<https://leetcode.ca/all/298.html>

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

Example 1:

Input:

1

\

3

/ \

2 4

\

5

Output: 3

Explanation: Longest consecutive sequence path is 3-4-5, so return 3.

Example 2:

Input:

2

\

3

/

2

/

1

Output: 2

Explanation: Longest consecutive sequence path is 2-3, not 3-2-1, so return 2.

**Attempt 1: 2022-12-28**

**Solution 1: Recursive traversal with global variable and update on the fly during each recursion (30min)**

**Style 1: Intuitive way**

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) {

/\*\*

\* 1

\* / \

\* 2 5

\* / \ \

\* 3 4 6

\* /

\* 7

\* \

\* 8

\*/

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);

one.left = two;

one.right = five;

two.left = three;

two.right = four;

five.right = six;

six.left = seven;

seven.right = eight;

int result = s.longestConsecutive(one);

System.out.println(result);

}

int count = 0;

public int longestConsecutive(TreeNode root) {

helper(root, 1);

return count;

}

private void helper(TreeNode root, int curPathCount) {

if(root == null) {

return;

}

// Update global variable on the fly in each recursion

count = Math.max(count, curPathCount);

if(root.left != null) {

// If match the consecutive sequence condition, continue on this path

if(root.left.val == root.val + 1) {

helper(root.left, curPathCount + 1);

// If not match the consecutive sequence condition, restart a new path from current node

} else {

helper(root.left, 1);

}

}

if(root.right != null) {

// If match the consecutive sequence condition, continue on this path

if(root.right.val == root.val + 1) {

helper(root.right, curPathCount + 1);

// If not match the consecutive sequence condition, restart a new path from current node

} else {

helper(root.right, 1);

}

}

}

}

Complexity Analysis

Time Complexity: O(N). Where N is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.

Space Complexity: O(N). This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed binary tree could be N.

**Refer to**

<https://www.cnblogs.com/grandyang/p/5252599.html>

下面这种写法是利用分治法的思想，对左右子节点分别处理，如果左子节点存在且节点值比其父节点值大1，则递归调用函数，如果节点值不是刚好大1，则递归调用重置了长度的函数，对于右子节点的处理情况和左子节点相同，参见代码如下：

class Solution {

public:

int longestConsecutive(TreeNode\* root) {

if (!root) return 0;

int res = 0;

dfs(root, 1, res);

return res;

}

void dfs(TreeNode \*root, int len, int &res) {

res = max(res, len);

if (root->left) {

if (root->left->val == root->val + 1) dfs(root->left, len + 1, res);

else dfs(root->left, 1, res);

}

if (root->right) {

if (root->right->val == root->val + 1) dfs(root->right, len + 1, res);

else dfs(root->right, 1, res);

}

}

};

**Style 2: Strick follow the Recursive traversal style (so called 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)**

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) {

/\*\*

\* 1

\* / \

\* 2 5

\* / \ \

\* 3 4 6

\* /

\* 7

\* \

\* 8

\*/

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);

one.left = two;

one.right = five;

two.left = three;

two.right = four;

five.right = six;

six.left = seven;

seven.right = eight;

int result = s.longestConsecutive(one);

System.out.println(result);

}

int count = 0;

public int longestConsecutive(TreeNode root) {

// Trick point: 2nd parameter represent parent node's value for

// each level recursion comparison, initialize as 'root.val'

// If not set up 2nd parameter, it is hard to build base condition

helper(root, root.val, 1);

return count;

}

private void helper(TreeNode root, int parentVal, int curPathCount) {

if(root == null) {

return;

}

// Base condition: build based on 2nd parameter 'parentVal'

// If match the consecutive sequence condition, continue on this path

if(root.val == parentVal + 1) {

curPathCount++;

// If not match the consecutive sequence condition, restart a new path from current node

} else {

curPathCount = 1;

}

count = Math.max(count, curPathCount);

helper(root.left, root.val, curPathCount);

helper(root.right, root.val, curPathCount);

}

}

Complexity Analysis

Time Complexity: O(N). Where N is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.

Space Complexity: O(N). This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed binary tree could be N.

