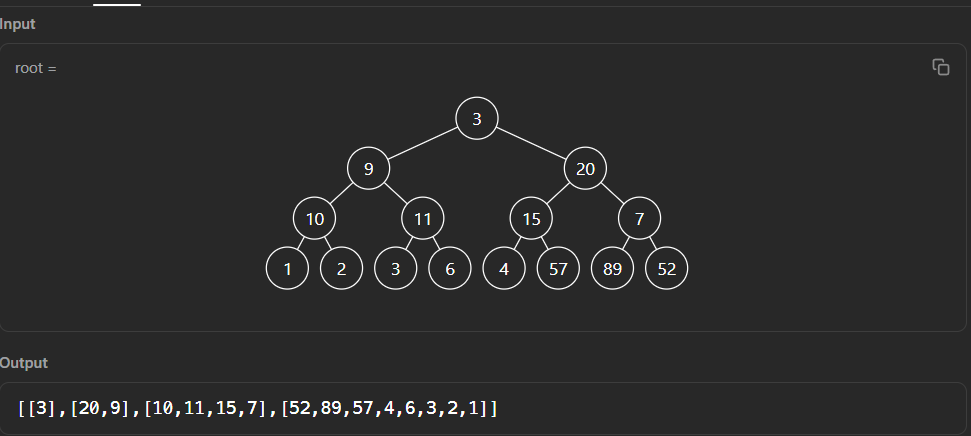
Tree Medium Questions

# **103. Binary Tree Zigzag Level Order Traversal**



Approach

* Level order traversal with an additional variable
* In each level need to make that variable opposite (if tree then false, else true)
* From level order traversal, for the 2nd level we will get [9, 20] for the 2nd level, and if the value of that flag variable is true then will rev the array means [20, 9].

class Solution {

public:

vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

vector<vector<int>> ans;

if(!root) return ans;

queue<TreeNode \*> q;

q.push(root);

bool needToRev = false;

while(!q.empty()){

vector<int> temp;

int s = q.size();

while(s--) {

TreeNode \*node = q.front();

q.pop();

if(node->left) q.push(node->left);

if(node->right) q.push(node->right);

temp.push\_back(node->val);

}

if(needToRev) reverse(temp.begin(), temp.end());

needToRev = !needToRev;

ans.push\_back(temp);

}

return ans;

}

};

# **Deepest Leaves Sum**

**Using Queue Approach:**

1. Store level wise sum and return last level sum.

class Solution {

public:

int deepestLeavesSum(TreeNode\* root) {

if(!root) return false;

queue<TreeNode \*> q;

q.push(root);

int lastLevelSum = root->val;

while(!q.empty()) {

int s = q.size();

int temp = 0;

while(s--) {

if(q.front()->left) {

q.push(q.front()->left);

temp += q.front()->left->val;

}

if(q.front()->right) {

q.push(q.front()->right);

temp += q.front()->right->val;

}

q.pop();

}

if(q.empty()) break;

lastLevelSum = temp;

}

return lastLevelSum;

}

};

**Recursion Solution**

class Solution {

public:

pair<int,int> ans = {0, INT\_MIN};

bool isleaf(TreeNode\* root) {

return !root->left && !root->right;

}

int sum = 0;

void traverse(TreeNode\* node, int level) {

if(!node) return;

if(isleaf(node)) {

if(ans.first < level) {

ans.first = level;

ans.second = node->val;

}

else if(ans.first == level) ans.second += node->val;

return;

}

traverse(node->left, level + 1);

traverse(node->right, level + 1);

}

int deepestLeavesSum(TreeNode\* root) {

traverse(root, 1);

