**TREES**

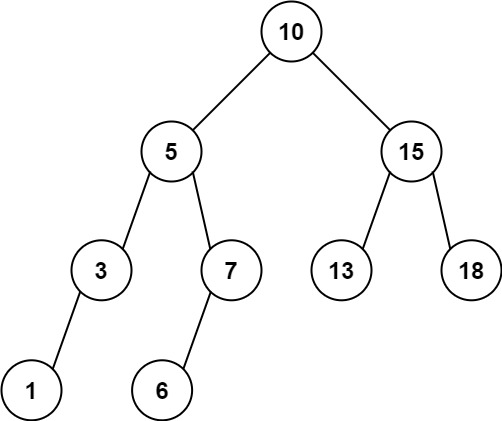
1. **Range Sum Of BST (Leetcode – 938) (Easy)**

Given the root node of a binary search tree and two integers low and high, return the sum of values of all nodes with a value in the inclusive range [low, high].

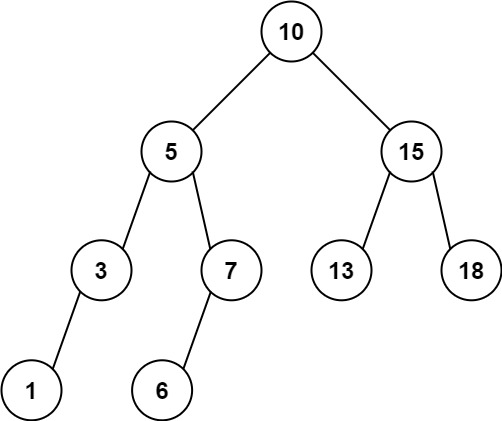
Example 1:

Input: root = [10,5,15,3,7,null,18], low = 7, high = 15

Output: 32

Explanation: Nodes 7, 10, and 15 are in the range [7, 15]. 7 + 10 + 15 = 32.

Example 2:



Input: root = [10,5,15,3,7,13,18,1,null,6], low = 6, high = 10

Output: 23

Explanation: Nodes 6, 7, and 10 are in the range [6, 10]. 6 + 7 + 10 = 23.

Constraints:

The number of nodes in the tree is in the range [1, 2 \* 104].

1 <= Node.val <= 105

1 <= low <= high <= 105

All Node.val are unique.

**Soution :**

/\*--------------------------

Time Complexity: O(n)

Space Complexity: O(n)

---------------------------\*/

class Solution {

public int rangeSumBST(TreeNode root, int low, int high) {

if(root == null)

return 0;

if(root.val > high)

return rangeSumBST(root.left, low, high);

if(root.val < low)

return rangeSumBST(root.right, low, high);

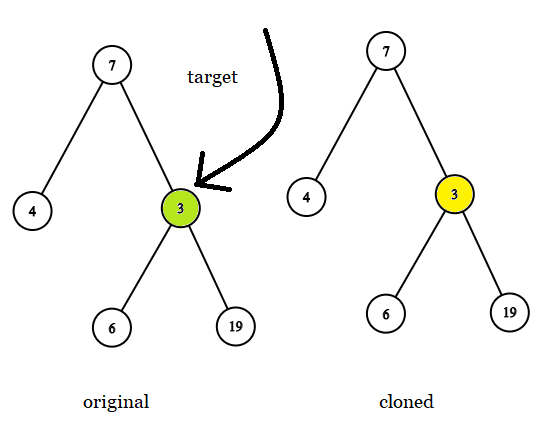
return root.val + rangeSumBST(root.left, low, high) + rangeSumBST(root.right, low, high);

}

}

**2)** **Find a Corresponding Node of a Binary Tree in a Clone of That Tree (Leetcode – 1379)**

Given two binary trees original and cloned and given a reference to a node target in the original tree.The cloned tree is a copy of the original tree.Return a reference to the same node in the cloned tree.Note that you are not allowed to change any of the two trees or the target node and the answer must be a reference to a node in the cloned tree.

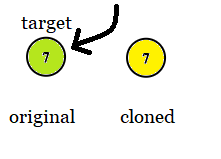
Example 1:

Input: tree = [7,4,3,null,null,6,19], target = 3

Output: 3

Explanation: In all examples the original and cloned trees are shown. The target node is a green node from the original tree. The answer is the yellow node from the cloned tree.

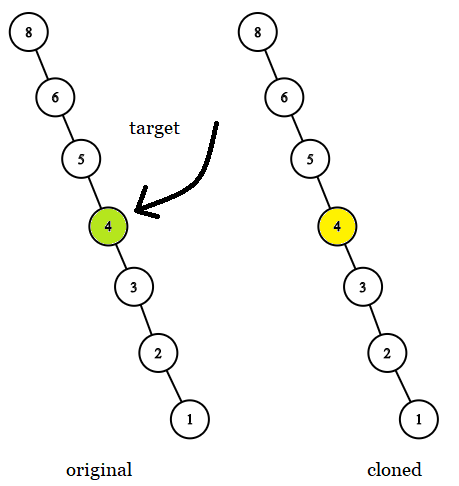
Example 2:



Input: tree = [7], target = 7

Output: 7

Example 3:



Input: tree = [8,null,6,null,5,null,4,null,3,null,2,null,1], target = 4

Output: 4

Constraints:

The number of nodes in the tree is in the range [1, 104].

The values of the nodes of the tree are unique.

target node is a node from the original tree and is not null.

**Solution :**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode(int x) { val = x; }

 \* }

 \*/

class Solution {

    public final TreeNode getTargetCopy(final TreeNode original, final TreeNode cloned, final TreeNode target) {

        if (original == null) return null;

        if (original == target)  return cloned;

        TreeNode left = getTargetCopy(original.left, cloned.left, target);

        if (left != null) return left;

        return getTargetCopy(original.right, cloned.right, target);

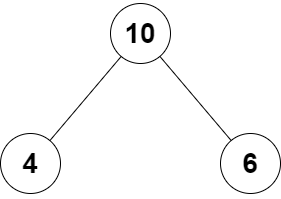
    }

}

**3)Root Equals Sum of Children (Leetcode - 2236)**

You are given the root of a binary tree that consists of exactly 3 nodes: the root, its left child, and its right child.Return true if the value of the root is equal to the sum of the values of its two children, or false otherwise.

Example 1:



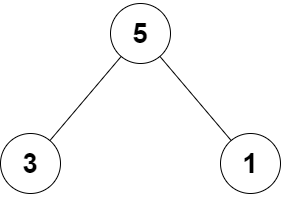
Input: root = [10,4,6]

Output: true

Explanation: The values of the root, its left child, and its right child are 10, 4, and 6, respectively.

10 is equal to 4 + 6, so we return true.

Example 2:



Input: root = [5,3,1]

Output: false

Explanation: The values of the root, its left child, and its right child are 5, 3, and 1, respectively.

5 is not equal to 3 + 1, so we return false.

Constraints:

The tree consists only of the root, its left child, and its right child.

-100 <= Node.val <= 100

**Solutions :**

class Solution {

    public boolean checkTree(TreeNode root) {

       return root.val == root.right.val + root.left.val;

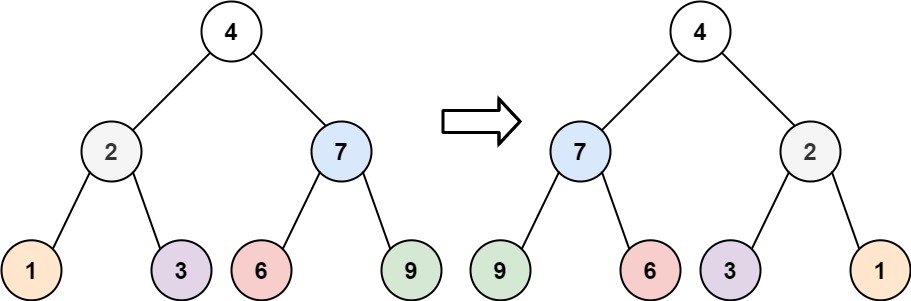
    }

}

**4 ) Invert Binary Tree (Leetcode – 226)**

Given the root of a binary tree, invert the tree, and return its root.

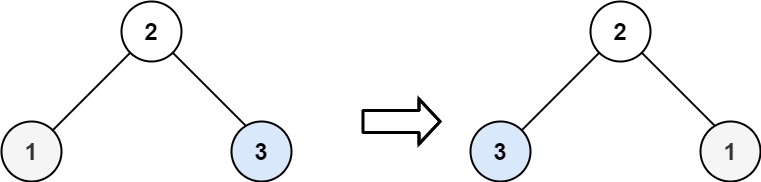
Example 1:



Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

Example 2:



Input: root = [2,1,3]

Output: [2,3,1]

Example 3:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

**Solution:**

class Solution {

    public TreeNode invertTree(TreeNode root) {

        if(root==null) return root;

        TreeNode temp=root.left;

        root.left=root.right;

        root.right=temp;

        invertTree(root.left);

        invertTree(root.right);

        return root;

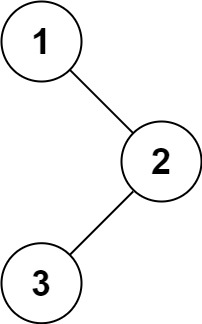
    }

}

**5) Binary Tree Inorder Traversal (Leetcode - 94)**

Given the root of a binary tree, return the inorder traversal of its nodes' values.

Example 1:



Input: root = [1,null,2,3]

Output: [1,3,2]

Example 2:

Input: root = []

Output: []

Example 3:

Input: root = [1]

Output: [1]

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

**Solution :**

class Solution {

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

    public List<Integer> inorderTraversal(TreeNode root) {

        if(root==null) return list;

        inorderTraversal(root.left);

        list.add(root.val);

        inorderTraversal(root.right);

        return list;

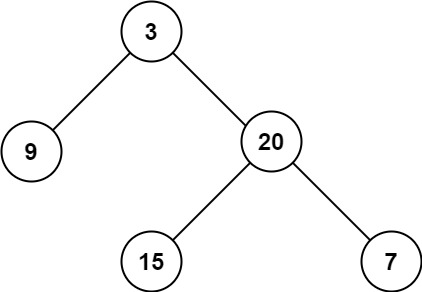
    }

}

**6 ) Maximum Depth of Binary Tree (Leetcode – 104)**

Given the root of a binary tree, return its maximum depth.A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: 3

Example 2:

Input: root = [1,null,2]

Output: 2

Constraints:

The number of nodes in the tree is in the range [0, 104].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    public int maxDepth(TreeNode root) {

        if(root==null) return 0;

        int left=maxDepth(root.left);

        int right=maxDepth(root.right);

        return Math.max(left,right)+1;

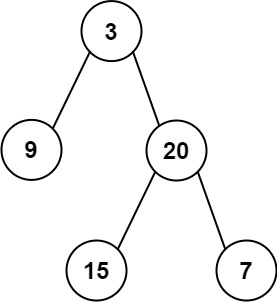
    }

}

**7) Average of Levels in Binary Tree (Leetcode - 637)**

Given the root of a binary tree, return the average value of the nodes on each level in the form of an array. Answers within 10-5 of the actual answer will be accepted.

Example 1:



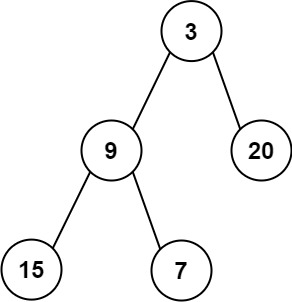
Input: root = [3,9,20,null,null,15,7]

Output: [3.00000,14.50000,11.00000]

Explanation: The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11.

Hence return [3, 14.5, 11].

Example 2:



Input: root = [3,9,20,15,7]

Output: [3.00000,14.50000,11.00000]

Constraints:

The number of nodes in the tree is in the range [1, 104].

-231 <= Node.val <= 231 – 1

**Solution :**

class Solution {

    public List<Double> averageOfLevels(TreeNode root) {

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

        if(root==null)

        {

            return res;

        }

        Queue<TreeNode> queue=new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty())

        {

            int levelSize=queue.size();

            double sum=0;

            for(int i=0;i<levelSize;i++)

            {

                TreeNode node=queue.poll();

                sum+=node.val;

                if(node.left!=null)

                {

                    queue.offer(node.left);

                }

                if(node.right!=null)

                {

                    queue.offer(node.right);

                }

            }

            res.add(sum/levelSize);

        }

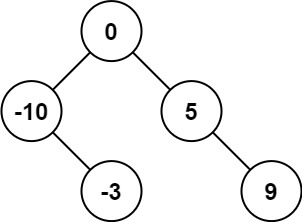
        return res;

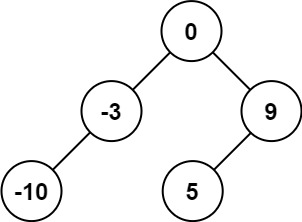
    }

}

**8) Convert Sorted Array to Binary Search Tree (Leetcode - 108)**

Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Example 1:

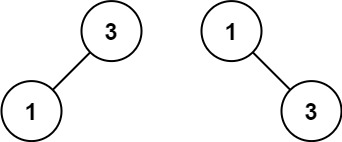


Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:

Example 2:



Input: nums = [1,3]

Output: [3,1]

Explanation: [1,null,3] and [3,1] are both height-balanced BSTs.

Constraints:

1 <= nums.length <= 104

-104 <= nums[i] <= 104

nums is sorted in a strictly increasing order.

**Solution :**

class Solution {

    public TreeNode sortedArrayToBST(int[] nums) {

        if(nums.length==0) return null;

        int mid=nums.length/2;

        TreeNode root = new TreeNode(nums[mid]);

        int left[]=new int[mid];

        /\* int i=0;

        for(i=0;i<mid;i++)

        {

            left[i]=nums[i];

        }

        int[] right = new int[nums.length-mid-1];

        i++;

        for(int j=0;j<nums.length-mid-1;j++)

        {

            right[j]=nums[i];

            i++;

        } \*/

        root.left=sortedArrayToBST(Arrays.copyOfRange(nums,0,mid));

        root.right=sortedArrayToBST(Arrays.copyOfRange(nums,mid+1,nums.length));

        return root;

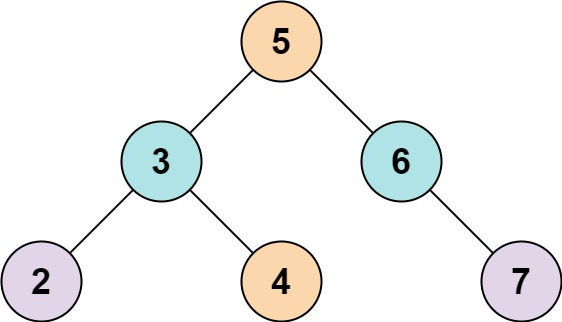
    }

}

**9) Two Sum IV - Input is a BST (Leetcode - 653)**

Given the root of a binary search tree and an integer k, return true if there exist two elements in the BST such that their sum is equal to k, or false otherwise.

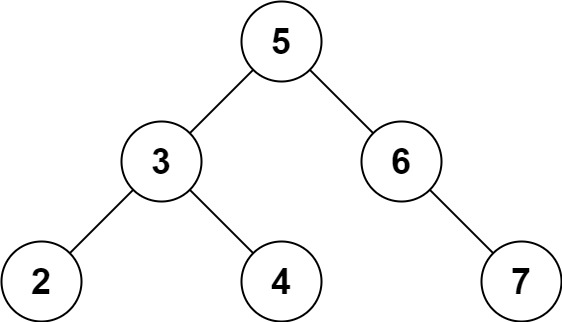
Example 1:



Input: root = [5,3,6,2,4,null,7], k = 9

Output: true

Example 2:



Input: root = [5,3,6,2,4,null,7], k = 28

Output: false

Constraints:

The number of nodes in the tree is in the range [1, 104].

-104 <= Node.val <= 104

root is guaranteed to be a valid binary search tree.

-105 <= k <= 105

**Solution :**

class Solution {

    Set<Integer> set = new HashSet<>();

    public boolean findTarget(TreeNode root, int k) {

        if(root==null)return false;

        if(set.contains(k-root.val))

        {

            return true;

        }

        set.add(root.val);

        return findTarget(root.left,k) || findTarget(root.right,k);

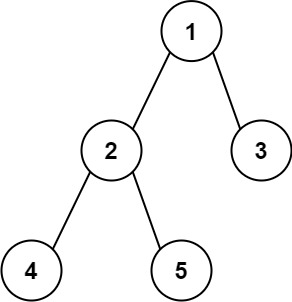
    }

}

**10) Diameter of Binary Tree (Leetcode - 543)**

Given the root of a binary tree, return the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.The length of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: root = [1,2,3,4,5]

Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

Example 2:

Input: root = [1,2]

Output: 1

Constraints:

The number of nodes in the tree is in the range [1, 104].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    int diameter=0;

    public int diameterOfBinaryTree(TreeNode root) {

        height(root);

        return diameter;

    }

    public int height(TreeNode node)

    {

        if(node==null) return 0;

        int leftHeight=height(node.left);

        int rightHeight=height(node.right);

        int dia=leftHeight+rightHeight;

        diameter=Math.max(dia,diameter);

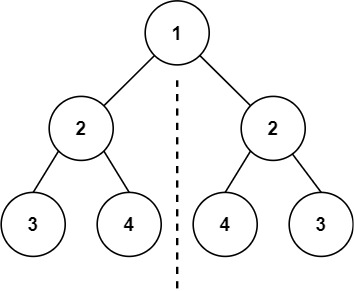
        return Math.max(leftHeight,rightHeight)+1;

    }

}

**11) Symmetric Tree (Leetcode - 101)**

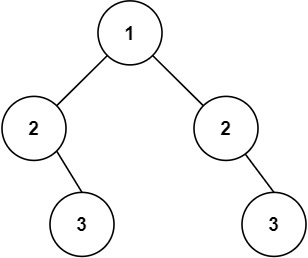
Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Example 1:

Input: root = [1,2,2,3,4,4,3]

Output: true

Example 2:



Input: root = [1,2,2,null,3,null,3]

Output: false

Constraints:

The number of nodes in the tree is in the range [1, 1000].