**Refer to**

<https://www.cnblogs.com/grandyang/p/5252599.html>

这道题让我们求二叉树的最长连续序列，关于二叉树的题基本都需要遍历树，而递归遍历写起来特别简单，下面这种解法是用到了递归版的先序遍历，对于每个遍历到的节点，看节点值是否比参数值(父节点值)大1，如果是则长度加1，否则长度重置为1，然后更新结果 res，再递归调用左右子节点即可，参见代码如下：

class Solution {

public:

int longestConsecutive(TreeNode\* root) {

if (!root) return 0;

int res = 0;

dfs(root, root->val, 0, res);

return res;

}

void dfs(TreeNode \*root, int v, int out, int &res) {

if (!root) return;

if (root->val == v + 1) ++out;

else out = 1;

res = max(res, out);

dfs(root->left, root->val, out, res);

dfs(root->right, root->val, out, res);

}

};

**Solution 2: Divide and Conquer with global variable (30min)**

**The style is exactly same as L124/P9.7.Binary Tree Maximum Path Sum**

**Style  1: Count current root node (+ 1) later in conquer step, strictly follow the Divide and Conquer style (so called 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)**

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) {

/\*\*

\* 1

\* / \

\* 2 5

\* / \ \

\* 3 4 6

\* /

\* 7

\* \

\* 8

\*/

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);

one.left = two;

one.right = five;

two.left = three;

two.right = four;

five.right = six;

six.left = seven;

seven.right = eight;

int result = s.longestConsecutive(one);

System.out.println(result);

}

int count = 0;

public int longestConsecutive(TreeNode root) {

helper(root);

return count;

}

private int helper(TreeNode root) {

if(root == null) {

return 0;

}

// Divide

int leftCount = helper(root.left);

int rightCount = helper(root.right);

// Process (node under processing is on bottom level resepect

// to current recursion, that's why so called Bottom Up)

if(root.left != null && root.left.val != root.val + 1) {

leftCount = 0;

}

if(root.right != null && root.right.val != root.val + 1) {

rightCount = 0;

}

// Conquer (plus current root node in conquer step)

count = Math.max(count, Math.max(leftCount, rightCount) + 1);

return Math.max(leftCount, rightCount) + 1;

}

}

Complexity Analysis

Time Complexity: O(N). Where N is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.

Space Complexity: O(N). This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed binary tree could be N.

**Refer to**

<https://eugenejw.github.io/2017/08/leetcode-298>

public class Solution {

int ret = 0;

public int longestConsecutive(TreeNode root) {

dfs(root);

return ret;

}

private int dfs(TreeNode root) {

if (root == null) return 0;

int leftSum = dfs(root.left);

int rightSum = dfs(root.right);

if (root.left != null && root.left.val != root.val+1) {

leftSum = 0;

}

if (root.right != null && root.right.val != root.val+1) {

rightSum = 0;

}

ret = Math.max(ret, Math.max(leftSum, rightSum)+1);

return Math.max(leftSum, rightSum)+1;

}

}

**Style  2: Count current root node (+ 1) first in divide step, strictly follow the Divide and Conquer style (so called 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)**

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) {

/\*\*

\* 1

\* / \

\* 2 5

\* / \ \

\* 3 4 6

\* /

\* 7

\* \

\* 8

\*/

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);

one.left = two;

one.right = five;

two.left = three;

two.right = four;

five.right = six;

six.left = seven;

seven.right = eight;

int result = s.longestConsecutive(one);

System.out.println(result);

}

int count = 0;

public int longestConsecutive(TreeNode root) {

helper(root);

return count;

}

private int helper(TreeNode root) {

if(root == null) {

return 0;

}

// Divide (plus current root node in divide step)

int leftCount = helper(root.left) + 1;

int rightCount = helper(root.right) + 1;

// Process (node under processing is on bottom level resepect

// to current recursion, that's why so called Bottom Up)

if(root.left != null && root.left.val != root.val + 1) {

leftCount = 1;

}

if(root.right != null && root.right.val != root.val + 1) {

rightCount = 1;

}

// Conquer

count = Math.max(count, Math.max(leftCount, rightCount));

return Math.max(leftCount, rightCount);

}

}

Complexity Analysis

Time Complexity: O(N). Where N is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.

Space Complexity: O(N). This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed binary tree could be N.

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/graph-search/298.binary-tree-longest-consecutive-sequence>

class Solution {

private int maxLength = 0;

public int longestConsecutive(TreeNode root) {

dfs(root);

return maxLength;

}

private int dfs(TreeNode p) {

if (p == null) return 0;

int L = dfs(p.left) + 1;

int R = dfs(p.right) + 1;

if (p.left != null && p.val + 1 != p.left.val) {

L = 1;

}

if (p.right != null && p.val + 1 != p.right.val) {

R = 1;

}

int length = Math.max(L, R);

maxLength = Math.max(maxLength, length);

return length;

}

}