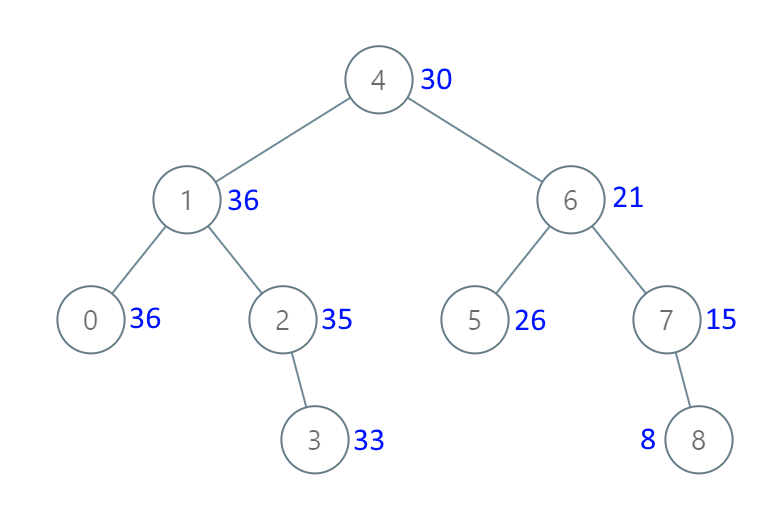
root->left = root->right = NULL;

return ans.second;

}

};

# Binary Search Tree to Greater Sum Tree



class Solution {

int recFunc(TreeNode \*root, int val) {

if(!root) return val;

int rv = recFunc(root->right, val);

root->val += rv;

return recFunc(root->left, root->val);

}

public:

TreeNode\* bstToGst(TreeNode\* root) {

recFunc(root, 0);

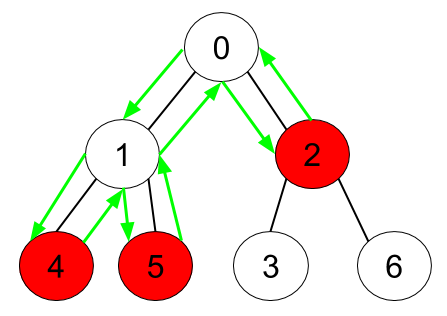
return root;

}

};

# **Minimum Time to Collect All Apples in a Tree**

* Given an undirected tree consisting of n vertices numbered from 0 to n-1, which has some apples in their vertices. You spend 1 second to walk over one edge of the tree. Return the minimum time in seconds you have to spend to collect all apples in the tree, starting at vertex 0 and coming back to this vertex.
* The edges of the undirected tree are given in the array edges, where edges[i] = [ai, bi] means that exists an edge connecting the vertices ai and bi. Additionally, there is a boolean array hasApple, where hasApple[i] = true means that vertex i has an apple; otherwise, it does not have any apple.



class Solution {

int recFunc(int node, vector<vector<int>> &adj, vector<bool>& hasApple,

vector<int> &visited){

int time = 0;

visited[node] = 1;

for(auto i : adj[node]){

if(visited[i] == 0) {

time += recFunc(i, adj, hasApple, visited);

}

}

if(hasApple[node] || time > 0) return time + 2;

return 0;

}

public:

int minTime(int n, vector<vector<int>>& edges, vector<bool>& hasApple) {

vector<vector<int>> adj (n);

vector<int> visited (n,0);

for(auto it : edges) {

adj[it[0]].push\_back(it[1]);

adj[it[1]].push\_back(it[0]);

}

int time = recFunc(0, adj, hasApple, visited);

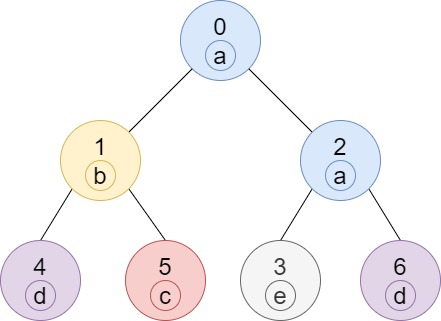
return time > 0 ? time - 2 : 0;