-100 <= Node.val <= 100

**Solution 1:**

class Solution {

    public boolean isSymmetric(TreeNode root) {

        return isSymmetric(root.left, root.right);

    }

    private boolean isSymmetric(TreeNode t1, TreeNode t2) {

        if (t1 == null && t2 == null) {

            return true;

        }

        if (t1 == null || t2 == null) {

            return false;

        }

        return (t1.val == t2.val) && isSymmetric(t1.left, t2.right) && isSymmetric(t1.right, t2.left);

    }

}

**Solution 2:**

class Solution {

    public boolean isSymmetric(TreeNode root) {

        Queue<TreeNode> queue=new LinkedList<>();

        queue.add(root.left);

        queue.add(root.right);

        while(!queue.isEmpty())

        {

            TreeNode left = queue.poll();

            TreeNode right = queue.poll();

            if(left==null && right==null) continue;

            if(left==null || right==null) return false;

            if(left.val!=right.val) return false;

            queue.add(left.left);

            queue.add(right.right);

            queue.add(left.right);

            queue.add(right.left);

        }

        return true;

    }

}

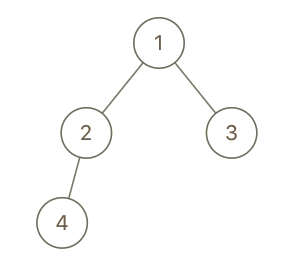
**12) Cousins in Binary Tree (Leetcode - 993)**

Given the root of a binary tree with unique values and the values of two different nodes of the tree x and y, return true if the nodes corresponding to the values x and y in the tree are cousins, or false otherwise.

Two nodes of a binary tree are cousins if they have the same depth with different parents.

Note that in a binary tree, the root node is at the depth 0, and children of each depth k node are at the depth k + 1.

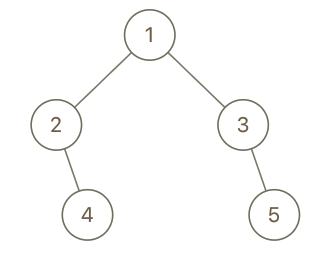
Example 1:



Input: root = [1,2,3,4], x = 4, y = 3

Output: false

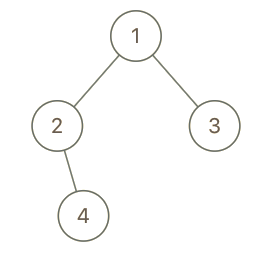
Example 2:



Input: root = [1,2,3,null,4,null,5], x = 5, y = 4

Output: true

Example 3:



Input: root = [1,2,3,null,4], x = 2, y = 3

Output: false

Constraints:

The number of nodes in the tree is in the range [2, 100].

1 <= Node.val <= 100

Each node has a unique value.

x != y

x and y are exist in the tree.

**Solution :**

class Solution {

    public boolean isCousins(TreeNode root, int x, int y) {

       TreeNode xx=findNode(root,x);

       TreeNode yy=findNode(root,y);

       return (

        (level(root,x,0)==level(root,y,0)) && (!isSibling(root,x,y))

        );

    }

    public static TreeNode findNode(TreeNode node,int v)

    {

        if(node==null) return null;

        if(node.val==v) return node;

        TreeNode n=findNode(node.left,v);

        if(n!=null) return n;

        return findNode(node.right,v);

    }

    public static boolean isSibling(TreeNode node,int x,int y)

    {

        if(node==null) return false;

        return (

            (node.left.val==x && node.right.val==y) || (node.left.val==y && node.right.val==x) || isSibling(node.left,x,y)

             || isSibling(node.right,x,y));

    }

    public static int level(TreeNode node,int x,int level)

    {

        if(node==null) return 0;

        if(node.val==x) return level;

        int lev=level(node.left,x,level+1);

        if(lev!=0) return lev;

        return level(node.right,x,level+1);

    }

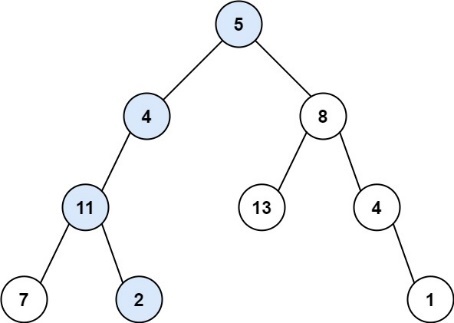
}

**13) Path Sum (Leetcode - 112)**

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

Example 1:

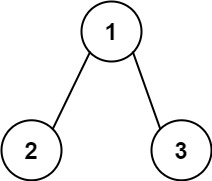


Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

Output: true

Explanation: The root-to-leaf path with the target sum is shown.

Example 2:



Input: root = [1,2,3], targetSum = 5

Output: false

Explanation: There two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

Example 3:

Input: root = [], targetSum = 0

Output: false

Explanation: Since the tree is empty, there are no root-to-leaf paths.

Constraints:

The number of nodes in the tree is in the range [0, 5000].

-1000 <= Node.val <= 1000

-1000 <= targetSum <= 1000

**Solution :**

class Solution {

    public boolean hasPathSum(TreeNode root, int targetSum) {

        if(root==null) return false;

        if(root.val==targetSum && root.left==null && root.right==null) return true;

        return hasPathSum(root.left,targetSum-root.val) || hasPathSum(root.right,targetSum-root.val);

    }

}

**14) Full Binary Tree**

A full binary tree, on the other hand, does not have any nodes that have only one child node.

**Solution:**

boolean isFullTree(Node node)

{

// if empty tree

if(node == null)

return true;

// if leaf node

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

return true;

// if both left and right subtrees are not null

// they are full

if((node.left!=null) && (node.right!=null))

return (isFullTree(node.left) && isFullTree(node.right));

// if none work(one child)

return false;

}

**15) Complete BinaryTree**

In a Complete Binary Tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

**Solution :**

public static String isComplete(Node root)

{

Queue<Node> queue = new LinkedList<>();

queue.add(root);

boolean flag=false;

while(!queue.isEmpty())

{

Node current = queue.poll();

if(current==null)

{

flag=true;

continue;

}

if(flag==true)

{

return "No";

}

queue.add(current.left);

queue.add(current.right);

}

return "Yes";

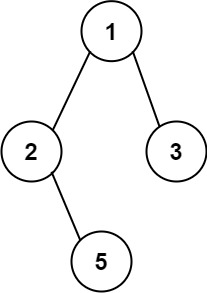
}

**16) Binary Tree Paths (Leetcode - 257)**

Given the root of a binary tree, return all root-to-leaf paths in any order.

A leaf is a node with no children.

Example 1:



Input: root = [1,2,3,null,5]

Output: ["1->2->5","1->3"]

Example 2:

Input: root = [1]

Output: ["1"]

Constraints:

The number of nodes in the tree is in the range [1, 100].

-100 <= Node.val <= 100

**Solution :**

class Solution {

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

    public void path(TreeNode root,String s)

    {

        if(root==null) return ;

        if(root.left==null && root.right==null)

        {

            res.add(s+root.val);

            return ;

        }

        path(root.left,s+Integer.toString(root.val)+"->");

        path(root.right,s+Integer.toString(root.val)+"->");

    }

    public List<String> binaryTreePaths(TreeNode root) {

        path(root,"");

        return res;

    }

}

**17) Merge Two Binary Trees (Leetcode - 617)**

You are given two binary trees root1 and root2.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return the merged tree.

Note: The merging process must start from the root nodes of both trees.

Example 1:



Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

Output: [3,4,5,5,4,null,7]

Example 2:

Input: root1 = [1], root2 = [1,2]

Output: [2,2]

Constraints:

The number of nodes in both trees is in the range [0, 2000].

-104 <= Node.val <= 104

**Solution :**

class Solution {

    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {

        if(root1==null && root2 == null)

        {

            return null;

        }

        int val1 = (root1!=null)?root1.val:0;

        int val2 = (root2!=null)?root2.val:0;

        TreeNode root = new TreeNode(val1+val2);

        TreeNode left = (root1!=null)?root1.left:null;

        TreeNode right = (root2!=null)?root2.left:null;

        root.left=mergeTrees(left,right);

        left = (root1!=null)?root1.right:null;

        right = (root2!=null)?root2.right:null;

        root.right=mergeTrees(left,right);

        return root;

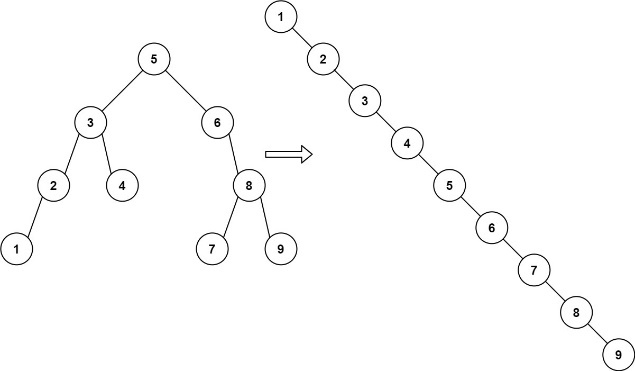
    }

}

**18) Increasing Order Search Tree (Leetcode - 897)**

Given the root of a binary search tree, rearrange the tree in in-order so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only one right child.

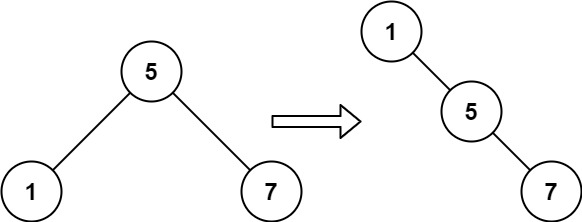
Example 1:



Input: root = [5,3,6,2,4,null,8,1,null,null,null,7,9]

Output: [1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]

Example 2:



Input: root = [5,1,7]

Output: [1,null,5,null,7]

Constraints:

The number of nodes in the given tree will be in the range [1, 100].

0 <= Node.val <= 1000

**Solution :**

class Solution {

    TreeNode newRoot ;

    public void inorder(TreeNode root)

    {

        if(root==null)

        {

            return ;

        }

        inorder(root.left);

        newRoot.right=new TreeNode(root.val);

        newRoot = newRoot.right;

        inorder(root.right);

    }

    public TreeNode increasingBST(TreeNode root) {

        TreeNode temp = new TreeNode(-1);

        newRoot = temp;

        inorder(root);

        return temp.right;

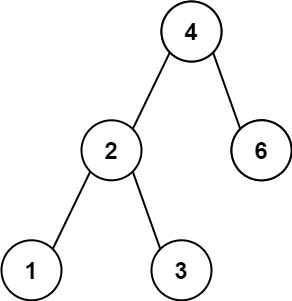
    }

}

**19) Minimum Absolute Difference in BST (Leetcode – 530)**

Given the root of a Binary Search Tree (BST), return the minimum absolute difference between the values of any two different nodes in the tree.

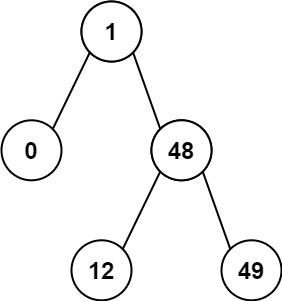
Example 1:



Input: root = [4,2,6,1,3]

Output: 1

Example 2:



Input: root = [1,0,48,null,null,12,49]

Output: 1

Constraints:

The number of nodes in the tree is in the range [2, 104].

0 <= Node.val <= 105

**Solution :**

class Solution {

    int min=Integer.MAX\_VALUE;

    int prev=Integer.MAX\_VALUE;

    public void inorder(TreeNode root)

    {

        if(root==null)

        {

            return ;

        }

        inorder(root.left);

        min=Math.min(min,Math.abs(root.val-prev));

        prev=root.val;

        inorder(root.right);

    }

    public int getMinimumDifference(TreeNode root) {

        inorder(root);

        return min;

    }

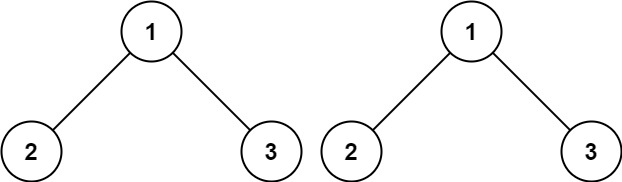
}

**20) Same Tree (Leetcode - 100)**

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.

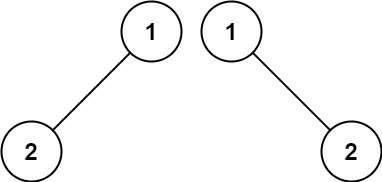
Example 1:



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

Output: true

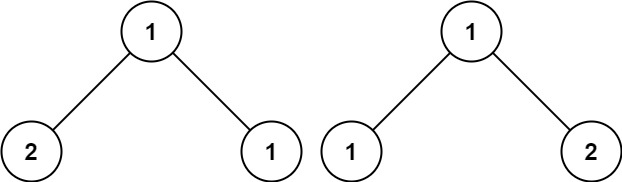
Example 2:



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

**Solution:**

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);

    }

}

**21)Top view BinaryTree**

Input:     1  
           /     \  
         2       3  
       /  \    / \  
     4    5  6   7

Output: Top view of the above binary tree is: 4 2 1 3 7

**Solution :**

 static class QueueObj {

        Node node;

        int hd;

        QueueObj(Node node, int hd)

        {

            this.node = node;

            this.hd = hd;

        }

    }

    static void topView(Node root)

    {

        if (root == null)

            return;

        Queue<QueueObj> q = new LinkedList<>();

        Map<Integer, Integer> map = new HashMap<>();

        int min = 0;

        int max = 0;

        // Level Order Traversal

        q.add(new QueueObj(root, 0));

        while (!q.isEmpty()) {

            QueueObj curr = q.poll();

            // only include in map if this is the

            // first node of this specific

            // horizontal distance

            if (!map.containsKey(curr.hd)) {

                map.put(curr.hd, curr.node.data);

            }

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

                // min can be found only in left side due to

                // "-1" minimum horizontal distance of any

                // node from root

                min = Math.min(min, curr.hd - 1);

                q.add(new QueueObj(curr.node.left,

                                   curr.hd - 1));

            }

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

                // max can be found only in right side due

                // to "+1" maximum horizontal distance of

                // any node from root

                max = Math.max(max, curr.hd + 1);

                q.add(new QueueObj(curr.node.right,

                                   curr.hd + 1));

            }

        }

        // traversal of (horizontal distance from root)

        // minimum to maximum

        for (; min <= max; min++) {

            System.out.print(map.get(min) + " ");

        }

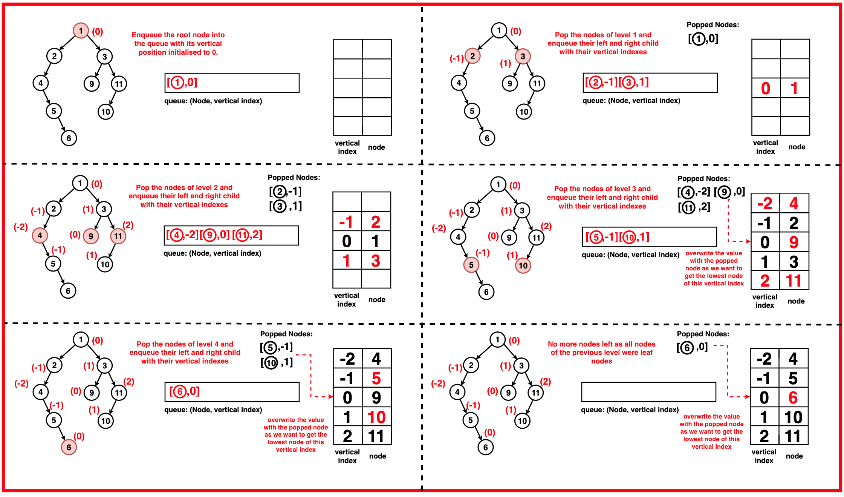
    }

**22 ) Bottom view :**

***Input:****20  
                    /     \  
                8         22  
             /     \     /     \  
          5        3  4      25  
                  /    \        
              10       14****Output:****5 10 4 14 25.*

**Solution :**

**Same as top view but in bottom view we consider last element in every HD.**

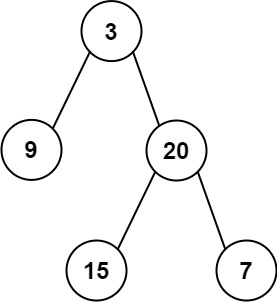


**23) Sum of Left Leaves (Leetcode - 404)**

Given the root of a binary tree, return the sum of all left leaves.

A leaf is a node with no children. A left leaf is a leaf that is the left child of another node.

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: 24

Explanation: There are two left leaves in the binary tree, with values 9 and 15 respectively.

Example 2:

Input: root = [1]

Output: 0

Constraints:

The number of nodes in the tree is in the range [1, 1000].

-1000 <= Node.val <= 1000

**Solution :**

class Solution {

    public int sumOfLeftLeaves(TreeNode root) {

        if(root==null) return 0;

        int sum=0;

        if(root.left!=null && root.left.left==null && root.left.right==null)

        {

           sum+=root.left.val;

        }

        else

        {

            sum+=sumOfLeftLeaves(root.left);

        }

        return sum+sumOfLeftLeaves(root.right);

    }

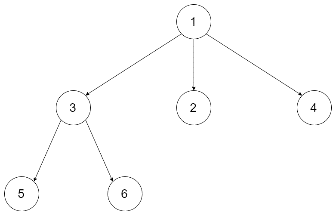
}

**24) N-ary Tree Postorder Traversal (Leetcode - 590)**

Given the root of an n-ary tree, return the postorder traversal of its nodes' values.

Nary-Tree input serialization is represented in their level order traversal. Each group of children is separated by the null value (See examples)

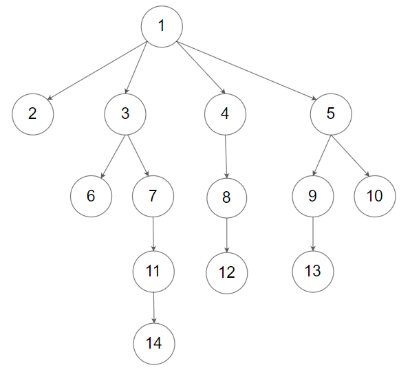
Example 1:



Input: root = [1,null,3,2,4,null,5,6]

Output: [5,6,3,2,4,1]

Example 2:



Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

Output: [2,6,14,11,7,3,12,8,4,13,9,10,5,1]

Constraints:

The number of nodes in the tree is in the range [0, 104].

0 <= Node.val <= 104

The height of the n-ary tree is less than or equal to 1000.

**Solution :**

class Solution {

    public void postOrder(Node root, List<Integer> res)

    {

        if(root==null) return ;

        for(Node child:root.children)

        {

            postOrder(child,res);

        }

        res.add(root.val);

    }

    public List<Integer> postorder(Node root){

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

        postOrder(root,res);

        return res;

    }

}

**1) Binary Search Tree to Greater Sum Tree (Leetcode - 1038) (Medium)**

Given the root of a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus the sum of all keys greater than the original key in BST.

