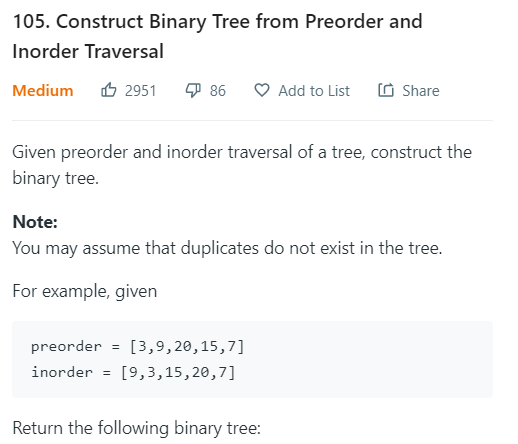
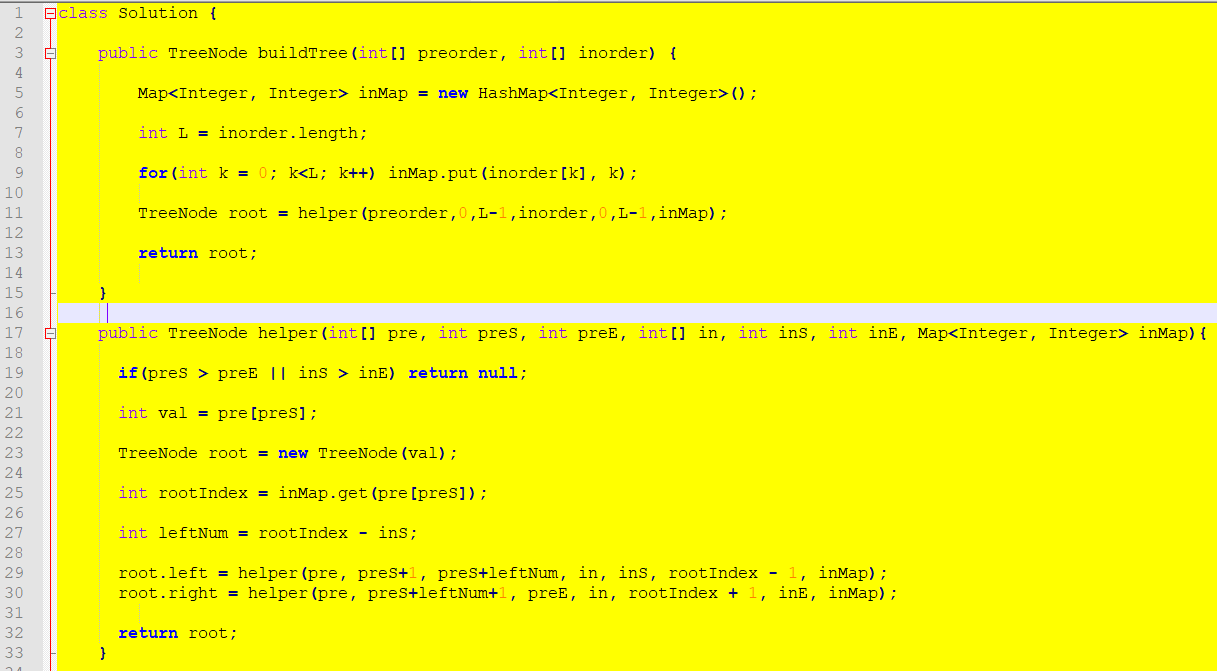
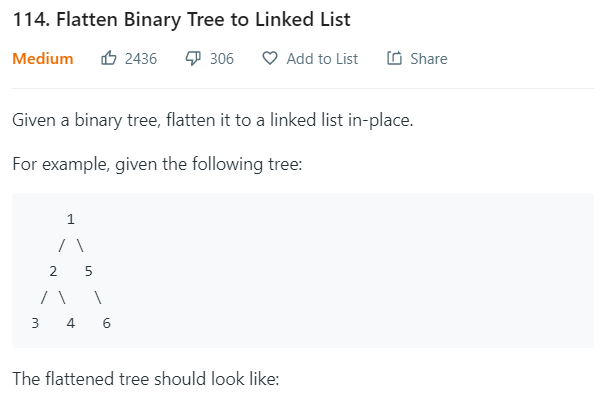
---------------------------------------------------------------------------------------------------------





