4 5

\*/

two.left = one;

two.right = three;

three.left = four;

three.right = five;

int result = s.largestBSTSubtree(two);

System.out.println(result);

}

// Top Down DFS (遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题)

public int largestBSTSubtree(TreeNode root) {

// Base

if(root == null) {

return 0;

}

// 进行当前层的处理计算

// Why in step[3.进行当前层的处理计算] we can return directly ?

// Because in Top Down DFS we traverse from root to leaf, if the tree

// current root is a BST, it naturally has more nodes than any BST

// start from current root's left or right, we can guarantee return

// directly still has the maximum number of nodes

if(isValidBST(root, Long.MIN\_VALUE, Long.MAX\_VALUE)) {

return countNodes(root);

}

// 递归成为更小的问题

int left = largestBSTSubtree(root.left);

int right = largestBSTSubtree(root.right);

return Math.max(left, right);

}

// Top Down DFS (遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题)

private boolean isValidBST(TreeNode root, long min, long max) {

// Base

if(root == null) {

return true;

}

// 进行当前层的处理计算

if(root.val <= min || root.val >= max) {

return false;

}

// 递归成为更小的问题

boolean left = isValidBST(root.left, min, root.val);

boolean right = isValidBST(root.right, root.val, max);

return left && right;

}

// Bottom Up DFS (分治法123: 1.base case -> 2.递归成为更小的问题 -> 3.进行当前层的处理计算)

private int countNodes(TreeNode root) {

// Base

if(root == null) {

return 0;

}

// Divide (递归成为更小的问题)

int left = countNodes(root.left);

int right = countNodes(root.right);

// Process & Conquer (进行当前层的处理计算)

return 1 + left + right;

}

}

Time Complexity: O(n^2)

Space Complexity: O(n^2)

**Refer to**

<https://www.cnblogs.com/grandyang/p/5188938.html>

这道题让我们求一棵二分树的最大二分搜索子树，所谓二分搜索树就是满足左<根<右的二分树，需要返回这个二分搜索子树的节点个数。题目中给的提示说可以用之前那道 [Validate Binary Search Tree](http://www.cnblogs.com/grandyang/p/4298435.html) 的方法来做，时间复杂度为 O(n2)，这种方法是把每个节点都当做根节点，来验证其是否是二叉搜索数，并记录节点的个数，若是二叉搜索树，就更新最终结果，对于每一个节点，都来验证其是否是 BST，如果是的话，就统计节点的个数即可，参见代码如下

class Solution {

public:

int largestBSTSubtree(TreeNode\* root) {

if (!root) return 0;

if (isValid(root, INT\_MIN, INT\_MAX)) return count(root);

return max(largestBSTSubtree(root->left), largestBSTSubtree(root->right));

}

bool isValid(TreeNode\* root, int mn, int mx) {

if (!root) return true;

if (root->val <= mn || root->val >= mx) return false;

return isValid(root->left, mn, root->val) && isValid(root->right, root->val, mx);

}

int count(TreeNode\* root) {

if (!root) return 0;

return count(root->left) + count(root->right) + 1;

}

};

**Style 2: Top Down DFS with classical global variable and void return (largestBSTSubtree + helper) + Top Down DFS (isValidBST) + Bottom Up DFS (countNodes)**

public class TreeSolution {

private class TreeNode {

public int val;

public TreeNode left, right;

public TreeNode(int val) {

this.val = val;

this.left = this.right = null;

}

}

public static void main(String[] args) {

TreeSolution s = new TreeSolution();

TreeNode one = s.new TreeNode(1);

TreeNode two = s.new TreeNode(2);

TreeNode three = s.new TreeNode(3);

TreeNode four = s.new TreeNode(4);

TreeNode five = s.new TreeNode(5);

TreeNode six = s.new TreeNode(6);

TreeNode seven = s.new TreeNode(7);

TreeNode eight = s.new TreeNode(8);

/\*\*

\* 2

\* / \

\* 1 3

\* / \

\* 4 5

\*/

two.left = one;

two.right = three;

three.left = four;

three.right = five;

int result = s.largestBSTSubtree(two);

System.out.println(result);

}

int maxNodes = 0;

public int largestBSTSubtree(TreeNode root) {

// Base

if(root == null) {

return 0;

}

helper(root);

return maxNodes;

}

// Top Down DFS (遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题)

private void helper(TreeNode root) {

// Base

if(root == null) {

return;

}

// 进行当前层的处理计算

// We can return directly to skip left and right branch redundant check

// because in Top Down DFS we traverse from root to leaf, if the tree

// current root is a BST, it naturally has more nodes than any BST

// start from current root's left or right, we can guarantee return

// directly still has the maximum number of nodes

if(isValidBST(root, Long.MIN\_VALUE, Long.MAX\_VALUE)) {

maxNodes = Math.max(maxNodes, countNodes(root));

