

切换主题：

默认主题

▼

# 题目地址（100. 相同的树）

https://leetcode-cn.com/problems/same-tree/

## 标签

- DFS
- BFS
- 树

## 难度

- 简单

## 入选理由

- 树的题目的一个中心是什么？
- 难度仍然是 easy，今天继续给大家平和一下
- 由于树的结构的递归性，树的题目用来练习问题分解很有用，本题也是一样

## 题目描述

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1:

输入：

1

/ \

2 3

1

/ \

2 3

[1,2,3], [1,2,3]

输出：true

示例 2:

输入：

1

1

```

      /      \
     2        2

[1,2],    [1,null,2]

```

输出: `false`

示例 3:

```

输入:      1      1
         / \    / \
        2  1  1  2

[1,2,1],  [1,1,2]

```

输出: `false`

## 前置知识

- 递归
- 层序遍历
- 前中序确定一棵树

## 递归

## 思路

最简单的想法是递归，这里先介绍下递归三要素

- 递归出口，问题最简单的情况
- 递归调用总是去尝试解决更小的问题，这样问题才会被收敛到最简单的情况
- 递归调用的父问题和子问题没有交集

尝试用递归去解决相同的树

1. 分解为子问题，相同的树分解为左子是否相同，右子是否相同
2. 递归出口: 当树高度为 1 时，判断递归出口

## 代码

语言支持: JS, Python3, Java, CPP

JS Code:

```

var isSameTree = function (p, q) {
  if (!p || !q) {
    return !p && !q;
  }
  return (
    p.val === q.val &&
    isSameTree(p.left, q.left) &&
    isSameTree(p.right, q.right)
  );
};

```

Python3 Code:

```

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not p and not q:
            return True
        if not p or not q:
            return False
        return p.val == q.val and self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

```

Java Code:

```

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;
        if (p == null || q == null) return false;
        if (p.val != q.val) return false;
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}

```

C++ Code:

```

class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (p == nullptr && q == nullptr) return true;
        if (p == nullptr && q != nullptr) return false;
        if (p != nullptr && q == nullptr) return false;
        return p->val == q->val && isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
}

```

```
};
```

## 复杂度分析

- 时间复杂度： $O(N)$ ，其中  $N$  为树的节点数。
- 空间复杂度： $O(h)$ ，其中  $h$  为树的高度。

## 层序遍历

### 思路

判断两棵树是否相同，只需要判断树的整个结构相同，判断树的结构是否相同，只需要判断树的每层内容是否相同。

我们可以借助队列实现层次遍历，并在每次取队头元素的时候比较两棵树的队头元素是否一致。如果不一致，直接返回 false。如果访问完都没有发现不一致就返回 true。

### 代码

语言支持：JS, Python3, Java, C++

JS Code:

```
var isSameTree = function (p, q) {  
    let curLevelA = [p];  
    let curLevelB = [q];  
  
    while (curLevelA.length && curLevelB.length) {  
        let nextLevelA = [];  
        let nextLevelB = [];  
        const isOK = isSameCurLevel(curLevelA, curLevelB, nextLevelA, nextLevelB);  
        if (isOK) {  
            curLevelA = nextLevelA;  
            curLevelB = nextLevelB;  
        } else {  
            return false;  
        }  
    }  
  
    return true;  
};  
  
function isSameCurLevel(curLevelA, curLevelB, nextLevelA, nextLevelB) {  
    if (curLevelA.length !== curLevelB.length) {  
        return false;  
    }  
}
```

```

for (let i = 0; i < curLevelA.length; i++) {
  if (!isSameNode(curLevelA[i], curLevelB[i])) {
    return false;
  }
  curLevelA[i] && nextLevelA.push(curLevelA[i].left, curLevelA[i].right);
  curLevelB[i] && nextLevelB.push(curLevelB[i].left, curLevelB[i].right);
}
return true;
}

function isSameNode(nodeA, nodeB) {
  if (!nodeA || !nodeB) {
    return nodeA === nodeB;
  }
  return nodeA.val === nodeB.val;
  // return nodeA === nodeB || (nodeA && nodeB && nodeA.val === nodeB.val);
}