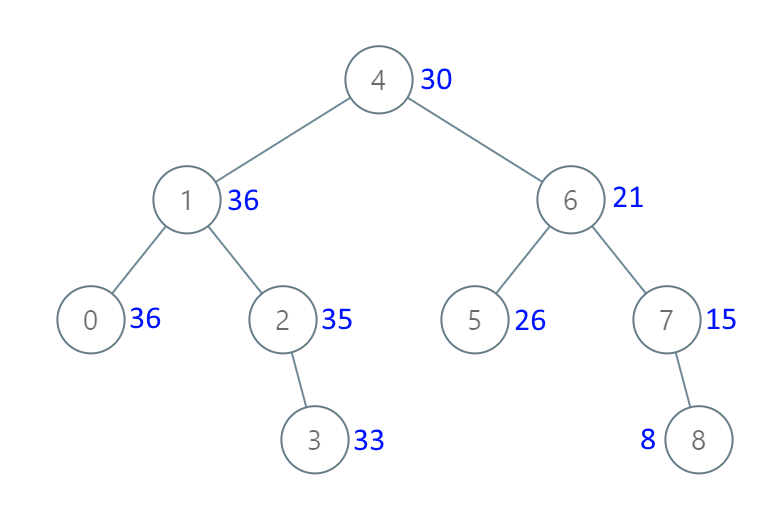
As a reminder, a binary search tree is a tree that satisfies these constraints:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Example 1:



Input: root = [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]

Output: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]

Example 2:

Input: root = [0,null,1]

Output: [1,null,1]

Constraints:

The number of nodes in the tree is in the range [1, 100].

0 <= Node.val <= 100

All the values in the tree are unique.

**Solution :**

class Solution {

    int sum=0;

    public TreeNode bstToGst(TreeNode root) {

        if(root==null) return null;

        bstToGst(root.right);

        sum+=root.val;

        root.val=sum;

        bstToGst(root.left);

        return root;

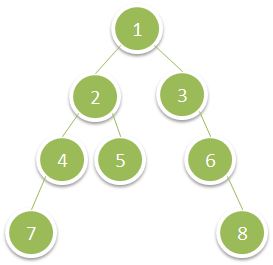
    }

}

**2) Deepest Leaves Sum (Leetcode – 1302)**

Given the root of a binary tree, return the sum of values of its deepest leaves.

Example 1:



Input: root = [1,2,3,4,5,null,6,7,null,null,null,null,8]

Output: 15

Example 2:

Input: root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]

Output: 19

Constraints:

The number of nodes in the tree is in the range [1, 104].

1 <= Node.val <= 100

**Solution :**

class Solution {

    public int deepestLeavesSum(TreeNode root) {

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        int sum=0;

        while(!queue.isEmpty())

        {

            int size=queue.size();

            sum=0;

            for(int i=0;i<size;i++)

            {

                TreeNode cur=queue.poll();

                sum+=cur.val;

                if(cur.left!=null)

                {

                    queue.offer(cur.left);

                }

                if(cur.right!=null)

                {

                    queue.offer(cur.right);

                }

            }

        }

        return sum;

    }

}

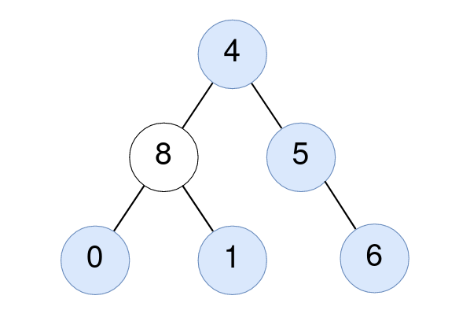
**3) Count Nodes Equal to Average of Subtree (Leetcode-2265)**

Given the root of a binary tree, return the number of nodes where the value of the node is equal to the average of the values in its subtree.

Note: The average of n elements is the sum of the n elements divided by n and rounded down to the nearest integer.

A subtree of root is a tree consisting of root and all of its descendants.

Example 1:



Input: root = [4,8,5,0,1,null,6]

Output: 5

Explanation:

For the node with value 4: The average of its subtree is (4 + 8 + 5 + 0 + 1 + 6) / 6 = 24 / 6 = 4.

For the node with value 5: The average of its subtree is (5 + 6) / 2 = 11 / 2 = 5.

For the node with value 0: The average of its subtree is 0 / 1 = 0.

For the node with value 1: The average of its subtree is 1 / 1 = 1.

For the node with value 6: The average of its subtree is 6 / 1 = 6.

Example 2:



Input: root = [1]

Output: 1

Explanation: For the node with value 1: The average of its subtree is 1 / 1 = 1.

**Solution :**

class Solution {

    int ans=0;

    public int[] help(TreeNode root){

        if(root==null){

            return new int[]{0,0};

        }

        int l[]=help(root.left);

        int r[]=help(root.right);

        int s=l[0]+r[0]+root.val;

        int c=1+l[1]+r[1];

        if(root.val==s/c)

        ans+=1;

        return new int[]{s,c};

    }

    public int averageOfSubtree(TreeNode root) {

        if(root==null)

        return 0;

        int t[]=help(root);

        return ans;

    }

}

**4) Maximum Binary Tree (Leetcode - 654)**

You are given an integer array nums with no duplicates. A maximum binary tree can be built recursively from nums using the following algorithm:

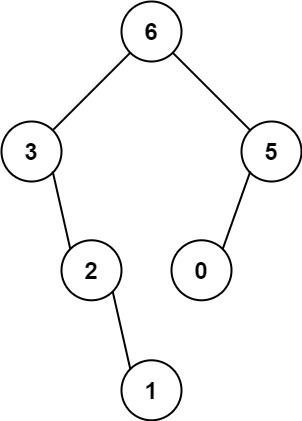
Create a root node whose value is the maximum value in nums.

Recursively build the left subtree on the subarray prefix to the left of the maximum value.

Recursively build the right subtree on the subarray suffix to the right of the maximum value.

Return the maximum binary tree built from nums.

Example 1:



Input: nums = [3,2,1,6,0,5]

Output: [6,3,5,null,2,0,null,null,1]

Explanation: The recursive calls are as follow:

- The largest value in [3,2,1,6,0,5] is 6. Left prefix is [3,2,1] and right suffix is [0,5].

- The largest value in [3,2,1] is 3. Left prefix is [] and right suffix is [2,1].

- Empty array, so no child.

- The largest value in [2,1] is 2. Left prefix is [] and right suffix is [1].

- Empty array, so no child.

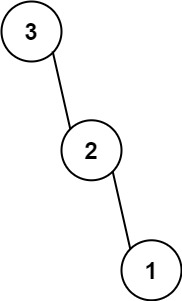
- Only one element, so child is a node with value 1.

- The largest value in [0,5] is 5. Left prefix is [0] and right suffix is [].

- Only one element, so child is a node with value 0.

- Empty array, so no child.

Example 2:



Input: nums = [3,2,1]

Output: [3,null,2,null,1]

Constraints:

1 <= nums.length <= 1000

0 <= nums[i] <= 1000

All integers in nums are unique.

**Solution 1:**

class Solution {

    public TreeNode max(int[] nums,int start,int end)

    {

        if(start>end) return null;

        int maxId=-1;

        int maxEle=Integer.MIN\_VALUE;

        for(int i=start;i<=end;i++)

        {

            if(nums[i]>maxEle)

            {

                maxId=i;

                maxEle=nums[i];

            }

        }

        TreeNode root = new TreeNode(nums[maxId]);

        root.left=max(nums,start,maxId-1);

        root.right=max(nums,maxId+1,end);

        return root;

    }

    public TreeNode constructMaximumBinaryTree(int[] nums) {

        return max(nums,0,nums.length-1);

    }

}

**Solution 2 : (Using Stack)**

class Solution {

    public TreeNode constructMaximumBinaryTree(int[] nums) {

        Deque<TreeNode> stack = new LinkedList<>();

        for(int i:nums)

        {

            TreeNode cur = new TreeNode(i);

            while(!stack.isEmpty() && stack.peek().val<i)

            {

                cur.left=stack.pop();

            }

            if(!stack.isEmpty())

            {

                stack.peek().right=cur;

            }

            stack.push(cur);

        }

        return stack.isEmpty()?null:stack.removeLast();

    }

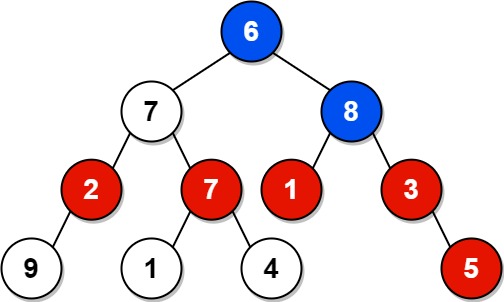
}

**5) Sum of Nodes with Even-Valued Grandparent (Leetcode - 1315)**

Given the root of a binary tree, return the sum of values of nodes with an even-valued grandparent. If there are no nodes with an even-valued grandparent, return 0.

A grandparent of a node is the parent of its parent if it exists.

Example 1:



Input: root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]

Output: 18

Explanation: The red nodes are the nodes with even-value grandparent while the blue nodes are the even-value grandparents.

Example 2:

Input: root = [1]

Output: 0

Constraints:

The number of nodes in the tree is in the range [1, 104].

1 <= Node.val <= 100

**Solution :**

class Solution {

    public int sum(TreeNode root,TreeNode parent,TreeNode grand)

    {

        if(root==null)

        {

            return 0;

        }

        int res=0;

        if( grand.val%2==0)

        {

            res=root.val;

        }

        return res+sum(root.left,root,parent)+sum(root.right,root,parent);

    }

    public int sumEvenGrandparent(TreeNode root) {

        return sum(root,new TreeNode(1),new TreeNode(1));

    }

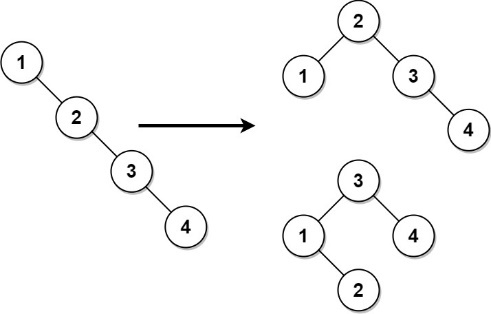
}

**6) Balance a Binary Search Tree (Leetcode - 1382)**

Given the root of a binary search tree, return a balanced binary search tree with the same node values. If there is more than one answer, return any of them.

A binary search tree is balanced if the depth of the two subtrees of every node never differs by more than 1.

Example 1:

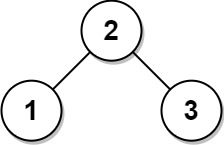


Input: root = [1,null,2,null,3,null,4,null,null]

Output: [2,1,3,null,null,null,4]

Explanation: This is not the only correct answer, [3,1,4,null,2] is also correct.

Example 2:



Input: root = [2,1,3]

Output: [2,1,3]

Constraints:

The number of nodes in the tree is in the range [1, 104].

1 <= Node.val <= 105

**Solution :**

class Solution {

    public void inorder(TreeNode root,ArrayList<Integer> list)

    {

        if(root==null)

        {

            return ;

        }

        inorder(root.left,list);

        list.add(root.val);

        inorder(root.right,list);

    }

    public TreeNode buildTree(int start,int end,ArrayList<Integer> list)

    {

        if(start>end)

        {

            return null;

        }

        int mid=start+(end-start)/2;

        TreeNode root = new TreeNode(list.get(mid));

        root.left=buildTree(start,mid-1,list);

        root.right=buildTree(mid+1,end,list);

        return root;

    }

    public TreeNode balanceBST(TreeNode root) {

        ArrayList list = new ArrayList<>();

        inorder(root,list);

        return buildTree(0,list.size()-1,list);

    }

}

**7) Construct Binary Search Tree from Preorder Traversal (Leetcode - 1008)**

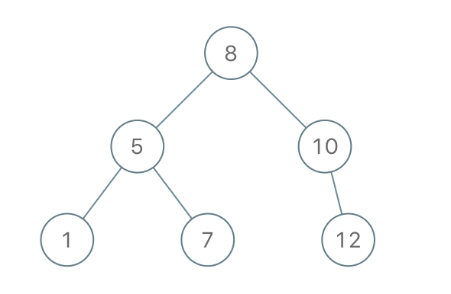
Given an array of integers preorder, which represents the preorder traversal of a BST (i.e., binary search tree), construct the tree and return its root.

It is guaranteed that there is always possible to find a binary search tree with the given requirements for the given test cases.

A binary search tree is a binary tree where for every node, any descendant of Node.left has a value strictly less than Node.val, and any descendant of Node.right has a value strictly greater than Node.val.

A preorder traversal of a binary tree displays the value of the node first, then traverses Node.left, then traverses Node.right.

Example 1:



Input: preorder = [8,5,1,7,10,12]

Output: [8,5,10,1,7,null,12]

Example 2:

Input: preorder = [1,3]

Output: [1,null,3]

Constraints:

1 <= preorder.length <= 100

1 <= preorder[i] <= 1000

All the values of preorder are unique.

**Solution :**

class Solution {

    int i=0;

    public TreeNode construct(int[] preorder,int bound)

    {

        if(i==preorder.length || preorder[i] > bound)

        {

            return null;

        }

        TreeNode root = new TreeNode(preorder[i++]);

        root.left=construct(preorder,root.val);

        root.right=construct(preorder,bound);

        return root;

    }

    public TreeNode bstFromPreorder(int[] preorder) {

        return construct(preorder,Integer.MAX\_VALUE);

    }

}

**8) Reverse Odd Levels of Binary Tree (Leetcode - 2415)**

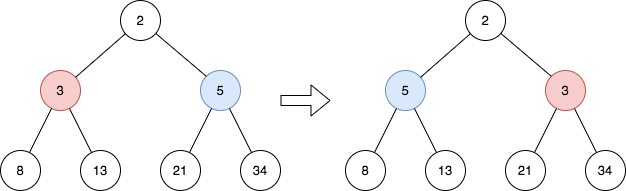
Given the root of a perfect binary tree, reverse the node values at each odd level of the tree.

For example, suppose the node values at level 3 are [2,1,3,4,7,11,29,18], then it should become [18,29,11,7,4,3,1,2].

Return the root of the reversed tree.

A binary tree is perfect if all parent nodes have two children and all leaves are on the same level.The level of a node is the number of edges along the path between it and the root node.

Example 1:



Input: root = [2,3,5,8,13,21,34]

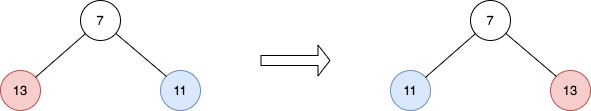
Output: [2,5,3,8,13,21,34]

Explanation:

The tree has only one odd level.

The nodes at level 1 are 3, 5 respectively, which are reversed and become 5, 3.

Example 2:



Input: root = [7,13,11]

Output: [7,11,13]

Explanation:

The nodes at level 1 are 13, 11, which are reversed and become 11, 13.

Example 3:

Input: root = [0,1,2,0,0,0,0,1,1,1,1,2,2,2,2]

Output: [0,2,1,0,0,0,0,2,2,2,2,1,1,1,1]

Explanation:

The odd levels have non-zero values.

The nodes at level 1 were 1, 2, and are 2, 1 after the reversal.

The nodes at level 3 were 1, 1, 1, 1, 2, 2, 2, 2, and are 2, 2, 2, 2, 1, 1, 1, 1 after the reversal.

Constraints:

The number of nodes in the tree is in the range [1, 214].

0 <= Node.val <= 105

root is a perfect binary tree.

**Solution :**

class Solution {

    public TreeNode reverseOddLevels(TreeNode root)

     {

        traverse(root.left,root.right,1);

        return root;

     }

    public void traverse(TreeNode node1,TreeNode node2,int level)

    {

        if(node1==null)

        {

            return ;

        }

        if(level%2==1)

        {

            int temp=node1.val;

            node1.val=node2.val;

            node2.val=temp;

        }

        traverse(node1.left,node2.right,level+1);

        traverse(node1.right,node2.left,level+1);

    }

}

**9) Delete Leaves With a Given Value (Leetcode - 1325)**

Given a binary tree root and an integer target, delete all the leaf nodes with value target.

Note that once you delete a leaf node with value target, if its parent node becomes a leaf node and has the value target, it should also be deleted (you need to continue doing that until you cannot).

Example 1:



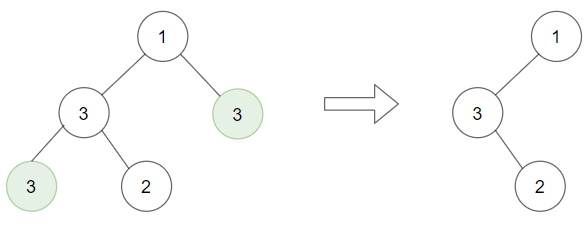
Input: root = [1,2,3,2,null,2,4], target = 2

Output: [1,null,3,null,4]

Explanation: Leaf nodes in green with value (target = 2) are removed (Picture in left).

After removing, new nodes become leaf nodes with value (target = 2) (Picture in center).

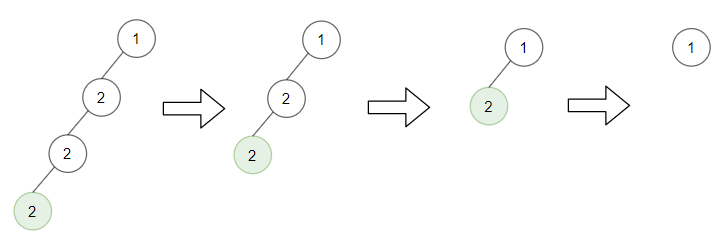
Example 2:



Input: root = [1,3,3,3,2], target = 3

Output: [1,3,null,null,2]

Example 3:



Input: root = [1,2,null,2,null,2], target = 2

Output: [1]

Explanation: Leaf nodes in green with value (target = 2) are removed at each step.

Constraints:

The number of nodes in the tree is in the range [1, 3000].

1 <= Node.val, target <= 1000

**Solution :**

class Solution {

    public TreeNode removeLeafNodes(TreeNode root, int target) {

        if (root == null) {

            return null;

        }

        root.left = removeLeafNodes(root.left, target);

        root.right = removeLeafNodes(root.right, target);

        if (root.val == target && root.left == null && root.right == null) {

            return null;

        }

        return root;

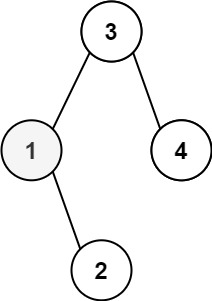
    }

}

**10) Kth Smallest Element in a BST (Leetcode - 230)**

Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

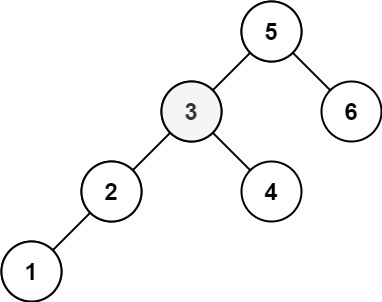
Example 1:



Input: root = [3,1,4,null,2], k = 1

Output: 1

Example 2:



Input: root = [5,3,6,2,4,null,null,1], k = 3

Output: 3

Constraints:

The number of nodes in the tree is n.