---------------------------------------------------------------------------------------------------------



class Solution **{**

public TreeNode prev **=** **null;**

public void flatten**(**TreeNode root**)** **{**

**if** **(**root **==** **null)** **return;**

flatten**(**root**.**right**);** // last prev == root.right

flatten**(**root**.**left**);** // go deep into the most right

root**.**right **=** prev**;**

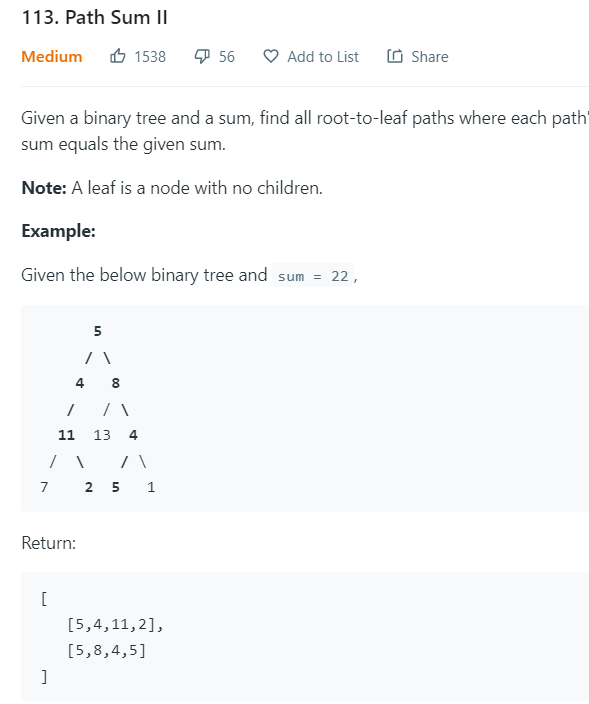
root**.**left **=** **null;**

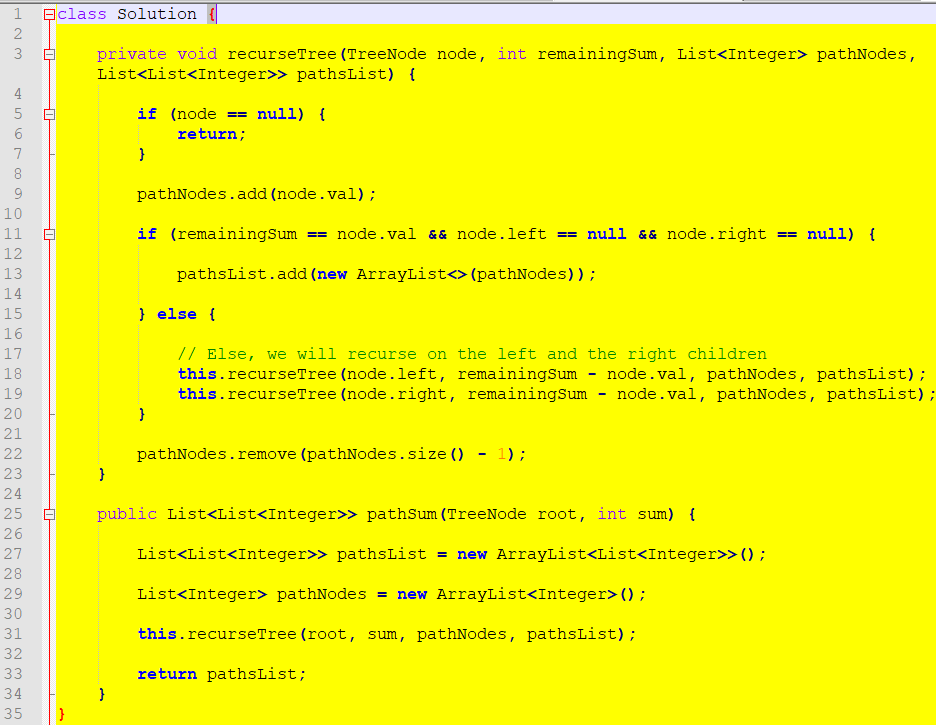
prev **=** root**;**

**}**

**}**

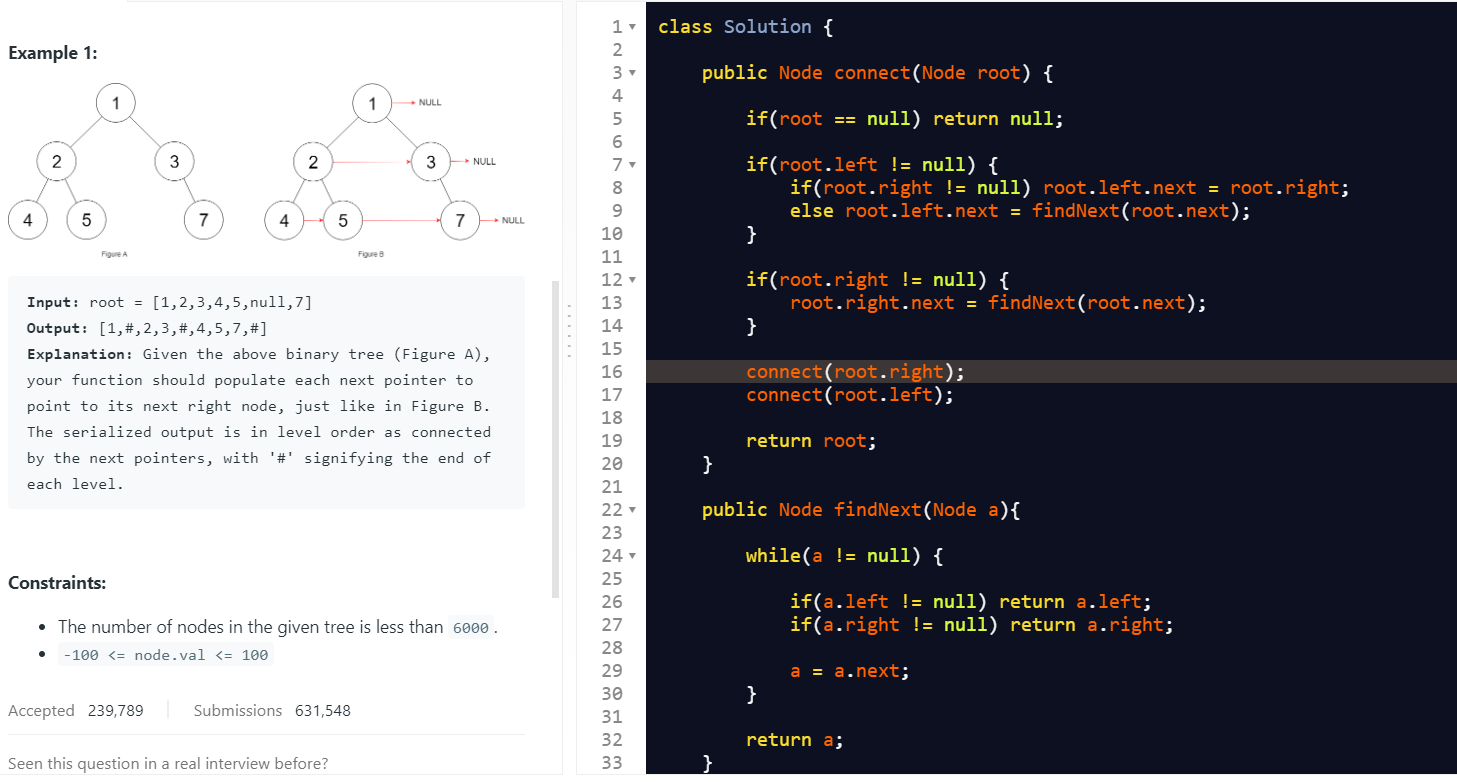
---------------------------------------------------------------------------------------------------------





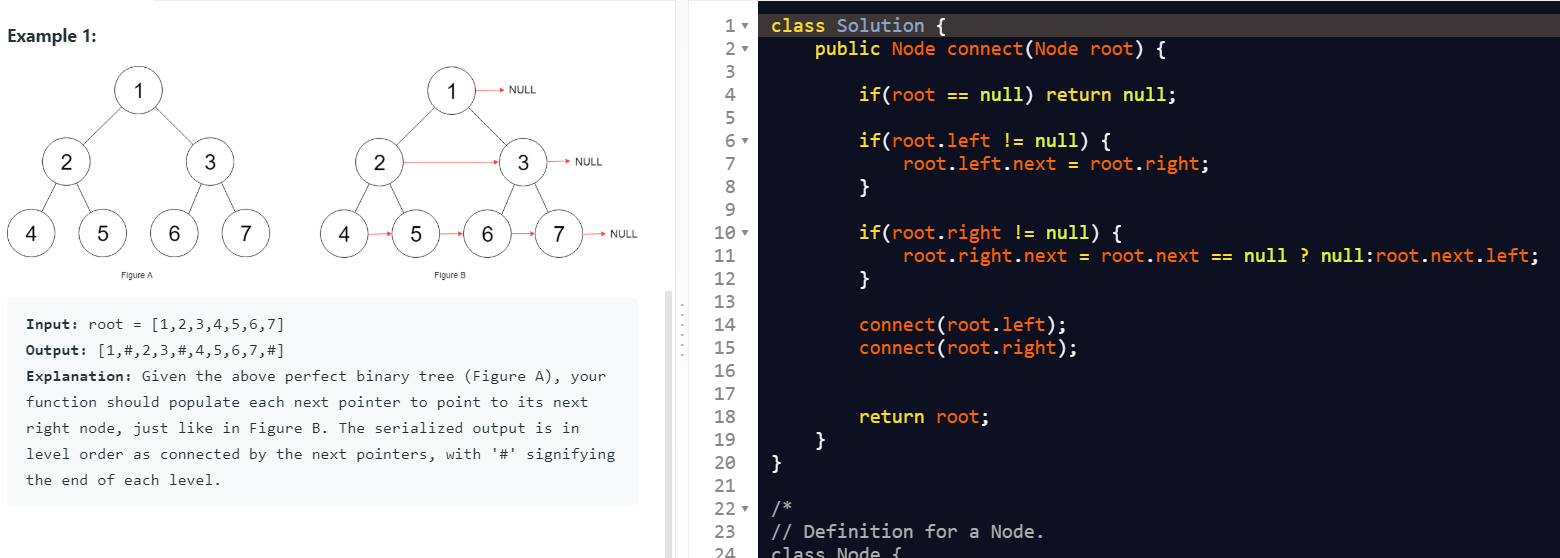
---------------------------------------------------------------------------------------------------------

