100. Same Tree

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

1. **Listen**

**Problem Statement:**

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

**Input:**

**roots** of **two binary trees** ***p*** and ***q***

**Goal:**

write a function to check if two binary trees are equal

same structure

same node values

**Return:**

true if the two binary trees are equal

false otherwise

1. **Examples**

Example 1:

A picture containing text, clipart, clock

Description automatically generated

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

**Output:** true

Example 2:

A picture containing text, clipart

Description automatically generated

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

**Output:** false

**Constraints:**

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

**Test Cases:**

* Trees are not equal
* Trees are equal
* Null tree(s)

1. **Brute Force**

**Approach 1: DFS Preorder Recursion**

**Intuition**

The simplest strategy here is to recursively traverse the tree with a preorder traversal.

Check if p and q nodes are not null.

We check nodes at the same point in each tree for equality (in terms of value).

Check that the left and right subtrees also are equal (recursively).

**Implementation**

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**

***N*** is a number of nodes in the p tree

***M*** is a number of nodes in the q tree

**Time Complexity**: ---------------------------------------------------------------------------------------------

**Worst Case**: Fully Skewed Tree OR Perfectly Balanced Tree – O(N)

We will visit each node exactly once to check if every node is equal.

It does not matter if the tree is skewed or balanced, it will have the same runtime.

**Worst Case Technicality:** The minimum height between the two trees

Technically, if the trees are not equal, then the time complexity would be

O(Math.min(N, M))

**Space Complexity**: --------------------------------------------------------------------------------------------

**Worst Case**: Fully Skewed Tree – O(N)

We will add up to N frames onto the stack once we reach a base case.

**Average Case**: Perfectly Balanced Tree – O(logN)

There will be up to logN frames on the stack at any time.

Once we reach a base case, we will only be H (height of tree) nodes deep. A frame will be popped of the stack and another will be added, therefore maintaining the space complexity.

1. **Optimize**

**Approach 2: DFS Preorder Iteration**

**Intuition**

We can do a basic preorder iterative traversal, and act like we are doing it on one tree. The only difference is

1. the equality checks
2. we perform double the number of stack operations

Start from both roots and then at each iteration pop the current nodes out from the stack.

Then do the same checks from Approach 1:

p and q are not null,

p is null or q is not OR q is not null and q is

p.val is not equal to q.val,

and if checks pass, push the child nodes.

**Implementation**

private boolean isSameTreeHelper2(TreeNode p, TreeNode q)

{

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

nodes.push(p);

nodes.push(q);

while(!nodes.isEmpty())

{

TreeNode node1 = nodes.pop();

TreeNode node2 = nodes.pop();

if(node1 == null && node2 == null) continue;

if(node1 == null || node2 == null) return false;

if(node1.val != node2.val) return false;

nodes.push(node1.right);

nodes.push(node2.right);

nodes.push(node1.left);

nodes.push(node2.left);

}

return true;

}

**Complexity Analysis**

***N*** is a number of nodes in the p tree

***M*** is a number of nodes in the q tree

**Time Complexity**: ---------------------------------------------------------------------------------------------

**Worst Case**: Fully Skewed Tree OR Perfectly Balanced Tree – O(N)

We will visit each node exactly once to check if every node is equal.

It does not matter if the tree is skewed or balanced, it will have the same runtime.

**Worst Case Technicality:** The minimum height between the two trees

Technically, if the trees are not equal, then the time complexity would be

O(Math.min(N, M))

**Space Complexity**: --------------------------------------------------------------------------------------------

**Worst Case**: Fully Skewed Tree – O(N) = O(H)

We will add up to N null nodes (from the right child) onto the stack.

**Average Case**: Perfectly Balanced Tree – O(logN) = O(H)

There will be up to logN frames on the stack at any time.

Once we reach a base case, we will only be H (height of tree) nodes deep.

Approach 3: Iterative BFS Traversal

**Complexity**

***N*** is a number of nodes in the p tree

***M*** is a number of nodes in the q tree

**Time Complexity**: ---------------------------------------------------------------------------------------------

**Worst Case**: Fully Skewed Tree OR Perfectly Balanced Tree – O(N)

We will visit each node exactly once to check if every node is equal.

It does not matter if the tree is skewed or balanced, it will have the same runtime.

**Worst Case Technicality:** The minimum height between the two trees

Technically, if the trees are not equal, then the time complexity would be

O(Math.min(N, M))

**Space Complexity**: --------------------------------------------------------------------------------------------

**Average Case**: Perfectly Balanced Tree – O(N/2) = O(N)

There will be up to N/2 nodes on the stack when we are on the last level.