1 <= k <= n <= 104

0 <= Node.val <= 104

**Solution :**

class Solution {

    int count=0;

    public int kthSmallest(TreeNode root, int k) {

        if(root==null) return 0;

        int left=kthSmallest(root.left,k);

        count+=1;

        if(count==k)

        {

            return root.val;

        }

        if(left!=0) return left;

        return kthSmallest(root.right,k);

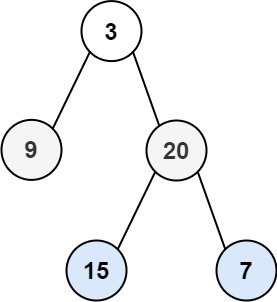
        }

}

**11) Binary Tree Level Order Traversal (Leetcode - 102)**

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

Example 2:

Input: root = [1]

Output: [[1]]

Example 3:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 2000].

-1000 <= Node.val <= 1000

**Solution :**

class Solution {

    public List<List<Integer>> levelOrder(TreeNode root) {

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

        if(root==null)

        {

            return res;

        }

        Queue<TreeNode> queue=new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty())

        {

            int levelSize=queue.size();

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

            for(int i=0;i<levelSize;i++)

            {

                TreeNode node=queue.poll();

                l.add(node.val);

                if(node.left!=null)

                {

                    queue.offer(node.left);

                }

                if(node.right!=null)

                {

                    queue.offer(node.right);

                }

            }

            res.add(l);

        }

        return res;

    }

}

**12) Sum Root to Leaf Numbers (Leetcode - 129)**

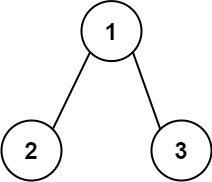
You are given the root of a binary tree containing digits from 0 to 9 only. Each root-to-leaf path in the tree represents a number.

For example, the root-to-leaf path 1 -> 2 -> 3 represents the number 123.

Return the total sum of all root-to-leaf numbers. Test cases are generated so that the answer will fit in a 32-bit integer.

A leaf node is a node with no children.

Example 1:



Input: root = [1,2,3]

Output: 25

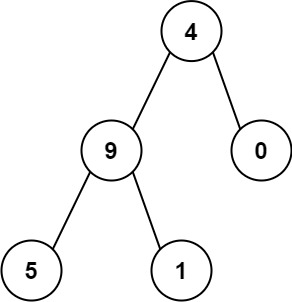
Explanation:

The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Therefore, sum = 12 + 13 = 25.

Example 2:



Input: root = [4,9,0,5,1]

Output: 1026

Explanation:

The root-to-leaf path 4->9->5 represents the number 495.

The root-to-leaf path 4->9->1 represents the number 491.

The root-to-leaf path 4->0 represents the number 40.

Therefore, sum = 495 + 491 + 40 = 1026.

Constraints:

The number of nodes in the tree is in the range [1, 1000].

0 <= Node.val <= 9

The depth of the tree will not exceed 10.

**Solution :**

class Solution {

    public int sumNumbers(TreeNode root) {

        return numbers(root,0);

    }

    public int numbers(TreeNode root,int sum)

    {

        if(root==null)

        {

            return 0;

        }

        sum=sum\*10+root.val;

        if(root.left==null && root.right==null)

        {

            return sum;

        }

        return numbers(root.left,sum)+numbers(root.right,sum);

    }

}

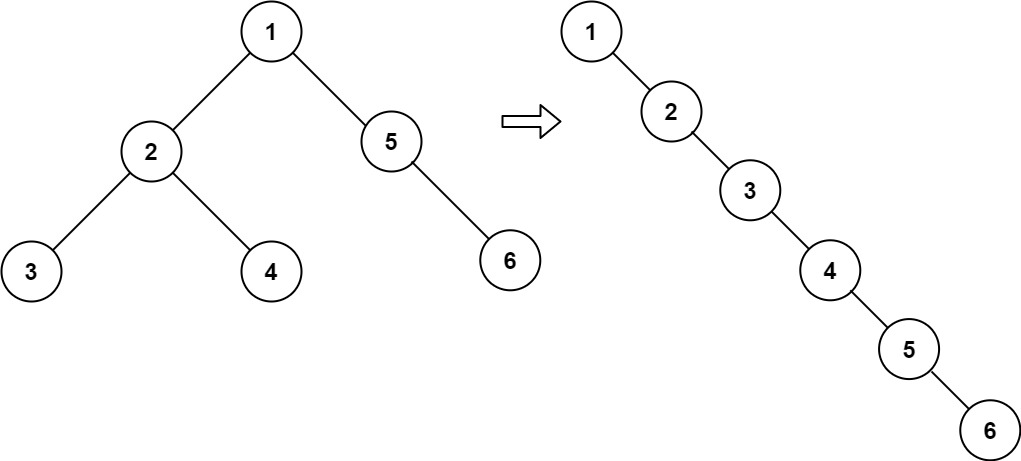
**13) Flatten Binary Tree to Linked List (Leetcode - 114 )**

Given the root of a binary tree, flatten the tree into a "linked list":

The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.

The "linked list" should be in the same order as a pre-order traversal of the binary tree.

Example 1:



Input: root = [1,2,5,3,4,null,6]

Output: [1,null,2,null,3,null,4,null,5,null,6]

Example 2:

Input: root = []

Output: []

Example 3:

Input: root = [0]

Output: [0]

Constraints:

The number of nodes in the tree is in the range [0, 2000].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    public void flatten(TreeNode root) {

        if(root==null)

        {

          return ;

        }

        TreeNode temp = root;

        while(temp!=null)

        {

            if(temp.left!=null)

            {

                TreeNode current = temp.left;

                while(current.right!=null)

                {

                    current=current.right;

                }

                current.right=temp.right;

                temp.right=temp.left;

                temp.left=null;

            }

            temp=temp.right;

        }

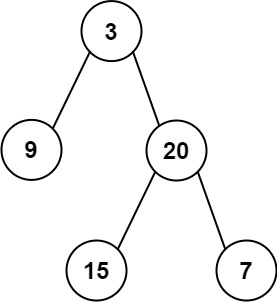
    }

}

**14) Construct Binary Tree from Preorder and Inorder Traversal (Leetcode - 105)**

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Example 1:



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

Output: [3,9,20,null,null,15,7]

Example 2:

Input: preorder = [-1], inorder = [-1]

Output: [-1]

Constraints:

1 <= preorder.length <= 3000

inorder.length == preorder.length

-3000 <= preorder[i], inorder[i] <= 3000

preorder and inorder consist of unique values.

Each value of inorder also appears in preorder.

preorder is guaranteed to be the preorder traversal of the tree.

inorder is guaranteed to be the inorder traversal of the tree.

**Solution 1 :**

class Solution {

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

        if(preorder.length==0) return null;

        int r=preorder[0];int index=0;

        for(int i=0;i<inorder.length;i++)

        {

            if(inorder[i]==r) index=i;

        }

        TreeNode root=new TreeNode(r);

        root.left=buildTree(Arrays.copyOfRange(preorder,1,index+1),Arrays.copyOfRange(inorder,0,index));

        root.right=buildTree(Arrays.copyOfRange(preorder,index+1,preorder.length),Arrays.copyOfRange(inorder,index+1,inorder.length));

        return root;

    }

}

**Solution – 2 : (Using Map)**

class Solution {

    Map<Integer, Integer> inorderPositions = new HashMap<>();

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

        if (preorder.length < 1 || inorder.length < 1) return null;

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

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

        }

        return builder(preorder, 0, 0, inorder.length - 1);

    }

    public TreeNode builder(

        int[] preorder,

        int preorderIndex,

        int inorderLow,

        int inorderHigh

    ) {

        if (

            preorderIndex > preorder.length - 1 || inorderLow > inorderHigh

        ) return null;

        int currentVal = preorder[preorderIndex];

        TreeNode n = new TreeNode(currentVal);

        int mid = inorderPositions.get(currentVal);

        n.left = builder(preorder, preorderIndex + 1, inorderLow, mid - 1);

        n.right =

            builder(

                preorder,

                preorderIndex + (mid - inorderLow) + 1,

                mid + 1,

                inorderHigh

            );

        return n;

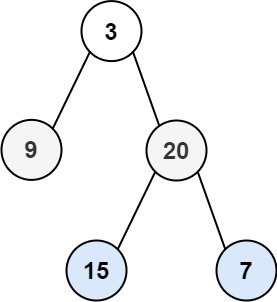
    }

}

**15) Binary Tree Level Order Traversal II (Leetcode - 107)**

Given the root of a binary tree, return the bottom-up level order traversal of its nodes' values. (i.e., from left to right, level by level from leaf to root).

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: [[15,7],[9,20],[3]]

Example 2:

Input: root = [1]

Output: [[1]]

Example 3:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 2000].

-1000 <= Node.val <= 1000

**Solution :**

class Solution {

    public List<List<Integer>> levelOrderBottom(TreeNode root) {

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

        if(root==null)

        {

            return res;

        }

        Queue<TreeNode> queue=new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty())

        {

            int levelSize=queue.size();

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

            for(int i=0;i<levelSize;i++)

            {

                TreeNode node=queue.poll();

                l.add(node.val);

                if(node.left!=null)

                {

                    queue.offer(node.left);

                }

                if(node.right!=null)

                {

                    queue.offer(node.right);

                }

            }

            res.add(0,l);

        }

        return res;

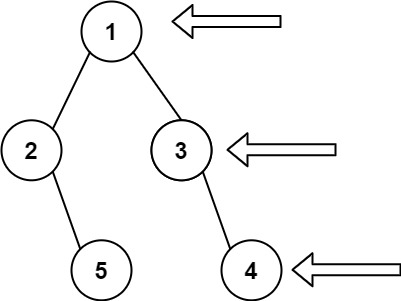
    }

}

**16) Binary Tree Right Side View (Leetcode - 199)**

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Example 1:



Input: root = [1,2,3,null,5,null,4]

Output: [1,3,4]

Example 2:

Input: root = [1,null,3]

Output: [1,3]

Example 3:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    public List<Integer> rightSideView(TreeNode root) {

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

        if(root==null)

        {

            return l;

        }

        Queue<TreeNode> queue=new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty())

        {

            int levelSize=queue.size();

            for(int i=0;i<levelSize;i++)

            {

                TreeNode node=queue.poll();

                if(i==levelSize-1)

                {

                    l.add(node.val);

                }

                if(node.left!=null)

                {

                    queue.offer(node.left);

                }

                if(node.right!=null)

                {

                    queue.offer(node.right);

                }

            }

        }

        return l;

    }

}

**17) Populating Next Right Pointers in Each Node (Leetcode - 116)**

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

struct Node {

int val;

Node \*left;

Node \*right;

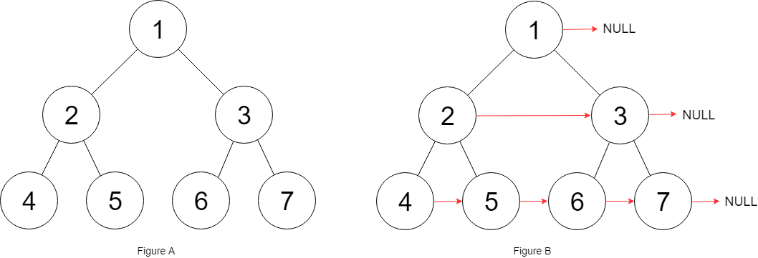
Node \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Example 1:



Input: root = [1,2,3,4,5,6,7]

Output: [1,#,2,3,#,4,5,6,7,#]

Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

Example 2:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 212 - 1].

-1000 <= Node.val <= 1000

**Solution :**

class Solution {

    public Node connect(Node root) {

        if(root==null)

        {

            return root;

        }

        Node leftMost = root;

        while(leftMost.left!=null)

        {

            Node current = leftMost;

            while(current!=null)

            {

                current.left.next=current.right;

                if(current.next!=null)

                {

                    current.right.next=current.next.left;

                }

                current=current.next;

            }

            leftMost=leftMost.left;

        }

        return root;

    }

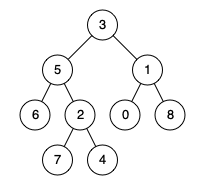
}

**18) Lowest Common Ancestor of a Binary Tree (Leetcode - 236)**

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:

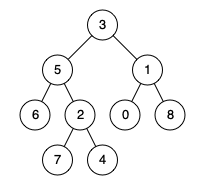


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Example 3:

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

Output: 1

Constraints:

The number of nodes in the tree is in the range [2, 105].

-109 <= Node.val <= 109

All Node.val are unique.

p != q

p and q will exist in the tree.

**Solution 1:**

class Solution {

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

        if(root==null) return null;

        if(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 root;

        return left==null?right:left;

    }

}

**Solution 2:**

class Solution {

    private static TreeNode res;

    public boolean search(TreeNode node,TreeNode e)

    {

        if(node==null) return false;

        if(node==e) return true;

        boolean left = search(node.left,e);

        if(left) return true;

        return search(node.right,e);

    }

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

        if(root==null) return root;

        if(search(root,p) && search(root,q))

        {

            res=root;

        }

        lowestCommonAncestor(root.left,p,q);

        lowestCommonAncestor(root.right,p,q);

        return res;

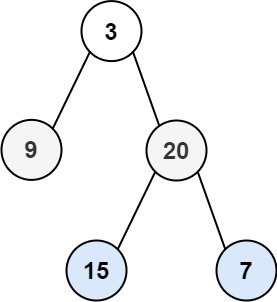
    }

}

**19) Binary Tree Zigzag Level Order Traversal (Leetcode - 103)**

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: [[3],[20,9],[15,7]]

Example 2:

Input: root = [1]

Output: [[1]]

Example 3:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 2000].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

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

        if(root==null)

        {

            return res;

        }

        Deque<TreeNode> queue=new LinkedList<>();

        queue.offer(root);

        boolean reverse=false;

        while(!queue.isEmpty())

        {

            int levelSize=queue.size();

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

            for(int i=0;i<levelSize;i++)

            {

                TreeNode node=null;

                if(!reverse)

                {

                    node=queue.poll();

                    l.add(node.val);

                    if(node.left!=null)

                    {

                        queue.addLast(node.left);

                    }

                    if(node.right!=null)

                    {

                        queue.addLast(node.right);

                    }

                }

                else

                {

                    node = queue.pollLast();

                    l.add(node.val);

                     if(node.right!=null)

                    {

                        queue.addFirst(node.right);

                    }

                    if(node.left!=null)

                    {

                        queue.addFirst(node.left);

                    }

                }

            }

           reverse=!reverse;

            res.add(l);

        }

        return res;

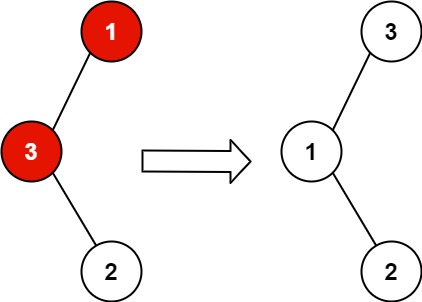
    }

}

**20) Recover Binary Search Tree (Leetcode - 99)**

You are given the root of a binary search tree (BST), where the values of exactly two nodes of the tree were swapped by mistake. Recover the tree without changing its structure.

Example 1:

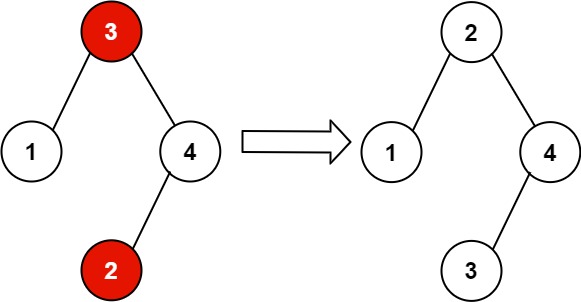


Input: root = [1,3,null,null,2]

Output: [3,1,null,null,2]

Explanation: 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BST valid.

Example 2:



Input: root = [3,1,4,null,null,2]

Output: [2,1,4,null,null,3]

Explanation: 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 makes the BST valid.

Constraints:

The number of nodes in the tree is in the range [2, 1000].

-231 <= Node.val <= 231 – 1

**Solution :**

class Solution {

    TreeNode prev,first,second;

    public void inorder(TreeNode root)

    {

        if(root==null)return ;

        inorder(root.left);

        if(prev!=null && prev.val>root.val)

        {

            if(first==null)

            {

                first=prev;

            }

            second=root;

        }

        prev=root;

        inorder(root.right);

    }

    public void recoverTree(TreeNode root) {

        inorder(root);

        int temp=first.val;

        first.val=second.val;

        second.val=temp;

    }

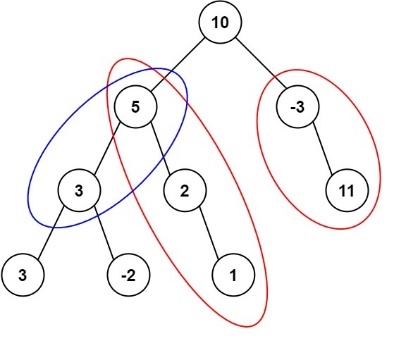
}

**21) Path Sum III (Leetcode - 437)**

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:



Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8

Output: 3

Explanation: The paths that sum to 8 are shown.

Example 2:

Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

Output: 3

Constraints:

The number of nodes in the tree is in the range [0, 1000].

-109 <= Node.val <= 109

-1000 <= targetSum <= 1000

**Solution :**

class Solution {

    int count=0;

    Map<Long,Integer> map;

    public int pathSum(TreeNode root, int targetSum) {

       map = new HashMap<>();

        map.put(0L,1);

        sum(root,0,targetSum);

        return count;

    }

    public void sum(TreeNode root,long prefix,int target)

    {

        if(root==null)

        {

            return ;

        }

        prefix+=root.val;

        count+=map.getOrDefault(prefix-target,0);

        map.put(prefix,map.getOrDefault(prefix,0)+1);

        sum(root.left,prefix,target);

        sum(root.right,prefix,target);

        map.put(prefix, map.get(prefix) - 1);

    }

}

**22) Validate Binary Search Tree (Leetcode - 98)**

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

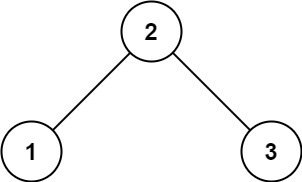
A valid BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

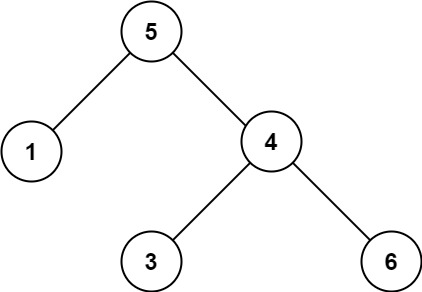
Example 1:



Input: root = [2,1,3]

Output: true

Example 2:



Input: root = [5,1,4,null,null,3,6]

Output: false

Explanation: The root node's value is 5 but its right child's value is 4.