LeetCode 117

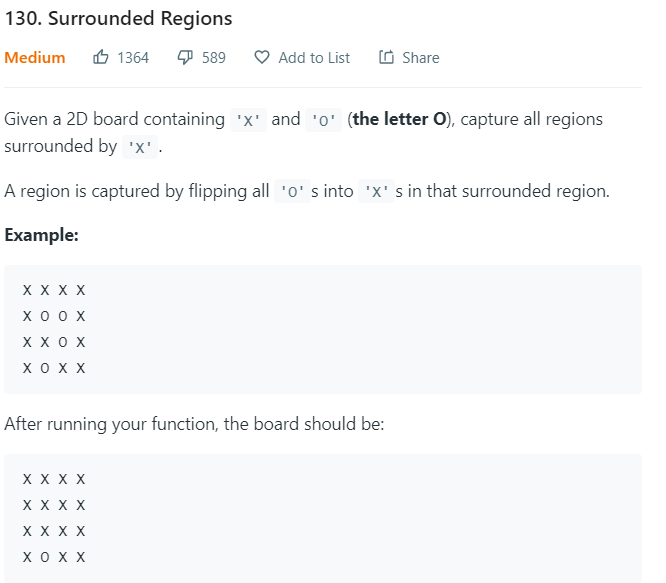


---------------------------------------------------------------------------------------------------------

116



---------------------------------------------------------------------------------------------------------