```

Python3 Code:

```

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not q and not p:
            return True
        if not q or not p:
            return False
        q1 = collections.deque([p])
        q2 = collections.deque([q])
        while q1 and q2:
            node1 = q1.popleft()
            node2 = q2.popleft()
            if node1.val != node2.val:
                return False
            left1, right1 = node1.left, node1.right
            left2, right2 = node2.left, node2.right
            if (not left1) ^ (not left2):
                return False
            if (not right1) ^ (not right2):
                return False
            if left1:
                q1.append(left1)
            if right1:
                q1.append(right1)
            if left2:
                q2.append(left2)
            if right2:
                q2.append(right2)
        return not q1 and not q2

```

Java Code:

```

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        Queue<TreeNode> queue = new LinkedList<>();
        if (p == null && q == null) return true;
        if (p == null || q == null) return false;
        queue.offer(p);
        queue.offer(q);

        while (!queue.isEmpty()) {
            TreeNode first = queue.poll();
            TreeNode second = queue.poll();
            if (first == null && second == null) continue;
            if (first == null || second == null) return false;
            if (first.val != second.val) return false;
            queue.offer(first.left);
            queue.offer(second.left);
            queue.offer(first.right);
            queue.offer(second.right);
        }
        return true;
    }
}

```

CPP Code:

```

class Solution {
public:

    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (p == NULL && q == NULL) return true;
        if (p == NULL || q == NULL) return false;
        queue<TreeNode*> que;
        que.push(p);    //
        que.push(q);    //
        while (!que.empty()) {    //
            TreeNode* leftNode = que.front(); que.pop();
            TreeNode* rightNode = que.front(); que.pop();
            if (!leftNode && !rightNode) {    //
                continue;
            }
            //
            if ((!leftNode || !rightNode || (leftNode->val != rightNode->val))) {
                return false;
            }
            que.push(leftNode->left);    //

```

```
        que.push(rightNode->left); //
        que.push(leftNode->right); //
        que.push(rightNode->right); //
    }
    return true;
}
};
```

## 复杂度分析

- 时间复杂度： $O(N)$ ，其中  $N$  为树的节点数。
- 空间复杂度： $O(Q)$ ，其中  $Q$  为队列的长度最大值，在这里不会超过相邻两层的节点数的最大值。

## 前中序确定一棵树

### 思路

前序和中序的遍历结果确定一棵树，那么当两棵树前序遍历和中序遍历结果都相同，那是否说明两棵树也相同。

### 代码

语言支持：JS, Java

JS Code:

```
var isSameTree = function (p, q) {
    const preorderP = preorder(p, []);
    const preorderQ = preorder(q, []);
    const inorderP = inorder(p, []);
    const inorderQ = inorder(q, []);
    return (
        preorderP.join("") === preorderQ.join("") &&
        inorderP.join("") === inorderQ.join("")
    );
};

function preorder(root, arr) {
    if (root === null) {
        arr.push(" ");
        return arr;
    }
    arr.push(root.val);
    preorder(root.left, arr);
    preorder(root.right, arr);
    return arr;
}
```

```

}

function inorder(root, arr) {
  if (root === null) {
    arr.push(" ");
    return arr;
  }
  inorder(root.left, arr);
  arr.push(root.val);
  inorder(root.right, arr);
  return arr;
}

```

Java Code:

```

/**
 * 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) {
        // preorder
        List<Integer> pretraversalP = new ArrayList<>();
        List<Integer> pretraversalQ = new ArrayList<>();
        preorder(p, pretraversalP);
        preorder(q, pretraversalQ);

        // inorder
        List<Integer> intraversalP = new ArrayList<>();
        List<Integer> intraversalQ = new ArrayList<>();
        inorder(p, intraversalP);
        inorder(q, intraversalQ);

        return (pretraversalP+ "").equals((pretraversalQ+ "")) && (intraversalP+ "").equals((intraversalQ+ ""));
    }
}

```



```
private void preorder(TreeNode root, List<Integer> traversal) {  
    if (root == null) {  
        traversal.add(null);  
        return;  
    }  
    traversal.add(root.val);  
    preorder(root.left, traversal);  
    preorder(root.right, traversal);  
}  
  
private void inorder(TreeNode root, List<Integer> traversal) {  
    if (root == null) {  
        traversal.add(null);  
        return;  
    }  
    preorder(root.left, traversal);  
    traversal.add(root.val);  
    preorder(root.right, traversal);  
}  
}
```

## 复杂度分析

- 时间复杂度： $O(N)$ ，其中  $N$  为树的节点数。
- 空间复杂度：使用了中序遍历的结果数组，因此空间复杂度为  $O(N)$ ，其中  $N$  为树的节点数。

[上一页](#)[下一页](#)

© 2020 lucifer. 保留所有权利