Constraints:

The number of nodes in the tree is in the range [1, 104].

-231 <= Node.val <= 231 – 1

**Solution 1:**

class Solution {

    private static int prev;

    public boolean isValid(TreeNode root)

    {

        if(root==null) return true;

        boolean t=isValid(root.left);

        if(root.val<=prev) return false;

        prev=root.val;

        return t && isValid(root.right);

    }

    public boolean isValidBST(TreeNode root) {

        prev=Int;

        return isValid(root);

    }

}

**Solution 2:**

class Solution {

    public boolean isValidBST(TreeNode root) {

        if (root == null) return true;

        return dfs(root, null, null);

    }

    private boolean dfs(TreeNode root, Integer min, Integer max) {

        if (root == null) return true;

        if (

            (min != null && root.val <= min) || max != null && root.val >= max

        ) {

            return false;

        }

        return dfs(root.left, min, root.val) && dfs(root.right, root.val, max);

    }

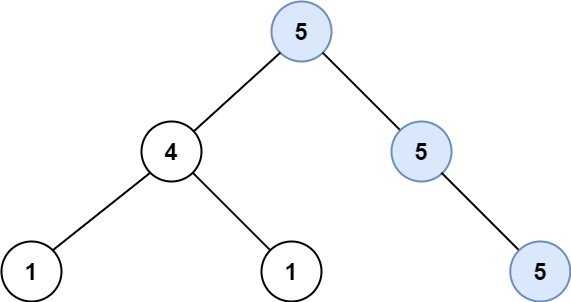
}

**23) Longest Univalue Path (Leetcode - 687)**

Given the root of a binary tree, return the length of the longest path, where each node in the path has the same value. This path may or may not pass through the root.

The length of the path between two nodes is represented by the number of edges between them.

Example 1:

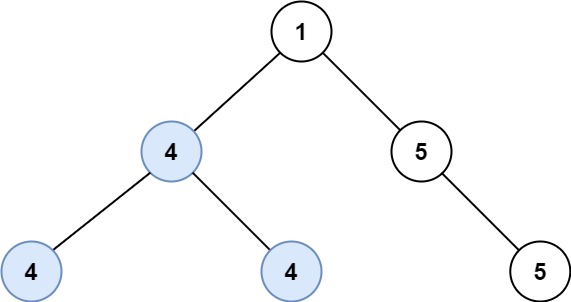


Input: root = [5,4,5,1,1,null,5]

Output: 2

Explanation: The shown image shows that the longest path of the same value (i.e. 5).

Example 2:



Input: root = [1,4,5,4,4,null,5]

Output: 2

Explanation: The shown image shows that the longest path of the same value (i.e. 4).

Constraints:

The number of nodes in the tree is in the range [0, 104].

-1000 <= Node.val <= 1000

The depth of the tree will not exceed 1000.

**Solution:**

class Solution {

    int ans;

    public int path(TreeNode root,int parent)

    {

        if(root==null) return 0;

        int left = path(root.left,root.val);

        int right = path(root.right,root.val);

         ans = Math.max(ans,left+right);

         return root.val==parent?Math.max(left,right)+1:0;

    }

    public int longestUnivaluePath(TreeNode root) {

        path(root,-1);

        return ans;

    }

}

**24) Closest Nodes Queries in a Binary Search Tree (Leetcode - 2476)**

You are given the root of a binary search tree and an array queries of size n consisting of positive integers.

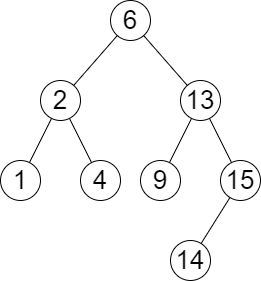
Find a 2D array answer of size n where answer[i] = [mini, maxi]:

mini is the largest value in the tree that is smaller than or equal to queries[i]. If a such value does not exist, add -1 instead.

maxi is the smallest value in the tree that is greater than or equal to queries[i]. If a such value does not exist, add -1 instead.

Return the array answer.

Example 1:



Input: root = [6,2,13,1,4,9,15,null,null,null,null,null,null,14], queries = [2,5,16]

Output: [[2,2],[4,6],[15,-1]]

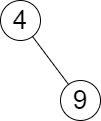
Explanation: We answer the queries in the following way:

- The largest number that is smaller or equal than 2 in the tree is 2, and the smallest number that is greater or equal than 2 is still 2. So the answer for the first query is [2,2].

- The largest number that is smaller or equal than 5 in the tree is 4, and the smallest number that is greater or equal than 5 is 6. So the answer for the second query is [4,6].

- The largest number that is smaller or equal than 16 in the tree is 15, and the smallest number that is greater or equal than 16 does not exist. So the answer for the third query is [15,-1].

Example 2:



Input: root = [4,null,9], queries = [3]

Output: [[-1,4]]

Explanation: The largest number that is smaller or equal to 3 in the tree does not exist, and the smallest number that is greater or equal to 3 is 4. So the answer for the query is [-1,4].

Constraints:

The number of nodes in the tree is in the range [2, 105].

1 <= Node.val <= 106

n == queries.length

1 <= n <= 105

1 <= queries[i] <= 106

**Solution :**

class Solution {

    public List<List<Integer>> closestNodes(TreeNode root, List<Integer> queries) {

        TreeSet<Integer> tree=new TreeSet<>();

        addition(root,tree);

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

        for(int i=0;i<queries.size();i++){

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

           Integer a=tree.floor(queries.get(i));

            Integer b=tree.ceiling(queries.get(i));

            if(a==null){

                temp.add(-1);

            }else{

                temp.add(a);

            }

            if(b==null){

                temp.add(-1);

            }else{

                temp.add(b);

            }

            output.add(temp);

        }

        return output;

    }

    public void addition(TreeNode root,Set<Integer> tree){

        if(root==null){

            return;

        }

        tree.add(root.val);

        addition(root.left,tree);

        addition(root.right,tree);

    }

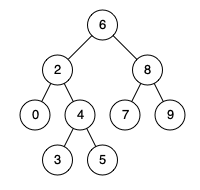
}

**25) Lowest Common Ancestor of a Binary Search Tree (Leetcode - 235)**

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:

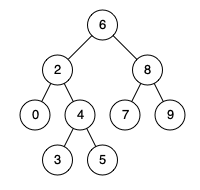


Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

Output: 6

Explanation: The LCA of nodes 2 and 8 is 6.

Example 2:



Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

Output: 2

Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Example 3:

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

Output: 2

Constraints:

The number of nodes in the tree is in the range [2, 105].

-109 <= Node.val <= 109

All Node.val are unique.

p != q

p and q will exist in the BST

**Solution :**

class Solution {

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

        if(root == null) return null;

        if(root.val<p.val && root.val<q.val)

        {

            return lowestCommonAncestor(root.right,p,q);

        }

        if(root.val>p.val && root.val>q.val)

        {

            return lowestCommonAncestor(root.left,p,q);

        }

        return root;

    }

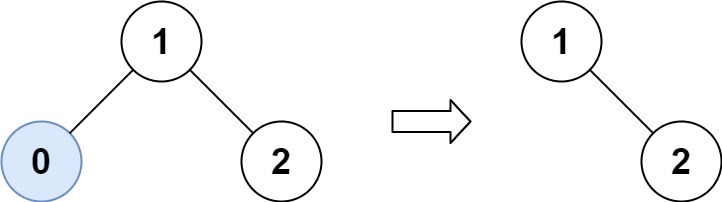
}

**26) Trim a Binary Search Tree (Leetcode - 669)**

Given the root of a binary search tree and the lowest and highest boundaries as low and high, trim the tree so that all its elements lies in [low, high]. Trimming the tree should not change the relative structure of the elements that will remain in the tree (i.e., any node's descendant should remain a descendant). It can be proven that there is a unique answer.

Return the root of the trimmed binary search tree. Note that the root may change depending on the given bounds.

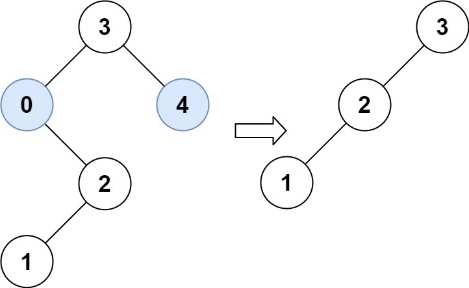
Example 1:



Input: root = [1,0,2], low = 1, high = 2

Output: [1,null,2]

Example 2:



Input: root = [3,0,4,null,2,null,null,1], low = 1, high = 3

Output: [3,2,null,1]

Constraints:

The number of nodes in the tree is in the range [1, 104].

0 <= Node.val <= 104

The value of each node in the tree is unique.

root is guaranteed to be a valid binary search tree.

0 <= low <= high <= 104

**Solution :**

class Solution {

    public TreeNode trimBST(TreeNode root, int low, int high) {

        if(root==null) return null;

        if(root.val > high)

        {

            return trimBST(root.left,low,high);

        }

        if(root.val < low)

        {

            return trimBST(root.right,low,high);

        }

        root.left=trimBST(root.left,low,high);

        root.right=trimBST(root.right,low,high);

        return root;

    }

}

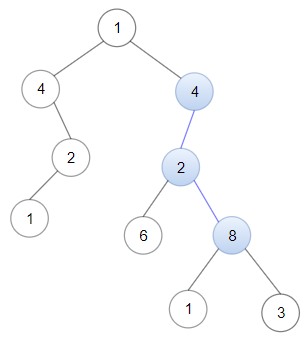
**27) Linked List in Binary Tree (Leetcode – 1367)**

Given a binary tree root and a linked list with head as the first node.

Return True if all the elements in the linked list starting from the head correspond to some downward path connected in the binary tree otherwise return False.

In this context downward path means a path that starts at some node and goes downwards.

Example 1:

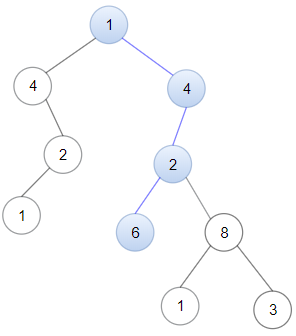
****

Input: head = [4,2,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]

Output: true

Explanation: Nodes in blue form a subpath in the binary Tree.

Example 2:



Input: head = [1,4,2,6], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]

Output: true

Example 3:

Input: head = [1,4,2,6,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]

Output: false

Explanation: There is no path in the binary tree that contains all the elements of the linked list from head.

Constraints:

The number of nodes in the tree will be in the range [1, 2500].

The number of nodes in the list will be in the range [1, 100].

1 <= Node.val <= 100 for each node in the linked list and binary tree.

**Solution :**

class Solution {

    public boolean findRoot(TreeNode root,ListNode head)

    {

        if(root==null) return false;

        if(check(root,head)) return true ;

        return findRoot(root.left,head) || findRoot(root.right,head);

    }

    public boolean check(TreeNode root,ListNode head)

    {

        if(head==null) return true;

        if(root==null) return false;

       if(root.val!=head.val) return false;

       return check(root.left,head.next) || check(root.right,head.next);

    }

    public boolean isSubPath(ListNode head, TreeNode root) {

        return findRoot(root,head);

    }

}

**28) Maximum Width of Binary Tree (Leetcode - 662)**

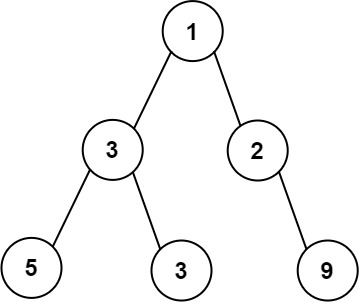
Given the root of a binary tree, return the maximum width of the given tree.

The maximum width of a tree is the maximum width among all levels.

The width of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

It is guaranteed that the answer will in the range of a 32-bit signed integer.

Example 1:

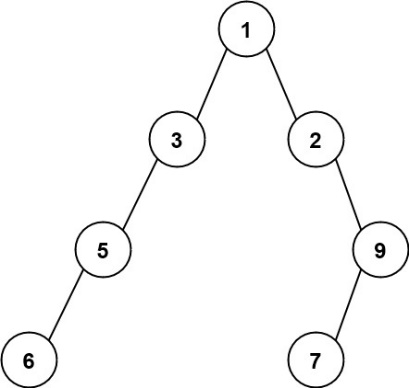


Input: root = [1,3,2,5,3,null,9]

Output: 4

Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).

Example 2:

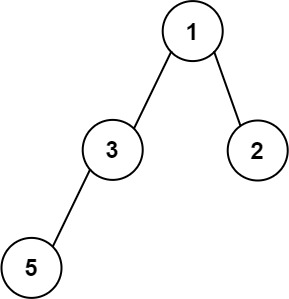


Input: root = [1,3,2,5,null,null,9,6,null,7]

Output: 7

Explanation: The maximum width exists in the fourth level with length 7 (6,null,null,null,null,null,7).

Example 3:



Input: root = [1,3,2,5]

Output: 2

Explanation: The maximum width exists in the second level with length 2 (3,2).

Constraints:

The number of nodes in the tree is in the range [1, 3000].

-100 <= Node.val <= 100

**Solution :**

class Solution {

    class Pair

    {

        int num;

        TreeNode node;

        Pair(TreeNode node,int num)

        {

            this.num = num;

            this.node = node;

        }

    }

    public int widthOfBinaryTree(TreeNode root) {

        Queue<Pair> queue = new LinkedList<>();

        queue.offer(new Pair(root,0));

        int width = 0;

        while(!queue.isEmpty())

        {

            int size = queue.size();

            int min = queue.peek().num;//to make every level starting with zero

            int last=0;

            for(int i=0;i<size;i++)

            {

                Pair cur = queue.poll();

                TreeNode currentNode= cur.node;

                int index = cur.num-min;

                if(i==size-1) last=cur.num;

                if(currentNode.left!=null)

                {

                    queue.offer(new Pair(currentNode.left,2\*index+1));

                }

                if(currentNode.right!=null)

                {

                    queue.offer(new Pair(currentNode.right,2\*index+2));

                }

            }

            width=Math.max(last-min+1,width);

        }

        return width;

    }

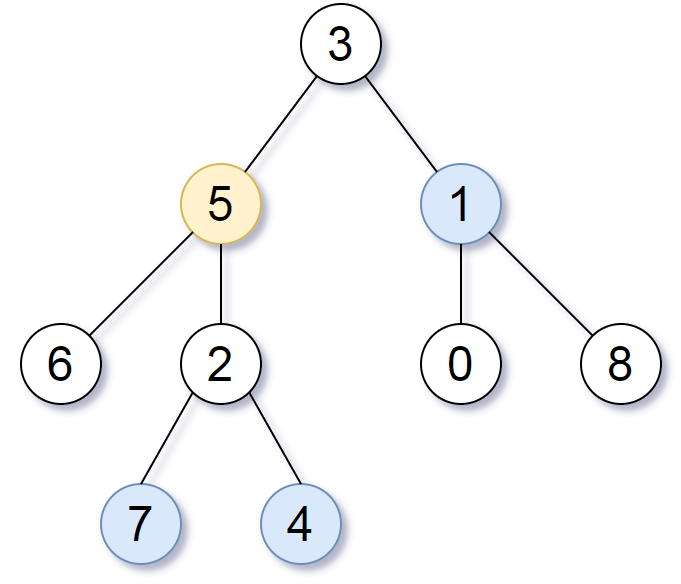
}

**29) All Nodes Distance K in Binary Tree (Leetcode - 863)**

Given the root of a binary tree, the value of a target node target, and an integer k, return an array of the values of all nodes that have a distance k from the target node.

You can return the answer in any order.

Example 1:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2

Output: [7,4,1]

Explanation: The nodes that are a distance 2 from the target node (with value 5) have values 7, 4, and 1.

Example 2:

Input: root = [1], target = 1, k = 3

Output: []

Constraints:

The number of nodes in the tree is in the range [1, 500].

0 <= Node.val <= 500

All the values Node.val are unique.

target is the value of one of the nodes in the tree.

0 <= k <= 1000

**Solution :**

class Solution {

    HashMap<TreeNode,TreeNode> parent = new HashMap<>();

    public void markParent(TreeNode root)

    {

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty())

        {

            TreeNode curr = queue.poll();

            if(curr.left!=null)

            {

                queue.offer(curr.left);

                parent.put(curr.left,curr);

            }

            if(curr.right!=null)

            {

                queue.offer(curr.right);

                parent.put(curr.right,curr);

            }

        }

    }

    public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {

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

        if(k==0)

        {

            res.add(target.val);

            return res;

        }

        HashSet<TreeNode> visited = new HashSet<>();

        Queue<TreeNode> queue = new LinkedList<>();

        markParent(root);

        queue.add(target);

        visited.add(target);

        int dis=0;

        while(!queue.isEmpty())

        {

            int size = queue.size();

            dis++;

            for(int i=0;i<size;i++)

            {

                TreeNode cur = queue.poll();

                if(cur.left!=null && !visited.contains(cur.left))

                {

                    queue.offer(cur.left);

                }

                if(cur.right!=null && !visited.contains(cur.right))

                {

                    queue.offer(cur.right);

                }

                if(parent.get(cur)!=null && !visited.contains(parent.get(cur)))

                {

                    queue.offer(parent.get(cur));

                }

                visited.add(cur);

            }

            if(dis==k)

            {

                break;

            }

        }

        while(!queue.isEmpty())

        {

            res.add(queue.poll().val);

        }

        return res;

    }

}

**30) Kth Largest Sum in a Binary Tree (Leetcode - 2583)**

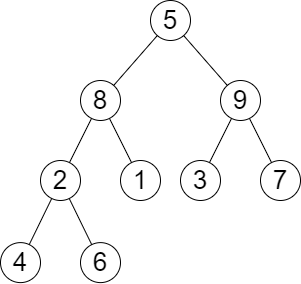
You are given the root of a binary tree and a positive integer k.

The level sum in the tree is the sum of the values of the nodes that are on the same level.

Return the kth largest level sum in the tree (not necessarily distinct). If there are fewer than k levels in the tree, return -1.

Note that two nodes are on the same level if they have the same distance from the root.

Example 1:



Input: root = [5,8,9,2,1,3,7,4,6], k = 2

Output: 13

Explanation: The level sums are the following:

- Level 1: 5.

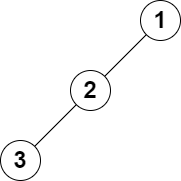
- Level 2: 8 + 9 = 17.