**public class Solution {**

**protected Integer ROWS = 0;**

**protected Integer COLS = 0;**

**public void solve(char[][] board) {**

**if (board == null || board.length == 0) {**

**return;**

**}**

**this.ROWS = board.length;**

**this.COLS = board[0].length;**

**List<Pair<Integer, Integer>> borders = new LinkedList<Pair<Integer, Integer>>();**

**// Step 1). construct the list of border cells**

**for (int r = 0; r < this.ROWS; ++r) {**

**borders.add(new Pair(r, 0));**

**borders.add(new Pair(r, this.COLS - 1));**

**}**

**for (int c = 0; c < this.COLS; ++c) {**

**borders.add(new Pair(0, c));**

**borders.add(new Pair(this.ROWS - 1, c));**

**}**

**// Step 2). mark the escaped cells**

**for (Pair<Integer, Integer> pair : borders) {**

**this.DFS(board, pair.first, pair.second);**

**}**

**// Step 3). flip the cells to their correct final states**

**for (int r = 0; r < this.ROWS; ++r) {**

**for (int c = 0; c < this.COLS; ++c) {**

**if (board[r][c] == 'O')**

**board[r][c] = 'X';**

**if (board[r][c] == 'E')**

**board[r][c] = 'O';**

**}**

**}**

**}**

**protected void DFS(char[][] board, int row, int col) {**

**if (board[row][col] != 'O')**

**return;**

**board[row][col] = 'E';**

**if (col < this.COLS - 1)**

**this.DFS(board, row, col + 1);**

**if (row < this.ROWS - 1)**

**this.DFS(board, row + 1, col);**

**if (col > 0)**

**this.DFS(board, row, col - 1);**

**if (row > 0)**

**this.DFS(board, row - 1, col);**

**}**

**}**

**class Pair<U, V> {**

**public U first;**

**public V second;**

**public Pair(U first, V second) {**

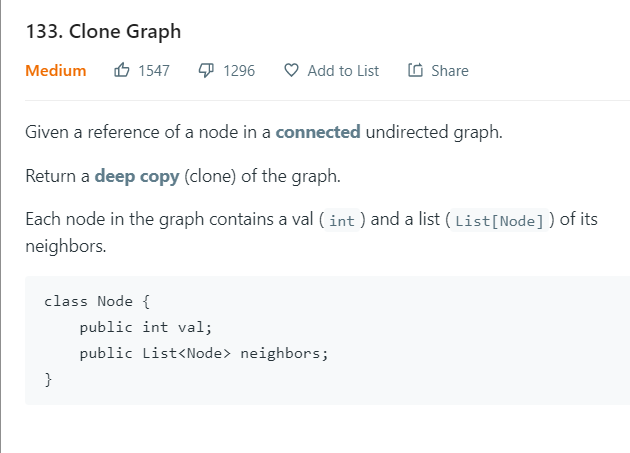
**this.first = first;**

**this.second = second;**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**class Solution {**

**private HashMap <Node, Node> visited = new HashMap <> ();**

**public Node cloneGraph(Node node) {**

**if (node == null) {**

**return node;**

**}**

**if (visited.containsKey(node)) {**

**return visited.get(node);**

**}**

**Node cloneNode = new Node(node.val, new ArrayList());**

**visited.put(node, cloneNode);**

**for (Node neighbor: node.neighbors) {**

**cloneNode.neighbors.add(cloneGraph(neighbor));**

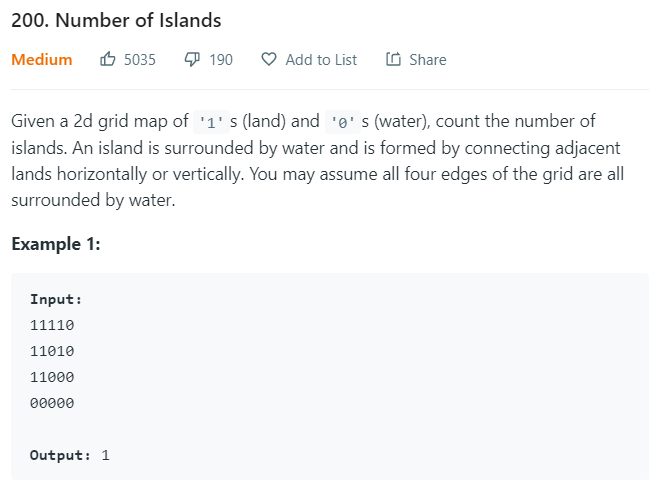
**}**

**return cloneNode;**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**public class Solution {**

**private int n;**

**private int m;**

**public int numIslands(char[][] grid) {**

**int count = 0;**

**n = grid.length;**

**if (n == 0) return 0;**

**m = grid[0].length;**

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

**for (int j = 0; j < m; j++)**

**if (grid[i][j] == '1') {**

**DFS(grid, i, j);**

**++count;**

**}**

**}**

**return count;**

**}**

**private void DFS(char[][] grid, int i, int j) {**

**if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1') return;**

**grid[i][j] = '0';**

**DFS(grid, i + 1, j);**

**DFS(grid, i - 1, j);**

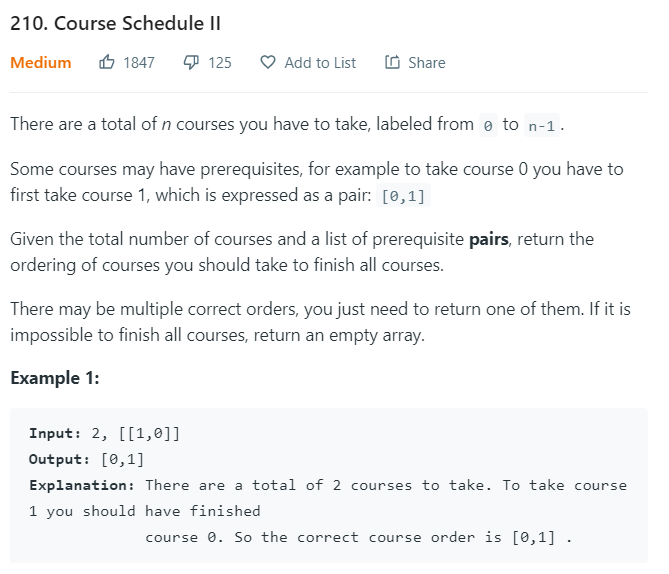