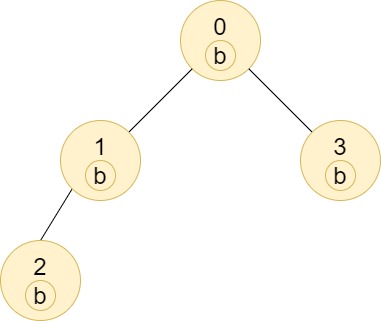
}

};

# Number of Nodes in the Sub-Tree With the Same Label

* You are given a tree (i.e. a connected, undirected graph that has no cycles) consisting of n nodes numbered from 0 to n - 1 and exactly n - 1 edges. The root of the tree is the node 0, and each node of the tree has a label which is a lower-case character given in the string labels (i.e. The node with the number i has the label labels[i]).
* The edges array is given on the form edges[i] = [ai, bi], which means there is an edge between nodes ai and bi in the tree.
* Return an array of size n where ans[i] is the number of nodes in the subtree of the ith node which have the same label as node i.
* A subtree of a tree T is the tree consisting of a node in T and all of its descendant nodes.

 **Output:** [2,1,1,1,1,1,1]

 **Output:** [4,2,1,1]

class Solution {

vector<int> recFunc(vector<vector<int>> &adj, vector<int> &ans, int node, string &labels){

vector<int> val(26, 0);

ans[node] = 1;

for(auto nextNode : adj[node]){

if(ans[nextNode] == 0){

vector<int> c = recFunc(adj, ans, nextNode, labels);

for(int i = 0; i < 26; i++) val[i] += c[i];

}

}

val[labels[node] - 'a'] += 1;

ans[node] = val[labels[node] - 'a'];

return val;

}

public:

vector<int> countSubTrees(int n, vector<vector<int>>& edges, string labels) {

vector<int> ans(n, 0), visited(n, 0);

vector<vector<int>> adj(n);

for(auto it : edges) {

adj[it[0]].push\_back(it[1]);

adj[it[1]].push\_back(it[0]);

}

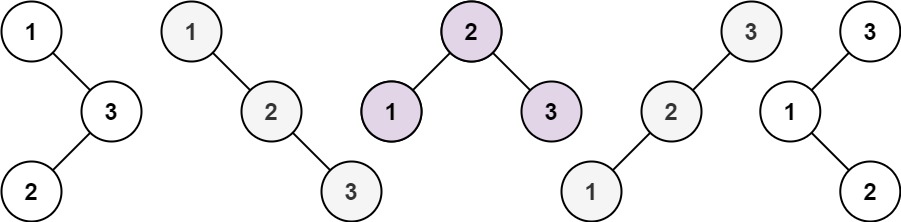
recFunc(adj, ans, 0, labels);

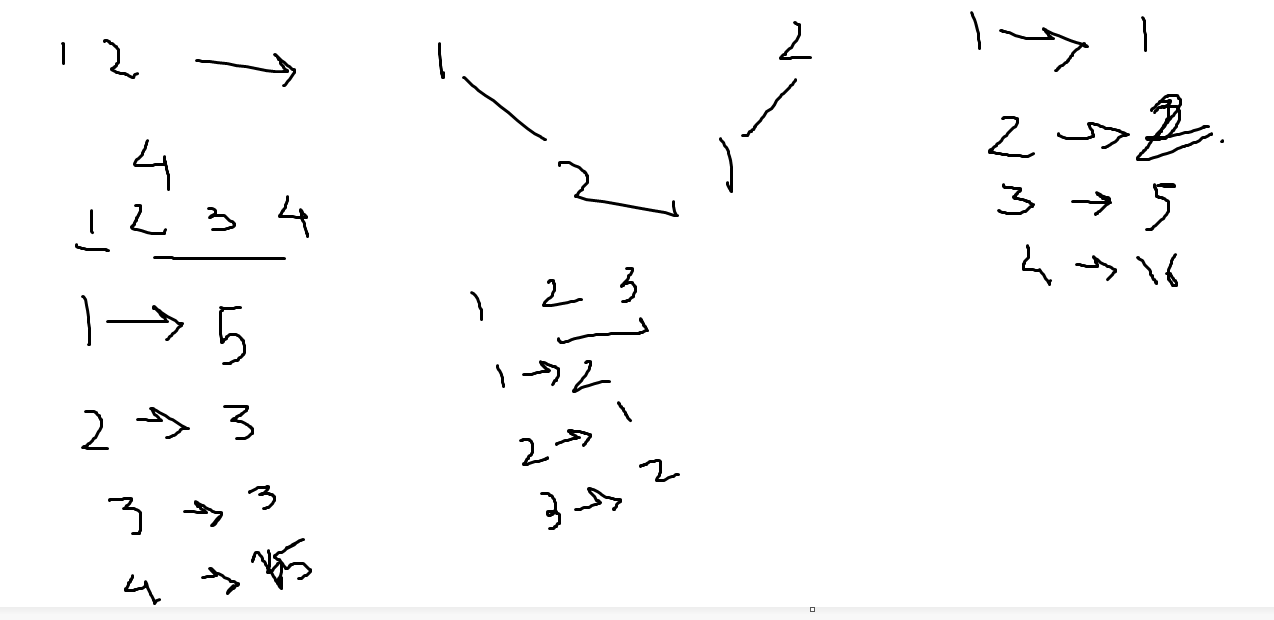
return ans;