- Level 3: 2 + 1 + 3 + 7 = 13.

- Level 4: 4 + 6 = 10.

The 2nd largest level sum is 13.

Example 2:



Input: root = [1,2,null,3], k = 1

Output: 3

Explanation: The largest level sum is 3.

Constraints:

The number of nodes in the tree is n.

2 <= n <= 105

1 <= Node.val <= 106

1 <= k <= n

**Solution :**

public class Solution {

    public long kthLargestLevelSum(TreeNode root, int k) {

        if (root == null) return -1;

        Queue<TreeNode> q = new LinkedList<>();

        PriorityQueue<Long> pq = new PriorityQueue<>(Collections.reverseOrder());

        q.add(root);

        while (!q.isEmpty()) {

            int size = q.size();

            long sum = 0;

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

                TreeNode curr = q.poll();

                sum += curr.val;

                if (curr.left != null) q.add(curr.left);

                if (curr.right != null) q.add(curr.right);

            }

            pq.add(sum);

        }

        if (k > pq.size()) return -1;

        while (!pq.isEmpty() && k-- > 1) {

            pq.poll();

        }

        return pq.peek();

    }

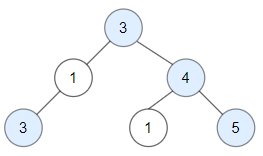
}

**31) Count Good Nodes in Binary Tree(Leetcode - 1448)**

Given a binary tree root, a node X in the tree is named good if in the path from root to X there are no nodes with a value greater than X.

Return the number of good nodes in the binary tree.

Example 1:



Input: root = [3,1,4,3,null,1,5]

Output: 4

Explanation: Nodes in blue are good.

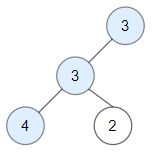
Root Node (3) is always a good node.

Node 4 -> (3,4) is the maximum value in the path starting from the root.

Node 5 -> (3,4,5) is the maximum value in the path

Node 3 -> (3,1,3) is the maximum value in the path.

Example 2:



Input: root = [3,3,null,4,2]

Output: 3

Explanation: Node 2 -> (3, 3, 2) is not good, because "3" is higher than it.

Example 3:

Input: root = [1]

Output: 1

Explanation: Root is considered as good.

Constraints:

The number of nodes in the binary tree is in the range [1, 10^5].

Each node's value is between [-10^4, 10^4].

**Solution :**

class Solution {

    public int preOrder(TreeNode root,int max)

    {

        if(root==null) return 0;

        int count=0;

        if(root.val>=max)

        {

            max=root.val;

            count+=1;

        }

        return count+preOrder(root.left,max)+preOrder(root.right,max);

    }

    public int goodNodes(TreeNode root) {

        return preOrder(root,root.val);

    }

}

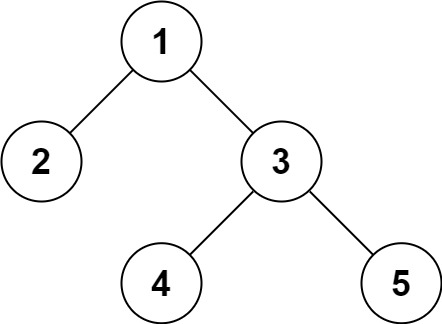
**1) Serialize and Deserialize Binary Tree (Leetcode - 297) (Hard)**

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification: The input/output format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:



Input: root = [1,2,3,null,null,4,5]

Output: [1,2,3,null,null,4,5]

Example 2:

Input: root = []

Output: []

Constraints:

The number of nodes in the tree is in the range [0, 104].

-1000 <= Node.val <= 1000

**Solution :**

public class Codec {

    private int i;

    // Encodes a tree to a single string.

    public String serialize(TreeNode root) {

        List<String> list = new ArrayList<>();

        serializeDFS(root, list);

        return String.join(",", list);

    }

    private void serializeDFS(TreeNode root, List<String> list) {

        if (root == null) {

            list.add("N");

            return;

        }

        list.add(String.valueOf(root.val));

        serializeDFS(root.left, list);

        serializeDFS(root.right, list);

    }

    // Decodes your encoded data to tree.

    public TreeNode deserialize(String data) {

        String[] tokens = data.split(",");

        return deserializeDFS(tokens);

    }

    private TreeNode deserializeDFS(String[] tokens) {

        String token = tokens[this.i];

        if (token.equals("N")) {

            this.i++;

            return null;

        }

        var node = new TreeNode(Integer.parseInt(token));

        this.i++;

        node.left = deserializeDFS(tokens);

        node.right = deserializeDFS(tokens);

        return node;

    }

}

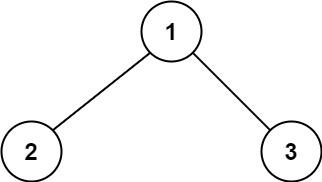
**2) Binary Tree Maximum Path Sum (Leetcode - 124)**

A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

Example 1:

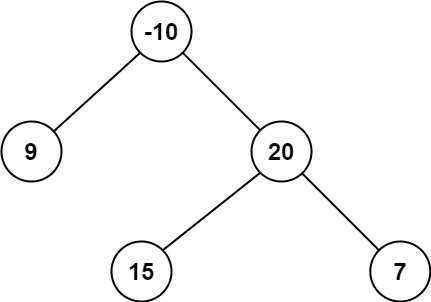


Input: root = [1,2,3]

Output: 6

Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

Example 2:



Input: root = [-10,9,20,null,null,15,7]

Output: 42

Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.

Constraints:

The number of nodes in the tree is in the range [1, 3 \* 104].

-1000 <= Node.val <= 1000

**Solution :**

class Solution {

    int max=-1000;

    public int maxPathSum(TreeNode node) {

        maxPath(node);

        return max;

    }

    public int maxPath(TreeNode node)

    {

        if(node==null) return 0;

        int left=maxPath(node.left);

        int right=maxPath(node.right);

        int sum=Math.max(right,left)+node.val;

        sum=Math.max(node.val,sum);

        max=Math.max(max,node.val+right+left);

        max=Math.max(max,sum);

        return sum;

    }

}

**3) Vertical Order Traversal of a Binary Tree (Leetcode - 987)**

Given the root of a binary tree, calculate the vertical order traversal of the binary tree.

For each node at position (row, col), its left and right children will be at positions (row + 1, col - 1) and (row + 1, col + 1) respectively. The root of the tree is at (0, 0).

The vertical order traversal of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

Return the vertical order traversal of the binary tree

Example 1:



Input: root = [3,9,20,null,null,15,7]

Output: [[9],[3,15],[20],[7]]

Explanation:

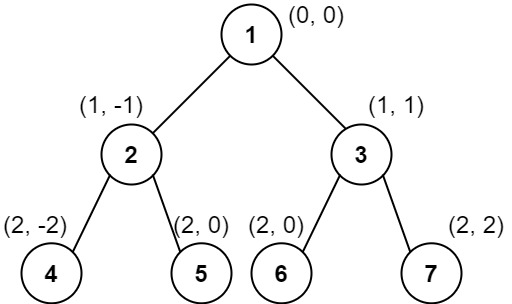
Column -1: Only node 9 is in this column.

Column 0: Nodes 3 and 15 are in this column in that order from top to bottom.

Column 1: Only node 20 is in this column.

Column 2: Only node 7 is in this column.

Example 2:



Input: root = [1,2,3,4,5,6,7]

Output: [[4],[2],[1,5,6],[3],[7]]

Explanation:

Column -2: Only node 4 is in this column.

Column -1: Only node 2 is in this column.

Column 0: Nodes 1, 5, and 6 are in this column.

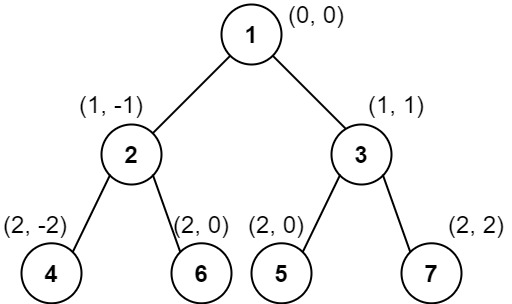
1 is at the top, so it comes first.

5 and 6 are at the same position (2, 0), so we order them by their value, 5 before 6.

Column 1: Only node 3 is in this column.

Column 2: Only node 7 is in this column.

Example 3:



Input: root = [1,2,3,4,6,5,7]

Output: [[4],[2],[1,5,6],[3],[7]]

Explanation:

This case is the exact same as example 2, but with nodes 5 and 6 swapped.

Note that the solution remains the same since 5 and 6 are in the same location and should be ordered by their values.

Constraints:

The number of nodes in the tree is in the range [1, 1000].

0 <= Node.val <= 1000

**Solution :**

class Solution {

    Map<Integer, TreeMap<Integer, PriorityQueue<Integer>>> map = new HashMap<>();

    int min=Integer.MAX\_VALUE,max=Integer.MIN\_VALUE;

    public void  vertical(TreeNode root,int pos,int level)

    {

        if(root==null) return ;

        map.putIfAbsent(pos, new TreeMap<>());

        map.get(pos).putIfAbsent(level, new PriorityQueue<>());

        map.get(pos).get(level).add(root.val);

        min=Math.min(min,pos);

        max=Math.max(max,pos);

        vertical(root.left,pos-1,level+1);

        vertical(root.right,pos+1,level+1);

    }

    public List<List<Integer>> verticalTraversal(TreeNode root) {

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

        vertical(root,0,0);

        for(int i = min;i<=max;i++)

        {

            ArrayList<Integer> list = new ArrayList<>();

            TreeMap<Integer, PriorityQueue<Integer>> tm = map.get(i);

            for (int k : tm.keySet()){

                PriorityQueue<Integer> pq = tm.get(k);

                while (!pq.isEmpty()){

                    list.add(pq.poll());

                }

            }

            res.add(list);

        }

        return res;

    }

}

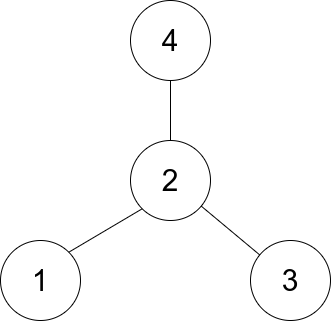
**Graphs**

**1) Find Center of Star Graph (Leetcode - 1791) (Easy)**

There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect the center node with every other node.

You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return the center of the given star graph.

Example 1:



Input: edges = [[1,2],[2,3],[4,2]]

Output: 2

Explanation: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

Example 2:

Input: edges = [[1,2],[5,1],[1,3],[1,4]]

Output: 1

Constraints:

3 <= n <= 105

edges.length == n - 1

edges[i].length == 2

1 <= ui, vi <= n

ui != vi

The given edges represent a valid star graph.

**Solution :**

class Solution {

    public int findCenter(int[][] e) {

        return e[0][0] == e[1][0] || e[0][0] == e[1][1] ? e[0][0] : e[0][1];

    }

}

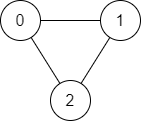
**2). Find if Path Exists in Graph (Leetcode - 1971)**

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex source to vertex destination.

Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

Example 1:



Input: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2

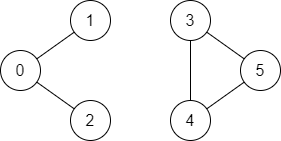
Output: true

Explanation: There are two paths from vertex 0 to vertex 2:

- 0 → 1 → 2

- 0 → 2

Example 2:



Input: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5

Output: false

Explanation: There is no path from vertex 0 to vertex 5.

Constraints:

1 <= n <= 2 \* 105

0 <= edges.length <= 2 \* 105

edges[i].length == 2

0 <= ui, vi <= n - 1

ui != vi

0 <= source, destination <= n - 1

There are no duplicate edges.

There are no self edges.

**Solution :**

class Solution {

    public boolean validPath(int n, int[][] edges, int source, int destination) {

        HashMap<Integer,List<Integer>> map = new HashMap<>();

        for(int[] edge:edges)

        {

            int u = edge[0];

            int v = edge[1];

            map.computeIfAbsent(u,k->new ArrayList<>()).add(v);

            map.computeIfAbsent(v,k->new ArrayList<>()).add(u);

        }

       return dfs(map,source,destination,n);

    }

    public boolean dfs(HashMap<Integer,List<Integer>> map,int source,int destination,int n)

    {

        Queue<Integer> queue = new LinkedList<>();

        boolean[] visited = new boolean[n];

        queue.offer(source);

        visited[source] = true;

        while (!queue.isEmpty()) {

            int current = queue.poll();

            if (current == destination) {

                return true;

            }

            for (int neighbor : map.getOrDefault(current, new ArrayList<>())) {

                if (!visited[neighbor]) {

                    visited[neighbor] = true;

                    queue.offer(neighbor);

                }

            }

        }

        return false;

    }

}

**3) Find the Town Judge (Leetcode - 997)**

In a town, there are n people labeled from 1 to n. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

The town judge trusts nobody.

Everybody (except for the town judge) trusts the town judge.

There is exactly one person that satisfies properties 1 and 2.

You are given an array trust where trust[i] = [ai, bi] representing that the person labeled ai trusts the person labeled bi. If a trust relationship does not exist in trust array, then such a trust relationship does not exist.

Return the label of the town judge if the town judge exists and can be identified, or return -1 otherwise.

Example 1:

Input: n = 2, trust = [[1,2]]

Output: 2

Example 2:

Input: n = 3, trust = [[1,3],[2,3]]

Output: 3

Example 3:

Input: n = 3, trust = [[1,3],[2,3],[3,1]]

Output: -1

Constraints:

1 <= n <= 1000

0 <= trust.length <= 104

trust[i].length == 2

All the pairs of trust are unique.

ai != bi

1 <= ai, bi <= n

**Solution :**

class Solution {

    public int findJudge(int n, int[][] trust) {

        int[] in=new int[n+1];

        int[] out=new int[n+1];

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

                out[trust[i][0]]=+1;

                in[trust[i][1]]+=1;

        }

        for (int i=1;i<=n;i++){

            if (in[i]==n-1 && out[i]==0)

            return i;

        }

        return -1;

    }

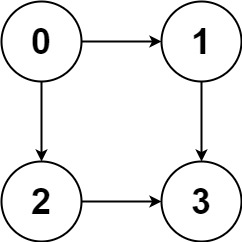
}

**1) All Paths From Source to Target (Leetcode - 797) (Medium)**

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to n - 1, find all possible paths from node 0 to node n - 1 and return them in any order.

The graph is given as follows: graph[i] is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node graph[i][j]).

Example 1:

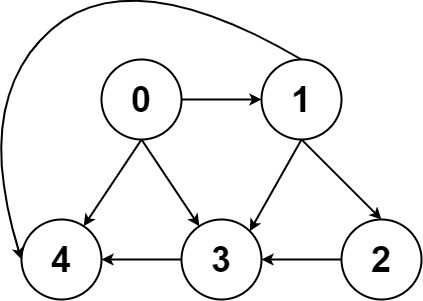


Input: graph = [[1,2],[3],[3],[]]

Output: [[0,1,3],[0,2,3]]

Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

Example 2:



Input: graph = [[4,3,1],[3,2,4],[3],[4],[]]

Output: [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

Constraints:

n == graph.length

2 <= n <= 15

0 <= graph[i][j] < n

graph[i][j] != i (i.e., there will be no self-loops).

All the elements of graph[i] are unique.

The input graph is guaranteed to be a DAG.

**Solution :**

class Solution {

    public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

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

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

        ans.add(0);

        dfs(graph,ans,list,0);

        return list;

    }

    public void dfs(int[][] graph,List<Integer> ans,List<List<Integer>> list,int i)

    {

        if(i==graph.length-1)

        {

            list.add(new ArrayList<>(ans));

        }

        for(int val: graph[i])

        {

            ans.add(val);

            dfs(graph,ans,list,val);

            ans.remove(ans.size()-1);

        }

    }

}

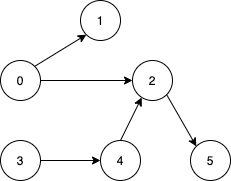
**2) Minimum Number of Vertices to Reach All Nodes (Leetcode - 1557)**

Given a directed acyclic graph, with n vertices numbered from 0 to n-1, and an array edges where edges[i] = [fromi, toi] represents a directed edge from node fromi to node toi.

Find the smallest set of vertices from which all nodes in the graph are reachable. It's guaranteed that a unique solution exists.

Notice that you can return the vertices in any order.

Example 1:

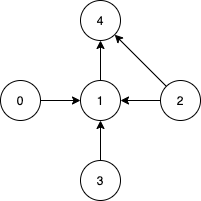


Input: n = 6, edges = [[0,1],[0,2],[2,5],[3,4],[4,2]]

Output: [0,3]

Explanation: It's not possible to reach all the nodes from a single vertex. From 0 we can reach [0,1,2,5]. From 3 we can reach [3,4,2,5]. So we output [0,3].

Example 2:



Input: n = 5, edges = [[0,1],[2,1],[3,1],[1,4],[2,4]]

Output: [0,2,3]

Explanation: Notice that vertices 0, 3 and 2 are not reachable from any other node, so we must include them. Also any of these vertices can reach nodes 1 and 4.

Constraints:

2 <= n <= 10^5

1 <= edges.length <= min(10^5, n \* (n - 1) / 2)

edges[i].length == 2

0 <= fromi, toi < n

All pairs (fromi, toi) are distinct.

**Solution :**

class Solution {

    public List<Integer> findSmallestSetOfVertices(int n, List<List<Integer>> edges) {

        int[] incoming = new int[n];

        for(List<Integer> i:edges)

        {

            incoming[i.get(1)]++;

        }

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

        for(int i=0;i<n;i++)

        {

            if(incoming[i]==0)

            {

                ans.add(i);

            }

        }

        return ans;

    }

}

**3) Keys and Rooms (Leetcode - 841)**

There are n rooms labeled from 0 to n - 1 and all the rooms are locked except for room 0. Your goal is to visit all the rooms. However, you cannot enter a locked room without having its key.

When you visit a room, you may find a set of distinct keys in it. Each key has a number on it, denoting which room it unlocks, and you can take all of them with you to unlock the other rooms.

Given an array rooms where rooms[i] is the set of keys that you can obtain if you visited room i, return true if you can visit all the rooms, or false otherwise.

Example 1:

Input: rooms = [[1],[2],[3],[]]

Output: true

Explanation:

We visit room 0 and pick up key 1.

We then visit room 1 and pick up key 2.

We then visit room 2 and pick up key 3.

We then visit room 3.

Since we were able to visit every room, we return true.

Example 2:

Input: rooms = [[1,3],[3,0,1],[2],[0]]

Output: false

Explanation: We can not enter room number 2 since the only key that unlocks it is in that room.