**DFS(grid, i, j + 1);**

**DFS(grid, i, j - 1);**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**class Solution {**

**private int N = 0;**

**public int[] findOrder(int numCourses, int[][] prerequisites) {**

**int[] result = new int[numCourses];**

**Course[] courses = new Course[numCourses];**

**for (int i = 0; i < numCourses; i++) {**

**courses[i] = new Course(i);**

**}**

**for (int i = 0; i < prerequisites.length; i++) {**

**courses[prerequisites[i][0]].add(courses[prerequisites[i][1]]);**

**}**

**for (int i = 0; i < numCourses; i++) {**

**if (isCyclic(courses[i], result)) return new int[0]; // return emtpy array if cyclic**

**}**

**return result;**

**}**

**private boolean isCyclic(Course cur, int[] result) { // if cyclic, then no solution**

**if (cur.tested == true) return false; // Check if cyclic**

**if (cur.visited == true) return true; // visited before --> cyclic!!!**

**cur.visited = true; // confirmed visited**

**for (Course c: cur.pre) {**

**if (isCyclic(c, result)) {**

**return true;**

**}**

**}**

**cur.tested = true; // confirmed already tested OK**

**result[N++] = cur.number; // build such result**

**return false;**

**}**

**class Course {**

**boolean visited = false;**

**boolean tested = false;**

**int number;**

**List < Course > pre = new ArrayList < Course > ();**

**public Course(int i) {**

**number = i;**

**}**

**public void add(Course c) {**

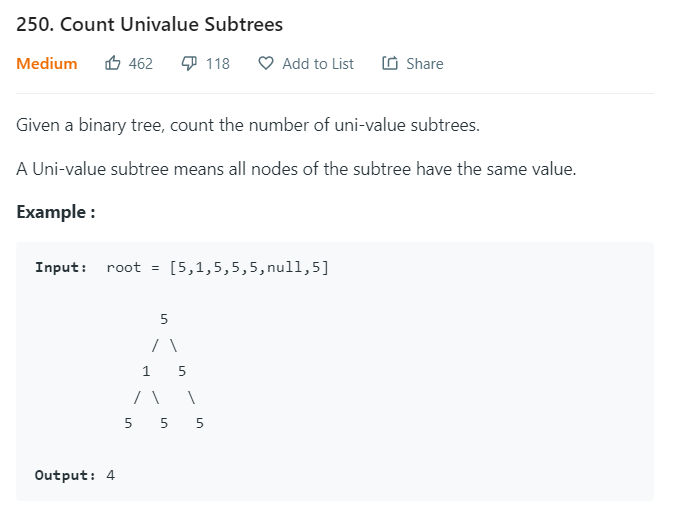
**pre.add(c);**

**}**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**class Solution {**

**int count = 0;**

**public int countUnivalSubtrees(TreeNode root) {**

**if (root == null) return 0;**

**is\_uni(root);**

**return count;**

**}**

**boolean is\_uni(TreeNode node) {**

**//base case - if the node has no children this is a univalue subtree 如果兩邊都是空，那麼這個節點，自己算一個**

**if (node.left == null && node.right == null) {**

**count++;**

**return true;**

**}**

**/\*\* 2. 製造參數 "is\_unival" \*\*/**

**boolean is\_unival = true;**

**/\*\* check if all of the node's children are univalue subtrees and if they have the same value also recursively call is\_uni for children \*/**

**/\*\* 3. 檢查左邊 \*\*/**

**if (node.left != null) {**

**is\_unival = is\_uni(node.left) /\*\* \*\*/ && is\_unival /\*\* 是否本身是True \*\*/ && node.left.val == node.val /\*\* 是否左節點，等於根節點 \*\*/;**

**}**

**/\*\* 3. 檢查右邊 \*\*/**

**if (node.right != null) {**

**is\_unival = is\_uni(node.right) && is\_unival && node.right.val == node.val;**

**}**

**/\*\* 4. 返回錯誤 \*\*/**

**// return if a univalue tree exists here and increment if it does**

**if (!is\_unival) return false;**

**/\*\* 5. 返回加一 \*\*/**

**count++;**

**return true;**

**}**

**}**

**/\*\***

**\* Definition for a binary tree node.**

**\* public class TreeNode {**

**\* int val;**

**\* TreeNode left;**

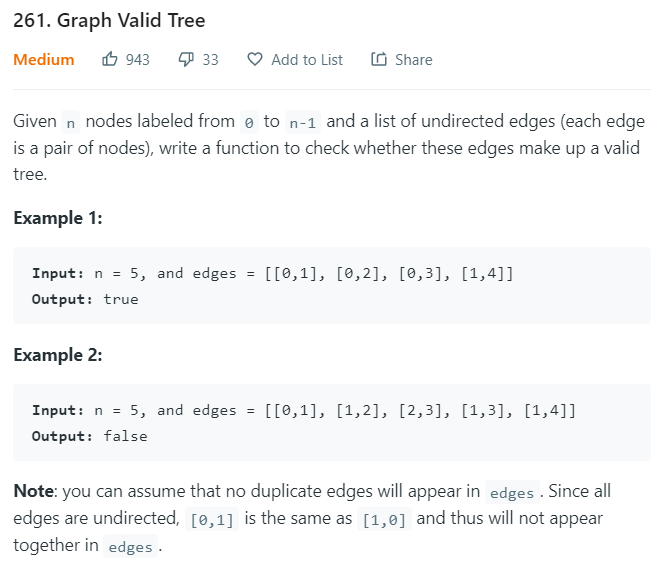
**\* TreeNode right;**

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

**\* }**

**\*/**

---------------------------------------------------------------------------------------------------------