// If not return directly, answer still right, but speed down

return;

}

// 递归成为更小的问题

helper(root.left);

helper(root.right);

}

// Top Down DFS (遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题)

private boolean isValidBST(TreeNode root, long min, long max) {

// Base

if(root == null) {

return true;

}

// 进行当前层的处理计算

if(root.val <= min || root.val >= max) {

return false;

}

// 递归成为更小的问题

boolean left = isValidBST(root.left, min, root.val);

boolean right = isValidBST(root.right, root.val, max);

return left && right;

}

// Bottom Up DFS (分治法123: 1.base case -> 2.递归成为更小的问题 -> 3.进行当前层的处理计算)

private int countNodes(TreeNode root) {

// Base

if(root == null) {

return 0;

}

// Divide (递归成为更小的问题)

int left = countNodes(root.left);

int right = countNodes(root.right);

// Process & Conquer (进行当前层的处理计算)

return 1 + left + right;

}

}

Time Complexity: O(n^2)

Space Complexity: O(n^2)

**Refer to**

<https://www.jianshu.com/p/fa7a1ce4e614>

**Top down approach 有两种，一种是Top down+ Top down, 另外一种是Top down + Bottom up，两种区别在于find\_tree utility function, 两种方案都是基于Top down的DFS是经典global variable + void return (Top Down DFS with classical global variable and void return).**

**Top down + Top down**

class Solution {

public:

// Top down helper

void FindTree\_util(TreeNode \*root, TreeNode \*large, TreeNode \*small, int &cur) {

if(!root) return;

if(large && large->val <= root->val) {

cur = -1;

return;

}

else if(small && small->val >= root->val) {

cur = -1;

return;

}

cur = cur + 1;

FindTree\_util(root->left, root, small, cur);

FindTree\_util(root->right, large, root, cur);

}

// Top down DFS with classical global variable &max\_ret and void return

void FindTree(TreeNode\* root, int &max\_ret) {

if(!root) return;

int cur = 0;

FindTree\_util(root, NULL, NULL, cur);

if(cur != -1){

max\_ret = max(max\_ret, cur);

}

FindTree(root->left, max\_ret);

FindTree(root->right, max\_ret);

}

int largestBSTSubtree(TreeNode\* root) {

if(!root) {

return 0;

}

int max\_ret = 0;

FindTree(root, max\_ret);

return max\_ret;

}

};

**Top down + Bottom up**

class Solution {

public:

// Bottom up helper

int FindTree\_util(TreeNode \*root, TreeNode \*large, TreeNode \*small) {

if(!root) return 0;

if(large && large->val <= root->val) {

return -1;

}

else if(small && small->val >= root->val) {

return -1;

}

int left\_value = FindTree\_util(root->left, root, small);

if(left\_value == -1) {

return -1;

}

int right\_value = FindTree\_util(root->right, large, root);

if(right\_value == -1) {

return -1;

}

return left\_value + right\_value + 1;

}

// Top down DFS with classical global variable &max\_ret and void return

void FindTree(TreeNode\* root, int &max\_ret) {

if(!root) return;

int ret = FindTree\_util(root, NULL, NULL);

if(ret > max\_ret) {

max\_ret = ret;

}

FindTree(root->left, max\_ret);

FindTree(root->right, max\_ret);

}

int largestBSTSubtree(TreeNode\* root) {

if(!root) return 0;

int max\_ret = 0;

FindTree(root, max\_ret);

return max\_ret;

}

};

**Solution 2: Divide and Conquer one pass DFS (60 min)**

public class TreeSolution {

private class TreeNode {

public int val;

public TreeNode left, right;

public TreeNode(int val) {

this.val = val;

this.left = this.right = null;

}

}

public static void main(String[] args) {

TreeSolution s = new TreeSolution();

TreeNode one = s.new TreeNode(1);

TreeNode two = s.new TreeNode(2);

TreeNode three = s.new TreeNode(3);

TreeNode four = s.new TreeNode(4);

TreeNode five = s.new TreeNode(5);

TreeNode six = s.new TreeNode(6);

TreeNode seven = s.new TreeNode(7);

TreeNode eight = s.new TreeNode(8);

/\*\*

\* 2

\* / \

\* 1 3

\* / \

\* 4 5

\*/

two.left = one;

two.right = three;

three.left = four;

three.right = five;

int result = s.largestBSTSubtree(two);

System.out.println(result);

