二叉树的最近公共祖先：  
/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

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

\* }

\*/

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

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

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

if(left == null && right == null) return null; // 1.

if(left == null) return right; // 3.

if(right == null) return left; // 4.

return root; // 2. if(left != null and right != null)

}

}

**小结**：目前完全是照搬答案，没看懂题目是首先，

LeetCode 105 从前序和中序遍历构造二叉树  
class Solution {

private Map<Integer, Integer> indexMap;

public TreeNode myBuildTree(int[] preorder, int[] inorder, int preorder\_left, int preorder\_right, int inorder\_left, int inorder\_right) {

if (preorder\_left > preorder\_right) {

return null;

}

// 前序遍历中的第一个节点就是根节点

int preorder\_root = preorder\_left;

// 在中序遍历中定位根节点

int inorder\_root = indexMap.get(preorder[preorder\_root]);

// 先把根节点建立出来

TreeNode root = new TreeNode(preorder[preorder\_root]);

// 得到左子树中的节点数目

int size\_left\_subtree = inorder\_root - inorder\_left;

// 递归地构造左子树，并连接到根节点

// 先序遍历中「从 左边界+1 开始的 size\_left\_subtree」个元素就对应了中序遍历中「从 左边界 开始到 根节点定位-1」的元素

root.left = myBuildTree(preorder, inorder, preorder\_left + 1, preorder\_left + size\_left\_subtree, inorder\_left, inorder\_root - 1);

// 递归地构造右子树，并连接到根节点

// 先序遍历中「从 左边界+1+左子树节点数目 开始到 右边界」的元素就对应了中序遍历中「从 根节点定位+1 到 右边界」的元素

root.right = myBuildTree(preorder, inorder, preorder\_left + size\_left\_subtree + 1, preorder\_right, inorder\_root + 1, inorder\_right);

return root;

}

public TreeNode buildTree(int[] preorder, int[] inorder) {

int n = preorder.length;

// 构造哈希映射，帮助我们快速定位根节点

indexMap = new HashMap<Integer, Integer>();

for (int i = 0; i < n; i++) {

indexMap.put(inorder[i], i);

}

return myBuildTree(preorder, inorder, 0, n - 1, 0, n - 1);

}

}

Leetcode 77组合  
import java.util.ArrayDeque;

import java.util.ArrayList;

import java.util.Deque;

import java.util.List;

public class Solution {

public List<List<Integer>> combine(int n, int k) {

List<List<Integer>> res = new ArrayList<>();

if (k <= 0 || n < k) {

return res;

}

// 从 1 开始是题目的设定

Deque<Integer> path = new ArrayDeque<>();

dfs(n, k, 1, path, res);

return res;

}

private void dfs(int n, int k, int begin, Deque<Integer> path, List<List<Integer>> res) {

// 递归终止条件是：path 的长度等于 k

if (path.size() == k) {

res.add(new ArrayList<>(path));

return;

}

// 遍历可能的搜索起点

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

// 向路径变量里添加一个数

path.addLast(i);

// 下一轮搜索，设置的搜索起点要加 1，因为组合数理不允许出现重复的元素

dfs(n, k, i + 1, path, res);

// 重点理解这里：深度优先遍历有回头的过程，因此递归之前做了什么，递归之后需要做相同操作的逆向操作

path.removeLast();

}

}

}

总结：递归还需要时间，作业完成有难度。