**public class Solution {**

**public boolean validTree(int n, int[][] edges) {**

**// build the graph**

**List<List<Integer>> adjList = new ArrayList<List<Integer>>(n);**

**// initialize vertices**

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

**adjList.add(i, new ArrayList<Integer>());**

**// add edges**

**for (int i = 0; i < edges.length; i++) {**

**int u = edges[i][0], v = edges[i][1];**

**adjList.get(u).add(v);**

**adjList.get(v).add(u);**

**}**

**// build the visited**

**boolean[] visited = new boolean[n];**

**// make sure there's no cycle**

**if (hasCycle(adjList, visited, 0, -1))**

**return false;**

**// make sure all vertices are connected**

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

**if (!visited[i])**

**return false;**

**}**

**return true;**

**}**

**// check if an undirected graph has cycle started from vertex u**

**boolean hasCycle(List<List<Integer>> adjList, boolean[] visited, int u, int parent) {**

**visited[u] = true;**

**// Get adjacent nodes**

**for (int i = 0; i < adjList.get(u).size(); i++) {**

**int v = adjList.get(u).get(i);**

**// parent != v --> because v is a node, and v's parent is also his adjacent**

**if ( (visited[v] && parent != v) || (!visited[v] && hasCycle(adjList, visited, v, u)) )**

**return true;**

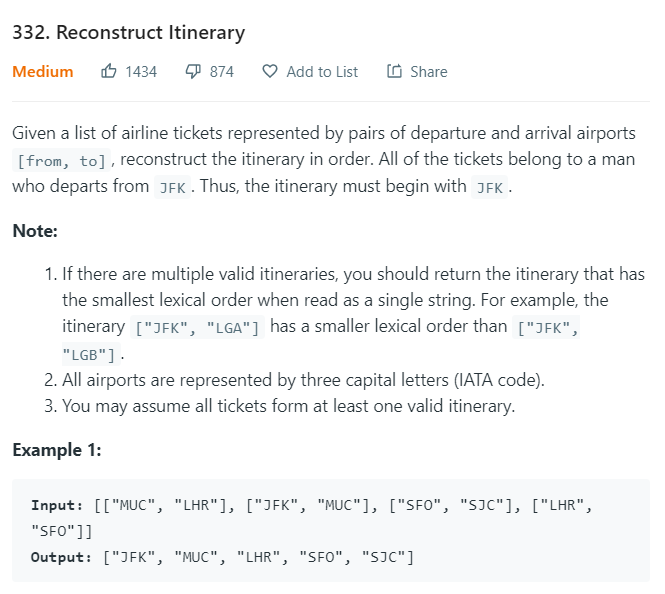
**}**

**return false;**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**public class Solution {**

**Map<String, PriorityQueue<String>> flights;**

**LinkedList<String> path;**

**public List<String> findItinerary(List<List<String>> tickets) {**

**flights = new HashMap<>();**

**path = new LinkedList<>();**

**for (List<String> ticket : tickets) {**

**flights.putIfAbsent(ticket.get(0), new PriorityQueue<>());**

**flights.get(ticket.get(0)).add(ticket.get(1));**

**}**

**dfs("JFK");**

**return path;**

**}**

**public void dfs(String departure) {**

**PriorityQueue<String> arrivals = flights.get(departure);**

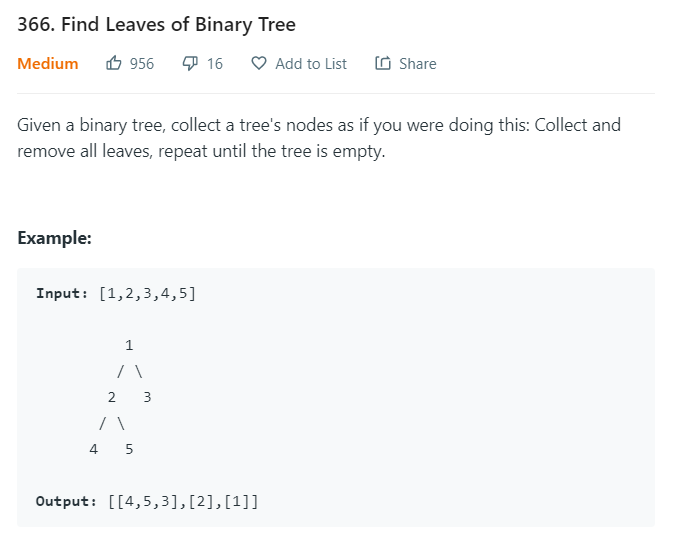
**while (arrivals != null && !arrivals.isEmpty()) dfs(arrivals.poll());**

