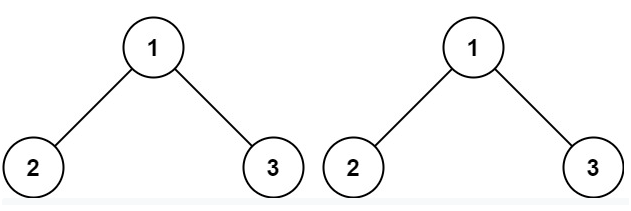
<https://leetcode.com/problems/same-tree/>

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

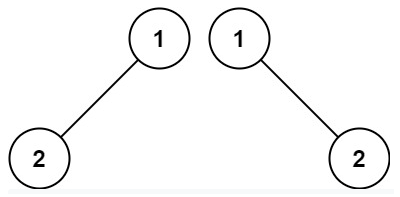
**Example 1:**



Input: p = [1,2,3], q = [1,2,3]

Output: true

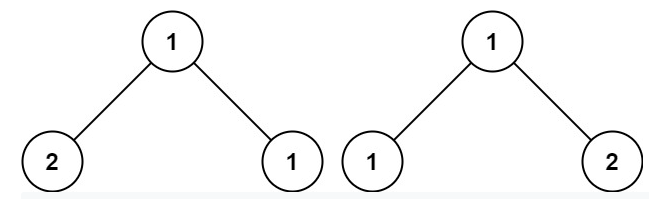
**Example 2:**



Input: p = [1,2], q = [1,null,2]

Output: false

**Example 3:**



Input: p = [1,2,1], q = [1,1,2]

Output: false

**Constraints:**

* The number of nodes in both trees is in the range [0, 100].
* -104 <= Node.val <= 104

**Attempt 1: 2022-11-08**

**Solution 1:  Divide and Conquer (10min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

// Base case

// Equal nullity denotes that this branch is the same (local equality)

// This is a base case, but also handles being given two empty trees

if(p == null && q == null) {

return true;

}

// Unequal nullity denotes that the trees aren't the same

// Note that we've already ruled out equal nullity above

if(p == null || q == null) {

return false;

}

// Both nodes have values; descend iff those values are equal

if(p.val != q.val) {

return false;

}

// Divide

boolean left = isSameTree(p.left, q.left);

boolean right = isSameTree(p.right, q.right);

// Conquer

return left && right;

}

}

Time Complexity: O(n), where n is number of nodes in the Binary Tree

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/same-tree/solution/>

**Approach 1: Recursion**

**Intuition**

The simplest strategy here is to use recursion. Check if p and q nodes are not None, and their values are equal. If all checks are OK, do the same for the child nodes recursively.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

// p and q are both null

if (p == null && q == null) return true;

// one of p and q is null

if (q == null || p == null) return false;

if (p.val != q.val) return false;

return isSameTree(p.right, q.right) &&

isSameTree(p.left, q.left);

}

}

**Complexity Analysis**

* Time complexity : O(N), where N is a number of nodes in the tree, since one visits each node exactly once.
* Space complexity : O(N)in the worst case of completely unbalanced tree, to keep a recursion stack.

**Solution 2: Iterative preorder traversal with stack (10min)**

**Note:  Iterative postorder traversal with stack will be similar, the only difference is switch the order between right and left subtree check in while loop, since we don't need list output like result.add(0, node.val) for postorder traversal, so the switch on order is the only change**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

if(p == null && q == null) {

return true;

}

if(p == null || q == null) {

return false;

}

Stack<TreeNode> sp = new Stack<TreeNode>();

sp.push(p);

Stack<TreeNode> sq = new Stack<TreeNode>();

sq.push(q);

while(!sp.isEmpty() && !sq.isEmpty()) {

TreeNode np = sp.pop();

TreeNode nq = sq.pop();

if(np.val != nq.val) {

return false;

}

if(np.right != null && nq.right != null) {

sp.push(np.right);

sq.push(nq.right);

} else if(np.right != null || nq.right != null) {

return false;

}

if(np.left != null && nq.left != null) {

sp.push(np.left);

sq.push(nq.left);

} else if(np.left != null || nq.left != null) {

return false;

}

}

return true;

}

}

Time Complexity: O(n), where n is number of nodes in the Binary Tree

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/same-tree/solution/>

**Approach 2: Iteration**

**Intuition**

Start from the root and then at each iteration pop the current node out of the deque. Then do the same checks as in the approach 1 :

* p and p are not None,
* p.val is equal to q.val,

and if checks are OK, push the child nodes.

**Implementation**

class Solution {

public boolean check(TreeNode p, TreeNode q) {

// p and q are null

if (p == null && q == null) return true;

// one of p and q is null

if (q == null || p == null) return false;

if (p.val != q.val) return false;

return true;

}

public boolean isSameTree(TreeNode p, TreeNode q) {

if (p == null && q == null) return true;

if (!check(p, q)) return false;

// init deques

ArrayDeque<TreeNode> deqP = new ArrayDeque<TreeNode>();

ArrayDeque<TreeNode> deqQ = new ArrayDeque<TreeNode>();

deqP.addLast(p);

deqQ.addLast(q);

while (!deqP.isEmpty()) {

p = deqP.removeFirst();

q = deqQ.removeFirst();

if (!check(p, q)) return false;

if (p != null) {

// in Java nulls are not allowed in Deque

if (!check(p.left, q.left)) return false;

if (p.left != null) {

deqP.addLast(p.left);

deqQ.addLast(q.left);

}

if (!check(p.right, q.right)) return false;

if (p.right != null) {

deqP.addLast(p.right);

deqQ.addLast(q.right);

}

}

}

return true;

}

}

**Complexity Analysis**

* Time complexity : O(N)since each node is visited exactly once.
* Space complexity : O(N)in the worst case, where the tree is a perfect fully balanced binary tree, since BFS will have to store at least an entire level of the tree in the queue, and the last level has O(N)nodes.

**Solution 3: Iterative inorder traversal with stack (10min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

if(p == null && q == null) {

return true;

}

if(p == null || q == null) {

return false;

}

Stack<TreeNode> sp = new Stack<TreeNode>();

//sp.push(p);

Stack<TreeNode> sq = new Stack<TreeNode>();

//sq.push(q);

while((p != null || !sp.isEmpty()) && (q != null || !sq.isEmpty())) {

while(p != null && q != null) {

sp.push(p);

p = p.left;

sq.push(q);

q = q.left;

}

// If while loop break out because of either p != null and q == null

// or p == null and q != null, then mismatch happen, we should return

// false before pop out previous stored nodes from stack

// Note: if p == null && q == null, we cannot judge easily by direct

// return true or false, because that only means two tree traversal

// sync up at same position with no more left leaves

if((p != null && q == null) || (p == null && q != null)) {

return false;

}

p = sp.pop();

q = sq.pop();

if(p.val != q.val) {

return false;

}

p = p.right;

q = q.right;

// Also need to check if p or q only one existing here, test out

// by input: [1] and [1,null,2]

if((p != null && q == null) || (p == null && q != null)) {

return false;

}

}

return true;

}

}

Time Complexity: O(n), where n is number of nodes in the Binary Tree

Space Complexity: O(n)