}

// Since no need to return 'TreeNode', the helper class no need contains 'TreeNode'

class Node {

// Record maximum BST size till current node

int size;

boolean isBST;

// Record min value in the subtree till current node

int min;

// Record max value in the subtree till current node

int max;

public Node(int size, boolean isBST, int min, int max) {

this.size = size;

this.isBST = isBST;

this.min = min;

this.max = max;

}

}

public int largestBSTSubtree(TreeNode root) {

if(root == null) {

return 0;

}

Node result = helper(root);

return result.size;

}

// Bottom Up DFS (分治法123: 1.base case -> 2.递归成为更小的问题 -> 3.进行当前层的处理计算)

private Node helper(TreeNode root) {

// Base

if(root == null) {

return new Node(0, true, Integer.MAX\_VALUE, Integer.MIN\_VALUE);

}

// 递归成为更小的问题

Node left = helper(root.left);

Node right = helper(root.right);

// 进行当前层的处理计算

// Update min and max value for current node based on its left and right subtree records

int curMin = Math.min(left.min, root.val);

int curMax = Math.max(root.val, right.max);

// If both left and right subtree are BST and current node value in range

// (left.max, right.min), then subtree start from current node is a BST

if(left.isBST && right.isBST && root.val > left.max && root.val < right.min) {

return new Node(1 + left.size + right.size, true, curMin, curMax);

} else {

return new Node(Math.max(left.size, right.size), false, curMin, curMax);

}

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://tenderleo.gitbooks.io/leetcode-solutions-/content/GoogleMedium/333.html>

1. you need to track each subtree is BST or not.

2. you need to track the size of subtree if it is a BST.

3. thus global variable / TreeNode won't keep consistent info regarding 1&2.

4. you need a wrapper to hold such 2 information. along with the current range of subtree.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

/\*

1. you need to track each subtree is bst or not.

2. you need to track the size of subtree if it is a bst.

3. thus global variable/TreeNode won't keep consistent info

regarding 1&2.

4. you need a wrapper to hold such 2 information. along with the

current range of substree.

\*/

class Node{

int size;

int left,right;

boolean isBst;

Node(){

size = 0;

isBst = true;

left = Integer.MAX\_VALUE;

right = Integer.MIN\_VALUE;

}

}

public int largestBSTSubtree(TreeNode root) {

Node n = isBST(root);

return n.size;

}

Node isBST(TreeNode root){

Node node = new Node();

if(root == null){

return node;

}

Node l = isBST(root.left);

Node r = isBST(root.right);

node.left = Math.min(l.left, root.val);

node.right = Math.max(r.right, root.val);

if(l.isBst && r.isBst && l.right <= root.val && r.left >= root.val){

node.size = l.size + r.size +1;

node.isBst = true;

}else{

node.size = Math.max(l.size, r.size);

node.isBst = false;

}

return node;

}

}

**Refer to**

<https://www.cnblogs.com/grandyang/p/5188938.html>

题目中的 Follow up 让用 O(n) 的时间复杂度来解决问题，还是采用 DFS 的思想来解题，由于时间复杂度的限制，只允许遍历一次整个二叉树，由于满足题目要求的二叉搜索子树必定是有叶节点的，所以思路就是先递归到最左子节点，然后逐层往上递归，对于每一个节点，都记录当前最大的 BST 的节点数，当做为左子树的最大值，和做为右子树的最小值，当每次遇到左子节点不存在或者当前节点值大于左子树的最大值，且右子树不存在或者当前节点值小于右子树的最小数时，说明 BST 的节点数又增加了一个，更新结果及其参数，如果当前节点不是 BST 的节点，那么更新 BST 的节点数 res 为左右子节点的各自的 BST 的节点数的较大值，参见代码如下：

class Solution {

public:

int largestBSTSubtree(TreeNode\* root) {

int res = 0, mn = INT\_MIN, mx = INT\_MAX;

isValidBST(root, mn, mx, res);

return res;

}

void isValidBST(TreeNode\* root, int& mn, int& mx, int& res) {

if (!root) return;

int left\_cnt = 0, right\_cnt = 0, left\_mn = INT\_MIN;

int right\_mn = INT\_MIN, left\_mx = INT\_MAX, right\_mx = INT\_MAX;

isValidBST(root->left, left\_mn, left\_mx, left\_cnt);

isValidBST(root->right, right\_mn, right\_mx, right\_cnt);

if ((!root->left || root->val > left\_mx) && (!root->right || root->val < right\_mn)) {

res = left\_cnt + right\_cnt + 1;

mn = root->left ? left\_mn : root->val;

mx = root->right ? right\_mx : root->val;

} else {

res = max(left\_cnt, right\_cnt);

}

}

};