**path.addFirst(departure);**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**public class Solution {**

**public List<List<Integer>> findLeaves(TreeNode root) {**

**Map<Integer, List<Integer>> res = new HashMap<>();**

**dfs(root, res);**

**return new ArrayList<>(res.values());**

**}**

**private int dfs(TreeNode node, Map<Integer, List<Integer>> res){**

**if(null==node) return 0;**

**int level = 1 + Math.max( dfs(node.left, res), dfs(node.right, res) );**

**res.putIfAbsent(level, new ArrayList<>());**

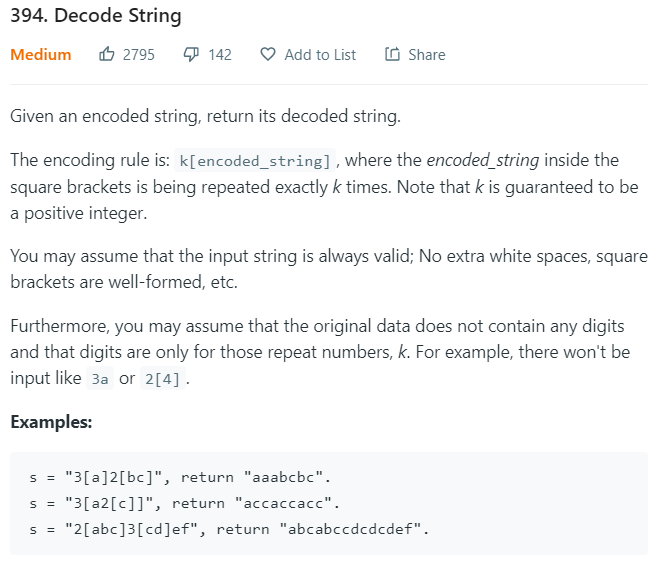
**res.get(level).add(node.val);**

**return level;**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**class Solution {**

**private int i = 0; /// 全局变量**

**public String decodeString(String s) { // 自己拿笔划一下!!**

**StringBuilder sb = new StringBuilder(); // 初始化**

**int num = 0; // 这里的num是重复的次数**

**for (; i < s.length(); i++) { // 循环每个字母**

**char c = s.charAt(i); // 拿到当前字母**

**/\*\* 如果是左括号 \*/**

**if (c == '[') {**

**/\*\* i 往后移一位 \*/**

**i++;**

**String str = decodeString(s); // 注意这里的 i 是全局变量!!! DFS的时候，i已经更新了**

**for (int k = 0; k < num; k++) sb.append(str);**

**num = 0; // num是存重复的次数 --> 使用完一次后归零**

**/\*\* 如果是右括号 \*/**

**} else if (c == ']') {**

**return sb.toString();**

**/\*\* 如果是数字 \*/**

**} else if (c >= '0' && c <= '9') {**

**num = num \* 10 + c - '0'; // 比如如果数字是 45， 那么先有4， 再来一个5**

**/\*\* 如果是字母 \*/**

**} else {**

**sb.append(c);**

**}**

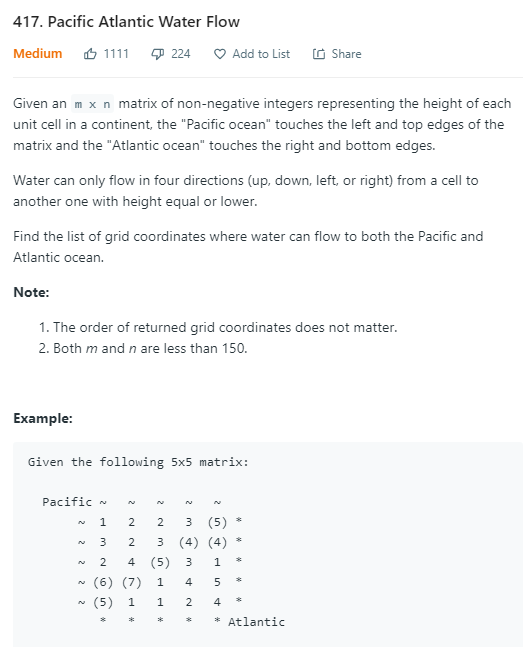
**}**

**return sb.toString();**

**}**

**}**

---------------------------------------------------------------------------------------------------------



**class Solution {**

**public List<List<Integer>> pacificAtlantic(int[][] matrix) {**

**List<List<Integer>> res = new LinkedList<>();**

**if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return res;**

**int n = matrix.length;**

**int m = matrix[0].length;**

**boolean[][] pacific = new boolean[n][m];**

**boolean[][] atlantic = new boolean[n][m];**

**for(int i=0; i<n; i++){ ///// 高度**

**dfs(matrix, n, m, pacific, Integer.MIN\_VALUE, i, 0); /\*\* 最上面一排 \*/**

**dfs(matrix, n, m, atlantic, Integer.MIN\_VALUE, i, m-1); /\*\* 最下面一排 \*/**

**}**

**for(int i=0; i<m; i++){ /\*\* 宽度 \*/**

**dfs(matrix, n, m, pacific, Integer.MIN\_VALUE, 0, i); /\*\* 最左一列 \*/**

**dfs(matrix, n, m, atlantic, Integer.MIN\_VALUE, n-1, i); /\*\* 最右一列 \*/**

**}**

**/\*\* 找出两边，都有的点 \*/**

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

**for (int j = 0; j < m; j++)**

**if (pacific[i][j] && atlantic[i][j]) {**

**List<Integer> x = new LinkedList<Integer>();**

**x.add(i);**

**x.add(j);**

**res.add(x);**

**}**

**return res;**

**}**

**int[][] dir = new int[][]{{0,1},{0,-1},{1,0},{-1,0}};**

**public void dfs(int[][] matrix, int n, int m, boolean[][] visited, int height, int x, int y){**

**if(x<0 || x>=n || y<0 || y>=m || visited[x][y] || height > matrix[x][y]) return;**

**visited[x][y] = true;**

**for(int[] d:dir){**

**dfs(matrix, n, m, visited, matrix[x][y], x+d[0], y+d[1]);**

**}**

**}**

**}**

---------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------