# 题目

给定一个二叉树，找到其中最大的二叉搜索树（BST）子树，并返回该子树的大小。其中，最大指的是子树节点数最多的。

二叉搜索树（BST）中的所有节点都具备以下属性：

左子树的值小于其父（根）节点的值。

右子树的值大于其父（根）节点的值。

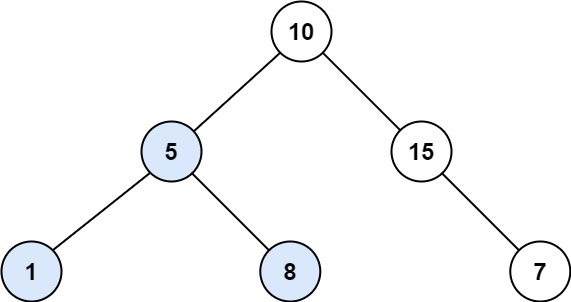
注意:

子树必须包含其所有后代。

进阶:

你能想出 O(n) 时间复杂度的解法吗？

示例 1：



输入：root = [10,5,15,1,8,null,7]

输出：3

解释：本例中最大的 BST 子树是高亮显示的子树。返回值是子树的大小，即 3 。

示例 2：

输入：root = [4,2,7,2,3,5,null,2,null,null,null,null,null,1]

输出：2

提示：

树上节点数目的范围是 [0, 104]

-104 <= Node.val <= 104

# 分析

预备知识：二叉搜索树（BST）

二叉搜索树，它或者是一棵空树，或者是具有下列性质的二叉树：若它的左子树不为空，则左子树上所有节点的值均小于它的根节点的值； 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；它的左右子树也为二叉搜索树。

## 方法一：枚举

直观的想法就是我枚举每一个节点，去检查这个节点为根的子树是不是一棵二叉搜索树，如果是的话就统计一下这个子树的节点数并更新答案，否则就继续递归到左右子树去找有没有符合条件的二叉搜索树。

那么先考虑如何判断一棵树是不是二叉搜索树，根据二叉搜索树的性质，我们可以从上往下递归判断。

设计一个函数valid(TreeNode∗ root,int l,int r) 表示考虑到当前节点，该节点为根的子树里面所有的值在(l,r) 范围（注意是开区间）内是不是一棵二叉搜索树， 如果 root->val不在(l,r) 的范围内说明不满足条件直接返回，如果在的话我们就要继续检查它的左右子树是否满足，如果都满足才说明这是一棵二叉搜索树。

那么根据二叉搜索树的性质我们分左右两个子树递归下去，即对于左子树，检查valid(root−>left,l,root−>val)，因为左子树里所有节点的值均小于它的根节点的值，所以我们要把范围从(l,r)改为(l,root−>val)，同理对于右子树我们递归检查valid(root−>right,root−>val,r)即可，最后函数的入口为valid(root,INT\_MIN,INT\_MAX)。

如果该节点为根的子树是一棵二叉搜索树，则我们再dfs一遍统计树的节点数即可，设计函数cnt(root)表示以root为根的子树节点个数，那么它为cnt(root−>left)+cnt(root−>right)+1，即左子树节点个数加右子树节点个数加上它自身，如此递归下去直到碰到空节点返回即可。

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool valid(TreeNode\* root,int l,int r){

if (root==NULL) return 1;

if (root->val<=l || root->val>=r) return 0;

return valid(root->left,l,root->val) && valid(root->right,root->val,r);

}

int cnt(TreeNode\* root){return root?cnt(root->left)+cnt(root->right)+1:0;}

int largestBSTSubtree(TreeNode\* root) {

if (root==NULL) return 0;

if (valid(root,INT\_MIN,INT\_MAX)) return cnt(root);

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

}

};