Constraints:

n == rooms.length

2 <= n <= 1000

0 <= rooms[i].length <= 1000

1 <= sum(rooms[i].length) <= 3000

0 <= rooms[i][j] < n

All the values of rooms[i] are unique.

**Solution :**

class Solution {

    public boolean canVisitAllRooms(List<List<Integer>> rooms) {

        boolean visited[] = new boolean[rooms.size()];

        Queue<Integer> queue = new LinkedList<>();

        queue.offer(0);

        while(!queue.isEmpty())

        {

            int cur = queue.poll();

            visited[cur]=true;

            for(int i:rooms.get(cur))

            {

                if(!visited[i])

                {

                    queue.add(i);

                }

            }

        }

        for(boolean i : visited)

        {

            if(!i)

            {

                return i;

            }

        }

        return true;

    }

}

**4) Maximum Total Importance of Roads (Leetcode - 2285)**

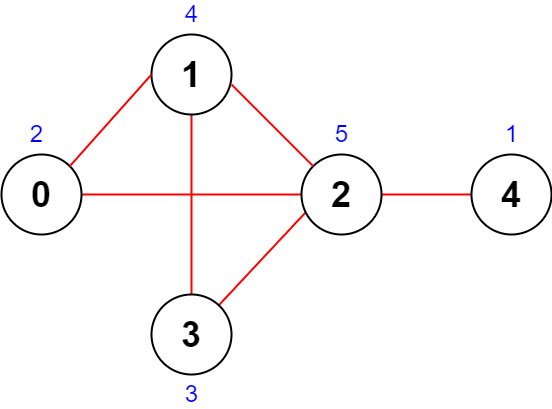
You are given an integer n denoting the number of cities in a country. The cities are numbered from 0 to n - 1.

You are also given a 2D integer array roads where roads[i] = [ai, bi] denotes that there exists a bidirectional road connecting cities ai and bi.

You need to assign each city with an integer value from 1 to n, where each value can only be used once. The importance of a road is then defined as the sum of the values of the two cities it connects.

Return the maximum total importance of all roads possible after assigning the values optimally.

Example 1:



Input: n = 5, roads = [[0,1],[1,2],[2,3],[0,2],[1,3],[2,4]]

Output: 43

Explanation: The figure above shows the country and the assigned values of [2,4,5,3,1].

- The road (0,1) has an importance of 2 + 4 = 6.

- The road (1,2) has an importance of 4 + 5 = 9.

- The road (2,3) has an importance of 5 + 3 = 8.

- The road (0,2) has an importance of 2 + 5 = 7.

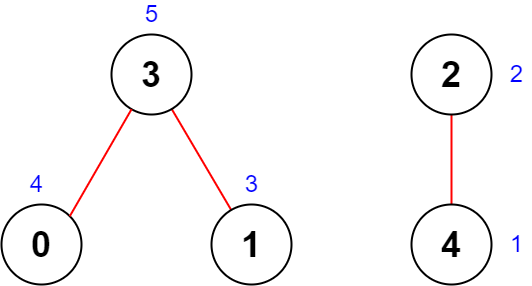
- The road (1,3) has an importance of 4 + 3 = 7.

- The road (2,4) has an importance of 5 + 1 = 6.

The total importance of all roads is 6 + 9 + 8 + 7 + 7 + 6 = 43.

It can be shown that we cannot obtain a greater total importance than 43.

Example 2:



Input: n = 5, roads = [[0,3],[2,4],[1,3]]

Output: 20

Explanation: The figure above shows the country and the assigned values of [4,3,2,5,1].

- The road (0,3) has an importance of 4 + 5 = 9.

- The road (2,4) has an importance of 2 + 1 = 3.

- The road (1,3) has an importance of 3 + 5 = 8.

The total importance of all roads is 9 + 3 + 8 = 20.

It can be shown that we cannot obtain a greater total importance than 20.

Constraints:

2 <= n <= 5 \* 104

1 <= roads.length <= 5 \* 104

roads[i].length == 2

0 <= ai, bi <= n - 1

ai != bi

There are no duplicate roads.

**Solution :**

class Solution {

    public long maximumImportance(int n, int[][] roads) {

        long degree[] = new long[n];

        for(int[] i:roads)

        {

            degree[i[0]]++;

            degree[i[1]]++;

        }

        Arrays.sort(degree);

        long max=0;

        for(int i=0;i<n;i++)

        {

            max+=degree[i]\*(i+1);

        }

        return max;

    }

}

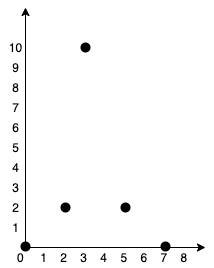
**5) Min Cost to Connect All Points (Leetcode - 1584)**

You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi].

The cost of connecting two points [xi, yi] and [xj, yj] is the manhattan distance between them: |xi - xj| + |yi - yj|, where |val| denotes the absolute value of val.

Return the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points.

Example 1:



Input: points = [[0,0],[2,2],[3,10],[5,2],[7,0]]

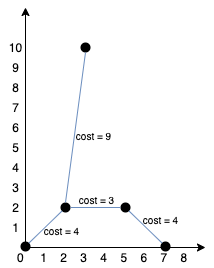
Output: 20

Explanation:

We can connect the points as shown above to get the minimum cost of 20.

Notice that there is a unique path between every pair of points.

Example 2:



Input: points = [[3,12],[-2,5],[-4,1]]

Output: 18

Constraints:

1 <= points.length <= 1000

-106 <= xi, yi <= 106

All pairs (xi, yi) are distinct.

**Solution :**

class Solution {

    public int minCostConnectPoints(int[][] points) {

        PriorityQueue<int[]> queue = new PriorityQueue<>((a,b)->a[0]-b[0]);

        Set<Integer> set = new HashSet<>();

        int len=points.length;

        int cost=0;

        queue.offer(new int[]{0,0});

        while(!queue.isEmpty())

        {

            int[] a=queue.poll();

            int weight=a[0];

            int current=a[1];

            if(set.contains(current)) continue;

            set.add(current);

            cost+=weight;

            for(int i=0;i<len;i++)

            {

                if(!set.contains(i))

                {

                    int nextWeight = Math.abs(points[current][0]-points[i][0])+

                                     Math.abs(points[current][1]-points[i][1]);

                    queue.offer(new int[]{nextWeight,i});

                }

            }

        }

        return cost;

    }

}

**6) Number of Provinces(Leetcode - 547)**

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A province is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.

Return the total number of provinces.

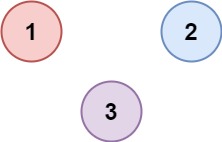
Example 1:



Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]

Output: 2

Example 2:



Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]

Output: 3

Constraints:

1 <= n <= 200

n == isConnected.length

n == isConnected[i].length

isConnected[i][j] is 1 or 0.

isConnected[i][i] == 1

isConnected[i][j] == isConnected[j][i]

**Solution 1:(Using union find)**

class Solution {

    public int find(int i,int[] parent)

    {

       if(i==parent[i])

       {

        return i;

       }

       return parent[i]=find(parent[i],parent);

    }

    public void union(int i,int j,int[] parent,int[] rank)

    {

        int parentOfI = find(i, parent);

        int parentOfJ = find(j, parent);

        if (parentOfI == parentOfJ) {

            return;

        }

        if (rank[parentOfI] < rank[parentOfJ]) {

            parent[parentOfI] = parentOfJ;

            rank[parentOfJ]+=1;

        } else  {

            parent[parentOfJ] = parentOfI;

            rank[parentOfI]+=1;

        }

    }

    public int findCircleNum(int[][] isConnected) {

       int n=isConnected.length;

       int[] parent = new int[n];

       int[] rank = new int[n];

       for(int i=0;i<n;i++)

       {

         parent[i]=i;

       }

       for(int i=0;i<n;i++)

       {

        for(int j=0;j<n;j++)

        {

            if(isConnected[i][j]==1)

            {

                union(i,j,parent,rank);

            }

        }

       }

        System.out.println(Arrays.toString(parent)+" "+Arrays.toString(rank));

       int provinces = 0;

        for (int i = 0; i < n; i++) {

            if (parent[i] == i) {

                provinces++;

            }

        }

        return provinces;

    }

}

**Solution 2:(Using BFS)**

class Solution {

     public void bfs(int node, int[][] isConnected, boolean[] visit) {

        Queue<Integer> q = new LinkedList<>();

        q.offer(node);

        visit[node] = true;

        while (!q.isEmpty()) {

            node = q.poll();

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

                if (isConnected[node][i] == 1 && !visit[i]) {

                    q.offer(i);

                    visit[i] = true;

                }

            }

        }

    }

    public int findCircleNum(int[][] isConnected) {

        int n = isConnected.length;

        int numberOfComponents = 0;

        boolean[] visit = new boolean[n];

        for (int i = 0; i < n; i++) {

            if (!visit[i]) {

                numberOfComponents++;

                bfs(i, isConnected, visit);

            }

        }

        return numberOfComponents;

    }

}

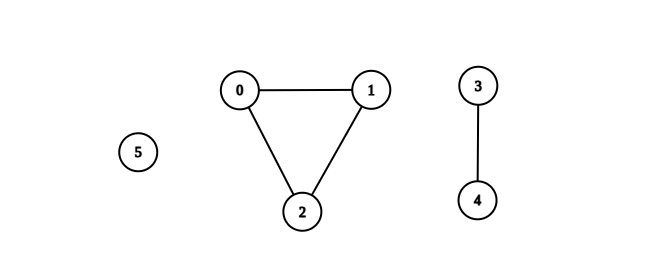
**7) Count the Number of Complete Components (Leetcode - 2685)**

You are given an integer n. There is an undirected graph with n vertices, numbered from 0 to n - 1. You are given a 2D integer array edges where edges[i] = [ai, bi] denotes that there exists an undirected edge connecting vertices ai and bi.

Return the number of complete connected components of the graph.

A connected component is a subgraph of a graph in which there exists a path between any two vertices, and no vertex of the subgraph shares an edge with a vertex outside of the subgraph.

A connected component is said to be complete if there exists an edge between every pair of its vertices.

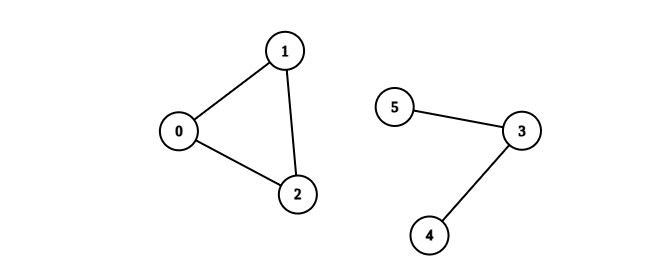
Example 1:  


Input: n = 6, edges = [[0,1],[0,2],[1,2],[3,4]]

Output: 3

Explanation: From the picture above, one can see that all of the components of this graph are complete.

Example 2:



Input: n = 6, edges = [[0,1],[0,2],[1,2],[3,4],[3,5]]

Output: 1

Explanation: The component containing vertices 0, 1, and 2 is complete since there is an edge between every pair of two vertices. On the other hand, the component containing vertices 3, 4, and 5 is not complete since there is no edge between vertices 4 and 5. Thus, the number of complete components in this graph is 1.

Constraints:

1 <= n <= 50

0 <= edges.length <= n \* (n - 1) / 2

edges[i].length == 2

0 <= ai, bi <= n - 1

ai != bi

There are no repeated edges.

**Solution :**

class Solution {

    public int findParent(int[] parent,int i)

    {

        if(i==parent[i])

        {

            return i;

        }

        return parent[i]=findParent(parent,parent[i]);

    }

    public int union(int i,int j,int[] parent,int rank[])

    {

        int first = findParent(parent,i);

        int second = findParent(parent,j);

        if(first==second) return 1;

        if(rank[second]>rank[first])

        {

            parent[first]=second;

            rank[second]++;

        }

        else

        {

            parent[second]=first;

            rank[first]++;

        }

        return 0;

    }

    public int countCompleteComponents(int n, int[][] edges)

    {

        int[] parent = new int[n];

        int[] rank = new int[n];

        for(int i=0;i<n;i++)

        {

            parent[i]=i;

        }

        int[] edgeCount = new int[n];

        for(int[] edge:edges)

        {

            union(edge[0],edge[1],parent,rank);

            edgeCount[edge[0]]++;

            edgeCount[edge[1]]++;

        }

        Set<Integer> set = new HashSet<Integer>();

        for(int i=0;i<n;i++)

        {

            set.add(findParent(parent,i));

        }

        for (int i = 0; i < n; ++i) {

            int root = findParent(parent,i);

            if (rank[root] != edgeCount[i]) {

                set.remove(root);

            }

        }

         System.out.println(Arrays.toString(rank));

        return set.size();

        //connected component no of edges in each vertex == rank[parent] in second case 4 and 5 not satisfy;

    }

}

**8) Minimum Fuel Cost to Report to the Capital (Leetcode - 2477)**

There is a tree (i.e., a connected, undirected graph with no cycles) structure country network consisting of n cities numbered from 0 to n - 1 and exactly n - 1 roads. The capital city is city 0. You are given a 2D integer array roads where roads[i] = [ai, bi] denotes that there exists a bidirectional road connecting cities ai and bi.

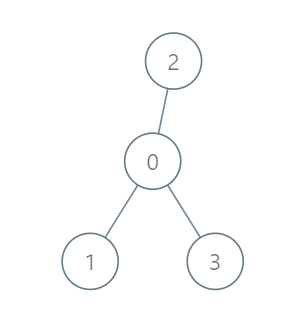
There is a meeting for the representatives of each city. The meeting is in the capital city.

There is a car in each city. You are given an integer seats that indicates the number of seats in each car.

A representative can use the car in their city to travel or change the car and ride with another representative. The cost of traveling between two cities is one liter of fuel.

Return the minimum number of liters of fuel to reach the capital city.

Example 1:



Input: roads = [[0,1],[0,2],[0,3]], seats = 5

Output: 3

Explanation:

- Representative1 goes directly to the capital with 1 liter of fuel.

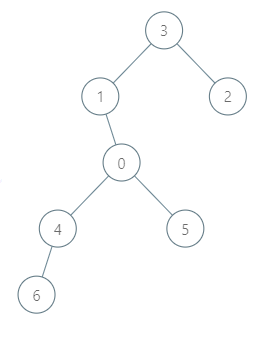
- Representative2 goes directly to the capital with 1 liter of fuel.

- Representative3 goes directly to the capital with 1 liter of fuel.

It costs 3 liters of fuel at minimum.

It can be proven that 3 is the minimum number of liters of fuel needed.

Example 2:



Input: roads = [[3,1],[3,2],[1,0],[0,4],[0,5],[4,6]], seats = 2

Output: 7

Explanation:

- Representative2 goes directly to city 3 with 1 liter of fuel.

- Representative2 and representative3 go together to city 1 with 1 liter of fuel.

- Representative2 and representative3 go together to the capital with 1 liter of fuel.

- Representative1 goes directly to the capital with 1 liter of fuel.

- Representative5 goes directly to the capital with 1 liter of fuel.

- Representative6 goes directly to city 4 with 1 liter of fuel.

- Representative4 and representative6 go together to the capital with 1 liter of fuel.

It costs 7 liters of fuel at minimum.

It can be proven that 7 is the minimum number of liters of fuel needed.

Example 3:



Input: roads = [], seats = 1

Output: 0

Explanation: No representatives need to travel to the capital city.

Constraints:

1 <= n <= 105

roads.length == n - 1

roads[i].length == 2

0 <= ai, bi < n

ai != bi

roads represents a valid tree.

1 <= seats <= 105

**Solution:**

class Solution {

    HashMap<Integer,ArrayList<Integer>> map ;

    long fuel=0;

    public long minimumFuelCost(int[][] roads, int seats) {

        map = new HashMap<>();

        for(int[] edge:roads)

        {

            map.computeIfAbsent(edge[0],val->new ArrayList<>()).add(edge[1]);

            map.computeIfAbsent(edge[1],val->new ArrayList<>()).add(edge[0]);

        }

        boolean[] vis = new boolean[map.size()];

        if(map.size()==0)

        {

            return 0;

        }

        dfs(0,vis,seats);

        return fuel;

    }

    public int dfs(int node,boolean[] vis,int seats)

    {

        int passengers=0;

        vis[node]=true;

        for(int child:map.get(node))

        {

            if(!vis[child])

            {

                int p= dfs(child,vis,seats);

                System.out.println(node+" "+child+" "+p);

                passengers+=p;

                fuel+=(int)Math.ceil((double)p/seats);

            }

        }

        return passengers+1;

    }

}

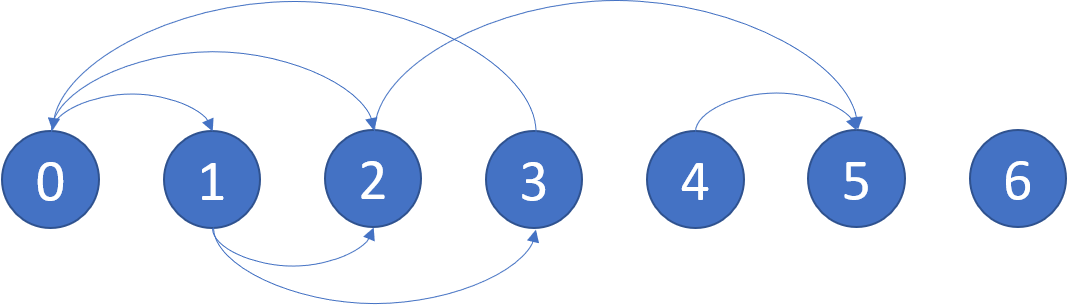
**9). Find Eventual Safe States (Leetcode - 802)**

There is a directed graph of n nodes with each node labeled from 0 to n - 1. The graph is represented by a 0-indexed 2D integer array graph where graph[i] is an integer array of nodes adjacent to node i, meaning there is an edge from node i to each node in graph[i].

A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node (or another safe node).

Return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

Example 1:



Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]

Output: [2,4,5,6]

Explanation: The given graph is shown above.

Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of them.

Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.

Example 2:

Input: graph = [[1,2,3,4],[1,2],[3,4],[0,4],[]]

Output: [4]

Explanation:

Only node 4 is a terminal node, and every path starting at node 4 leads to node 4.

Constraints:

n == graph.length

1 <= n <= 104

0 <= graph[i].length <= n

0 <= graph[i][j] <= n - 1

graph[i] is sorted in a strictly increasing order.

The graph may contain self-loops.

The number of edges in the graph will be in the range [1, 4 \* 104].