}

};

# **Unique Binary Search Trees**





class Solution {

public:

int recFunc(int n, vector<int> &dp) {

cout << n << endl;

if(dp[n] != -1) return dp[n];

int ans = 0;

for(int i = 1; i <= n; i++) {

int n\_left\_node = i - 1;

int n\_right\_node = n - i;

ans += (recFunc(n\_left\_node, dp) \* recFunc(n\_right\_node, dp));

}

return dp[n] = ans;

}

int numTrees(int n){

if(n <= 1) return 1;

vector<int> dp(n+1, -1);

dp[0] = 1; dp[1] = 1; dp[2] = 2;

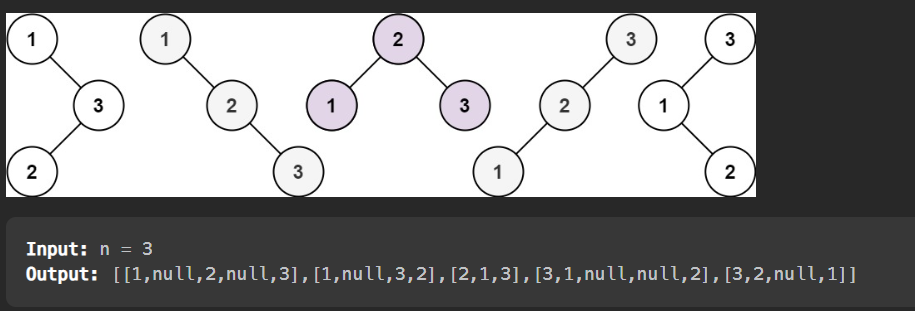
recFunc(n, dp);

return dp[n];

}

};

# **Unique Binary Search Trees II**



class Solution {

vector<TreeNode \*> recFunc(int s, int e) {

if(s > e) return {NULL};

if(s == e) return {new TreeNode(s)};

vector<TreeNode \*> ans;

for(int i = s; i <= e; i++) {

auto left\_trees = recFunc(s,i-1);

auto right\_trees = recFunc(i+1, e);

for(auto lt : left\_trees) {

for(auto rt : right\_trees) {

TreeNode \*node = new TreeNode(i);

node->left = lt;

node->right = rt;

ans.push\_back(node);

}

}

}

return ans;

}

public:

vector<TreeNode\*> generateTrees(int n) {

return recFunc(1, n);

}

};

# Validate Binary Search Tree

class Solution {

bool recFunc(TreeNode \*node, long long int left\_bound, long long int right\_bound){

if(!node) return true;

if(!(node->val > left\_bound && node->val < right\_bound))

return false;

return recFunc(node->left, left\_bound, node->val)