复杂度分析

时间复杂度：我们枚举每个节点需要O(n)的时间，而检查每个节点为根的子树是不是二叉搜索树需要O(n)的时间，所以一共需要 O(n^2)的时间复杂度。

空间复杂度：由于递归函数在递归过程中需要为每一层递归函数分配栈空间，所以这里需要额外的空间且该空间取决于递归的深度，即二叉树的高度。最坏情况下二叉树为一条链，树的高度为n，递归最深达到n层，故最坏情况下空间复杂度为O(n)，最好情况下二叉树是满二叉树，这种情况下树的高度最小，为 log2n ，故最好情况下空间复杂度为O(log2n) 。

## 方法二：复用子树信息

方法一只是简单的将一个问题割裂成了若干个判断子树是不是二叉搜索树的问题，并没有复用已有的信息，如果我们能够学会复用一些已有的信息，则就有可能优化现有算法的时间复杂度。

那么对于这题而言，我们从预备知识里知道，一棵树如果是二叉搜索树，那么它的左右子树也必然是二叉搜索树，则对于一个节点为根的子树，如果我们已经知道了左右子树是不是二叉搜索树，以及左右子树的值的范围[l,r] ，那么如果左右子树均为二叉搜索树，根据性质我们只要判断该节点的值是不是大于左子树的最大值和小于右子树的最小值即能推断出该节点为根的子树是不是二叉搜索树，而又因为我们已经拿到了左右子树的信息，所以这个推断只需要O(1) 的时间复杂度，而方法一不复用信息的话判断一棵子树是不是二叉搜索树则需要 O(n) 的时间复杂度。

根据上面说的我们设计一个递归函数dfs(TreeNode∗ root) ，表示考虑当前节点是不是一棵二叉搜索树，先递归左右子树，拿到左右子树是不是二叉搜索树，以及相应值的范围，再根据上文说的更新当前的节点的信息即可。由于递归函数需要返回三个信息，则我们建一个结构体存储信息：

//Node:l,r表示当前节点为根的二叉搜索树里的值的范围[l,r]，sz为这棵树的节点数，如果不是BST,则sz=-1，还未递归前l=r=root->val

struct Node{

int l,r,sz;

};

如果当前节点是二叉搜索树的话，则需要更新当前值的范围，这个直接拿左右子树各自的[l,r] 去更新当前子树的[l,r]即可，如果不是把sz改为−1表示这个节点为根的子树不是二叉搜索树，然后再递归回上一层更新其父亲节点的信息，可以看出这是一个从下往上更新信息的过程。

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

//Node:l,r表示当前节点为根的二叉搜索树里的值的范围[l,r]，sz为这棵树的节点数，如果不是BST,则sz=-1

struct Node{

int l,r,sz;

};

class Solution {

int ans=0;

public:

Node dfs(TreeNode\* root){

if (root->left==NULL && root->right==NULL){

ans=max(ans,1);

return (Node){root->val,root->val,1};

}

int sz=1;

bool valid=1;

int l=root->val,r=root->val;// 起初的范围是[root->val,root->val]，再根据左右子树更新范围

if (root->left!=NULL){

Node L=dfs(root->left);

if (L.sz!=-1 && root->val>L.r){

sz+=L.sz;

l=L.l;

}

else valid=0;

}

if (root->right!=NULL){

Node R=dfs(root->right);

if (R.sz!=-1 && root->val<R.l){

sz+=R.sz;

r=R.r;

}

else valid=0;

}

if (valid){

ans=max(ans,sz);

return (Node){l,r,sz};

}

// 不是BST，sz设为-1标记不是BST，l,r多少都可以

return (Node){-1,-1,-1};

}

int largestBSTSubtree(TreeNode\* root) {

if (root==NULL) return 0;

dfs(root);

return ans;

}

};

复杂度分析

时间复杂度：O(n) ，即遍历一棵树的时间复杂度。

空间复杂度：由于递归函数在递归过程中需要为每一层递归函数分配栈空间，所以这里需要额外的空间且该空间取决于递归的深度，即二叉树的高度。最坏情况下二叉树为一条链，树的高度为n，递归最深达到n层，故最坏情况下空间复杂度为O(n) ，最好情况下二叉树是满二叉树，这种情况下树的高度最小，为log2n，故最好情况下空间复杂度为O(log2n)。