&& recFunc(node->right, node->val, right\_bound);

}

public:

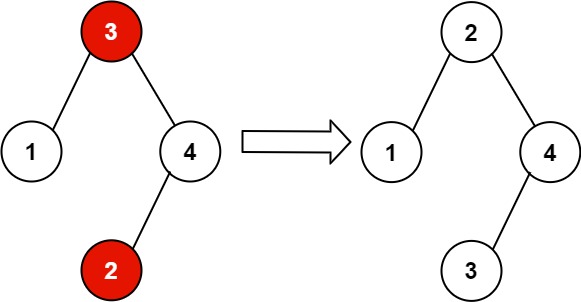
bool isValidBST(TreeNode\* root) {

return recFunc(root, -1 \* 1e10, 1e10);

}

};

# **Recover Binary Search Tree**



**Approach:**

* Do InOrder Traversal to get the tree in sorted order.
* Store Previous node value in any temp value.

class Solution {

TreeNode \*t1 = NULL, \*t2 = NULL, \*prev\_node = NULL;

void recFunc(TreeNode \*node) {

if(!node) return ;

recFunc(node->left);

if(!prev\_node) prev\_node = node;

else{

if(node->val < prev\_node->val) {

if(t2) t2 = node;

else {

t1 = prev\_node;

t2 = node;

}

}

prev\_node = node;

}

recFunc(node->right);

}

public:

void recoverTree(TreeNode\* root) {

recFunc(root);

int v = t1->val;

t1->val = t2->val;

t2->val = v;

}

};

# Binary Tree Level Order Traversal



class Solution {

public:

vector<vector<int>> levelOrder(TreeNode\* root) {

vector<vector<int>> ans;

queue<TreeNode \*> q;

if(!root) return ans;

q.push(root);

while(!q.empty()) {

int s = q.size();

vector<int> temp;

while(s--) {

TreeNode \*node = q.front();

q.pop();

temp.push\_back(node->val);

if(node->left) q.push(node->left);

if(node->right) q.push(node->right);

}

ans.push\_back(temp);

}

return ans;

}

};

# **230. Kth Smallest Element in a BST**

**InOrder Traversal always return an sorted array.**

class Solution {

int val = INT\_MIN;

void recFunc(TreeNode \*node, int &k) {

if(!node || k <= 0) return;

recFunc(node->left, k);

k--;

if(k == 0) val = node->val;

recFunc(node->right, k);

}

public:

int kthSmallest(TreeNode\* node, int k) {

recFunc(node, k);

return val;

}

};

# **Count Complete Tree Nodes**

class Solution {

int recFunc1(TreeNode \*node) {

if(!node) return 0;

return 1 + recFunc1(node->left);

}

int recFunc2(TreeNode \*node) {

if(!node) return 0;

return 1 + recFunc2(node->right);

}

public:

int countNodes(TreeNode\* root) {

if(!root) return 0;

int lh = recFunc1(root);

int rh = recFunc2(root);

if(lh == rh) return (1 << lh) - 1;

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

}

};

# **Sum Root to Leaf Numbers**

class Solution {

int ans = 0;

void recFunc(TreeNode \*root, int val) {

if(!root) return;

val = (val \* 10) + root->val;

if(!root->left && !root->right) {

ans += val;

return;

}

recFunc(root->left, val);

recFunc(root->right, val);

}

public:

int sumNumbers(TreeNode\* root) {

recFunc(root, 0);

return ans;

}

};

# **Check Completeness of a Binary Tree**

class Solution {

public:

bool isCompleteTree(TreeNode\* root) {

queue<TreeNode \*> q;

if(!root) return false;

q.push(root);

bool isLastNode = false;

while(!q.empty()) {

TreeNode \*t = q.front();

q.pop();

if(isLastNode && t->left) return false;

if(t->left) q.push(t->left);

else isLastNode = true;

if(isLastNode && t->right) return false;

if(t->right) q.push(t->right);

else isLastNode = true;

}

return true;

}

};