**Solution:**

class Solution {

    public List<Integer> eventualSafeNodes(int[][] graph) {

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

       HashMap<Integer,Boolean> safe = new HashMap<>();

       for(int i=0;i<graph.length;i++)

       {

            if(dfs(i,graph,safe))

            {

                res.add(i);

            }

       }

       return res;

    }

    public boolean dfs(int i,int[][] graph,HashMap<Integer,Boolean> safe)

    {

        if(safe.containsKey(i))

        {

            return safe.get(i);

        }

        safe.put(i,false);

        for(int neighbor:graph[i])

        {

            System.out.println(neighbor);

            if(!dfs(neighbor,graph,safe))

            {

                return false;

            }

        }

        safe.put(i,true);

        return true;

    }

}

**Using topoSort :**

class Solution {

List<Integer> eventualSafeNodes(int V, List<List<Integer>> adj) {

//Reverse the edges

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

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

adjRev.add(new ArrayList<>());

}

int indegree[] = new int[V];

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

*// i -> it*

*// it -> i*

for (int it : adj.get(i)) {

adjRev.get(it).add(i);

indegree[i]++;

}

}

Queue<Integer> q = new LinkedList<>();

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

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

if (indegree[i] == 0) {

q.add(i);

}

}

while (!q.isEmpty()) {

int node = q.peek();

q.remove();

safeNodes.add(node);

for (int it : adjRev.get(node)) {

indegree[it]--;

if (indegree[it] == 0) q.add(it);

}

}

Collections.sort(safeNodes);

return safeNodes;

}

}

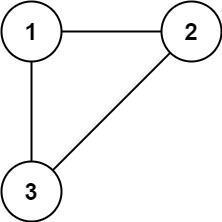
**10) Redundant Connection (Leetcode - 684)**

In this problem, a tree is an undirected graph that is connected and has no cycles.

You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed. The graph is represented as an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the graph.

Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input.

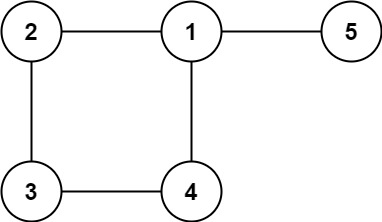
Example 1:



Input: edges = [[1,2],[1,3],[2,3]]

Output: [2,3]

Example 2:



Input: edges = [[1,2],[2,3],[3,4],[1,4],[1,5]]

Output: [1,4]

Constraints:

n == edges.length

3 <= n <= 1000

edges[i].length == 2

1 <= ai < bi <= edges.length

ai != bi

There are no repeated edges.

The given graph is connected

**Solution :**

class Solution {

    public int find(int i,int[] parent)

    {

        if(i==parent[i])

        {

            return i;

        }

        return parent[i]=find(parent[i],parent);

    }

    public boolean union(int i,int j,int[] parent,int rank[])

    {

        int first = find(i,parent);

        int second = find(j,parent);

        if(first==second) return false;,

        if(rank[second]>rank[first])

        {

            parent[first]=second;

            rank[second]+=1;

        }

        else{

            parent[second]=first;

            rank[first]+=1;

        }

        return true;

    }

    public int[] findRedundantConnection(int[][] edges) {

        int n=edges.length;

        int[] parent = new int[n+1];

        int[] rank = new int[n+1];

        for(int i=0;i<n;i++)

        {

            parent[i]=i;

        }

        for(int[] edge : edges)

        {

            if(!union(edge[0],edge[1],parent,rank))

            {

                return edge;

            }

        }

        return new int[] {0,0};

    }

}

**11) Clone Graph (Leetcode - 133)**

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

class Node {

public int val;

public List<Node> neighbors;

}

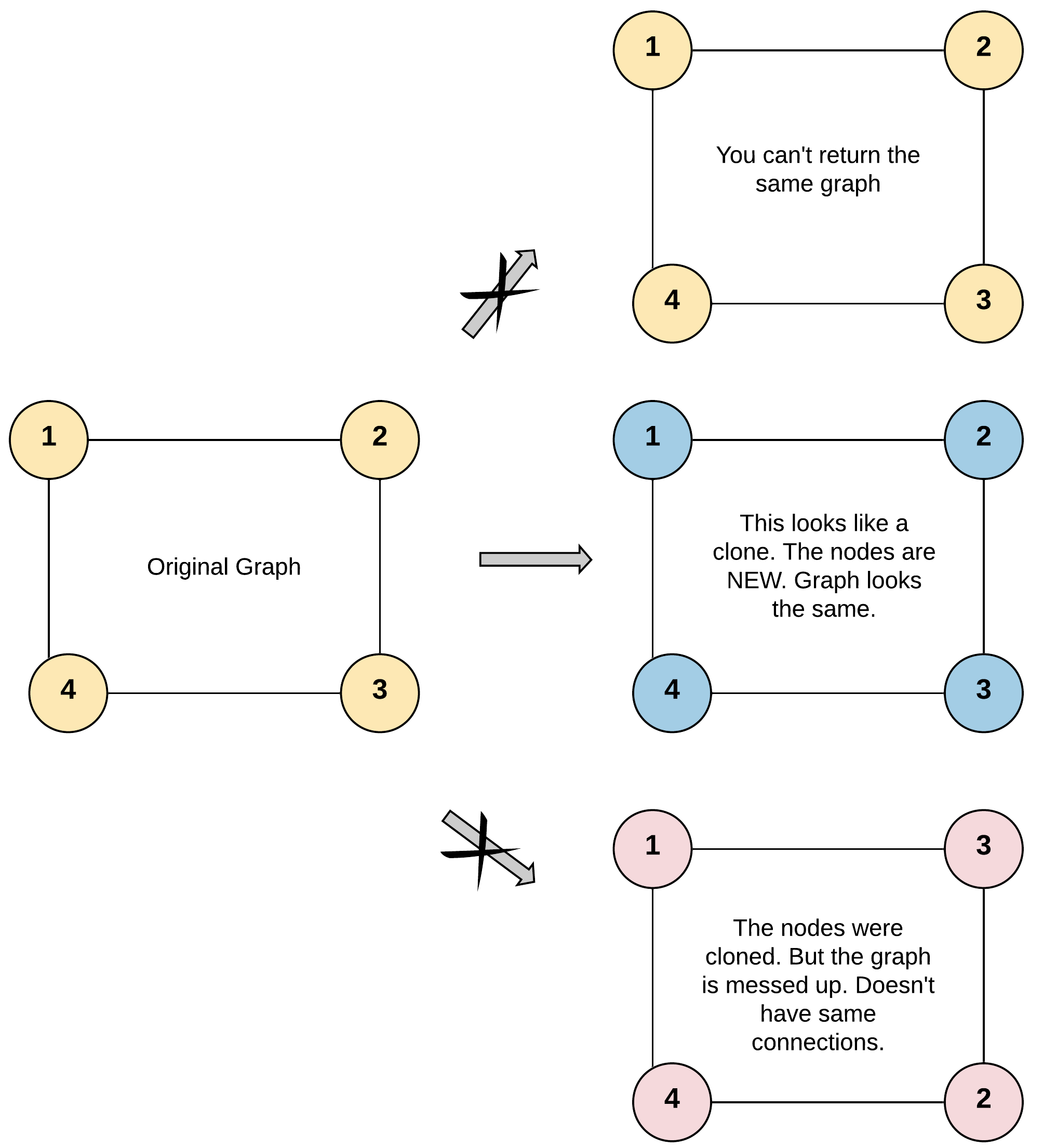
Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the copy of the given node as a reference to the cloned graph.

Example 1:



Input: adjList = [[2,4],[1,3],[2,4],[1,3]]

Output: [[2,4],[1,3],[2,4],[1,3]]

Explanation: There are 4 nodes in the graph.

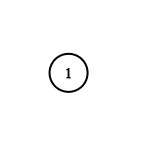
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

Example 2:



Input: adjList = [[]]

Output: [[]]

Explanation: Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

Example 3:

Input: adjList = []

Output: []

Explanation: This an empty graph, it does not have any nodes.

Constraints:

The number of nodes in the graph is in the range [0, 100].

1 <= Node.val <= 100

Node.val is unique for each node.

There are no repeated edges and no self-loops in the graph.

The Graph is connected and all nodes can be visited starting from the given node.

**Solution :**

class Solution {

    public Node cloneGraph(Node node) {

        if(node==null) return null;

        HashMap<Node,Node> map1 = new HashMap<>();

        Queue<Node> queue = new LinkedList<>();

        map1.put(node,new Node(node.val));

        queue.add(node);

        while(!queue.isEmpty())

        {

            Node curr=queue.poll();

            for(Node neighbor:curr.neighbors)

            {

                if(!map1.containsKey(neighbor))

                {

                     map1.put(neighbor,new Node(neighbor.val));

                    queue.offer(neighbor);

                }

                map1.get(curr).neighbors.add(map1.get(neighbor));

            }

        }

        return map1.get(node);

    }

}

**12) Number of Islands (Leetcode - 200)**

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input: grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

Output: 1

Example 2:

Input: grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

Output: 3

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 300

grid[i][j] is '0' or '1'.

**Solutions :**

class Solution {

    public void dfs(char[][] grid,int i,int j)

    {

        if( i<0 || j<0 || i>=grid.length || j>=grid[0].length|| grid[i][j]=='0')

        {

            return ;

        }

        grid[i][j]='0';

        dfs(grid,i,j-1); //left

        dfs(grid,i,j+1); //right

        dfs(grid,i-1,j); //up

        dfs(grid,i+1,j); //down

    }

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

        int islands=0;

        int n=grid.length;

        for(int i=0;i<n;i++)

        {

            for(int j=0;j<grid[0].length;j++)

            {

                if(grid[i][j]=='1')

                {

                    System.out.println(i+" "+j);

                    dfs(grid,i,j);

                    islands++;

                }

            }

        }

        return islands;

    }

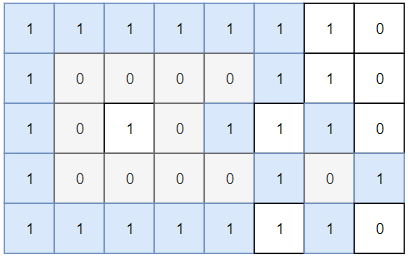
}

**13) Number of Closed Islands (Leetcode – 1254)**

Given a 2D grid consists of 0s (land) and 1s (water). An island is a maximal 4-directionally connected group of 0s and a closed island is an island totally (all left, top, right, bottom) surrounded by 1s.

Return the number of closed islands.

Example 1:



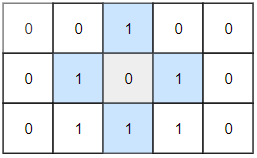
Input: grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],[1,0,0,0,0,1,0,1],[1,1,1,1,1,1,1,0]]

Output: 2

Explanation:

Islands in gray are closed because they are completely surrounded by water (group of 1s).

Example 2:



Input: grid = [[0,0,1,0,0],[0,1,0,1,0],[0,1,1,1,0]]

Output: 1

Example 3:

Input: grid = [[1,1,1,1,1,1,1],

[1,0,0,0,0,0,1],

[1,0,1,1,1,0,1],

[1,0,1,0,1,0,1],

[1,0,1,1,1,0,1],

[1,0,0,0,0,0,1],

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

Output: 2

Constraints:

1 <= grid.length, grid[0].length <= 100

0 <= grid[i][j] <=1

**Solution :**

class Solution {

    public int dfs(int[][] grid,int i,int j)

    {

        if( i<0 || j<0 || i>=grid.length || j>=grid[0].length)

        {

            return 0;

        }

        if(grid[i][j]==1) return 1;

        grid[i][j]=1;

        int a=dfs(grid,i-1,j);

        int b=dfs(grid,i+1,j);

        int c=dfs(grid,i,j-1);

        int d=dfs(grid,i,j+1);

        if(a==1 && b==1 && c==1 && d==1)

        return 1;

        return 0;

    }

    public int closedIsland(int[][] grid) {

        int islands=0;

        int n=grid.length;

        for(int i=0;i<n;i++)

        {

            for(int j=0;j<grid[0].length;j++)

            {

                if(grid[i][j]==0)

                {

                    islands+=dfs(grid,i,j);

                }

            }

        }

        return islands;

    }

}

**14) Network Delay Time (Leetcode - 743) (Dijkstra's Algo)**

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Example 1:

Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

Output: 2

Example 2:

Input: times = [[1,2,1]], n = 2, k = 1

Output: 1

Example 3:

Input: times = [[1,2,1]], n = 2, k = 2

Output: -1

Constraints:

1 <= k <= n <= 100

1 <= times.length <= 6000

times[i].length == 3

1 <= ui, vi <= n

ui != vi

0 <= wi <= 100

All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

**Solution : (Using priorityQueue) It can also solved using treeset.**

class Solution {

    class Edge

    {

        int v,w;

        Edge(int w,int v)

        {

            this.w=w;

            this.v=v;

        }

        public String toString()

        {

            return this.w+" "+this.v;

        }

    }

    public int dijkstras(ArrayList<Edge>[] graph,int n,int k)

    {

        int dist[] = new int[n+1];

        for(int i=0;i<=n;i++)

        {

            dist[i]=Integer.MAX\_VALUE;

        }

        dist[k]=0;

        int max=Integer.MIN\_VALUE;

        PriorityQueue<Edge> pq = new PriorityQueue<>((a,b)->a.w-b.w);

        pq.add(new Edge(0,k));

        while(!pq.isEmpty())

        {

            Edge node = pq.poll();

            int w=node.w,v=node.v;

            for(Edge ed:graph[v])

            {

                if(w+ed.w<dist[ed.v])

                {

                    dist[ed.v]=w+ed.w;

                    pq.offer(new Edge(dist[ed.v],ed.v));

                }

            }

        }

        for(int i=1;i<=n;i++)

        {

             max=Math.max(dist[i],max);

        }

        return (max==Integer.MAX\_VALUE)?-1:max;

    }

    public int networkDelayTime(int[][] times, int n, int k) {

        ArrayList<Edge>[] graph = new ArrayList[n+1];

        for(int i=0;i<=n;i++)

        {

            graph[i]=new ArrayList<>();

        }

        for(int[] edge:times)

        {

            int u=edge[0],v=edge[1],w=edge[2];

            graph[u].add(new Edge(w,v));

        }

        return dijkstras(graph,n,k);

    }

}

**15) Flood Fill (Leetcode - 733)**

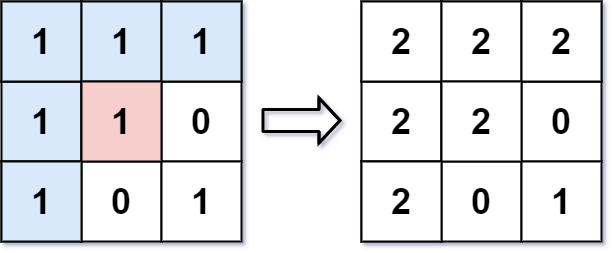
An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and color. You should perform a flood fill on the image starting from the pixel image[sr][sc].

To perform a flood fill, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with color.

Return the modified image after performing the flood fill.

Example 1:



Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

Explanation: From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

Example 2:

Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

Output: [[0,0,0],[0,0,0]]

Explanation: The starting pixel is already colored 0, so no changes are made to the image.

Constraints:

m == image.length

n == image[i].length

1 <= m, n <= 50

0 <= image[i][j], color < 216

0 <= sr < m

0 <= sc < n

**Solution :**

class Solution {

    public void dfs(int i,int j,int[][] image,int same,int color)

    {

        if(i>=image.length || i<0 || j<0 || j>=image[0].length || image[i][j]!=same)

        {

            return ;

        }

        //making visited

        image[i][j]=-1;

        dfs(i-1,j,image,same,color);

        dfs(i,j-1,image,same,color);

        dfs(i,j+1,image,same,color);

        dfs(i+1,j,image,same,color);

        //updating new color

        image[i][j]=color;

    }

    public int[][] floodFill(int[][] image, int sr, int sc, int color) {

        dfs(sr,sc,image,image[sr][sc],color);

        return image;

    }

}

**16) Rotting Oranges (Leetcode - 994)**

You are given an m x n grid where each cell can have one of three values:

0 representing an empty cell,

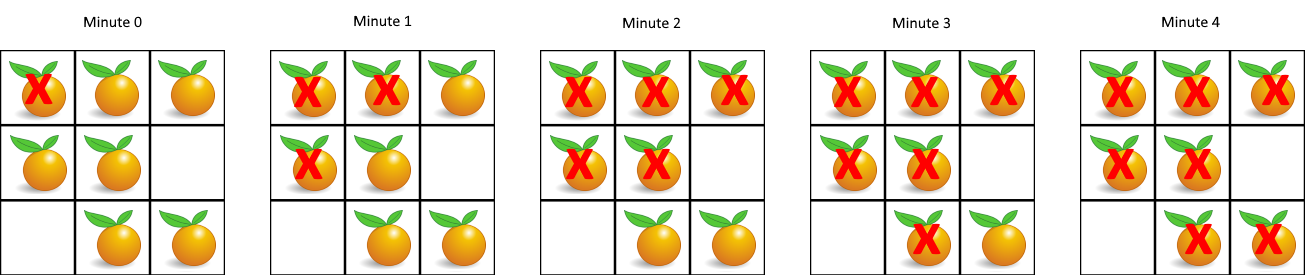
1 representing a fresh orange, or

2 representing a rotten orange.

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

Example 1:



Input: grid = [[2,1,1],[1,1,0],[0,1,1]]

Output: 4

Example 2:

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]

Output: -1

Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

Example 3:

Input: grid = [[0,2]]

Output: 0

Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 10

grid[i][j] is 0, 1, or 2.

**Solution:**

class Solution {

    public int BFS(int[][] grid)

    {

       return 0;

    }

    public int orangesRotting(int[][] grid) {

       Queue<int[]> queue = new LinkedList<>();

       int fresh = 0;

       for(int i=0;i<grid.length;i++)

       {

        for(int j=0;j<grid[0].length;j++)

        {

**//add all rotten oranges to queue to move all rotten oranges, one level at a time**

            if(grid[i][j]==2)

            {

                queue.offer(new int[]{i,j});

            }

            else if(grid[i][j]==1)

            {

                fresh++;

            }

        }

       }

       int min=0;

       //4 dirs

       int[][] dirs = { { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 } };

       while(!queue.isEmpty() && fresh!=0)

       {

        min++;

        int size = queue.size();

        for(int l=0;l<size;l++)

        {

            int[] cur = queue.poll();

            int i = cur[0],j=cur[1];

            for(int[] dir:dirs)

            {

                int  x=i+dir[0],y=j+dir[1];

                if(x>=0 && x<grid.length && y>=0 && y<grid[0].length && grid[x][y]==1)

                {

                    grid[x][y]=2;

                    queue.offer(new int[]{x,y});

                    fresh--;

                }

            }

        }

       }

        return (fresh==0)?min:-1;

    }

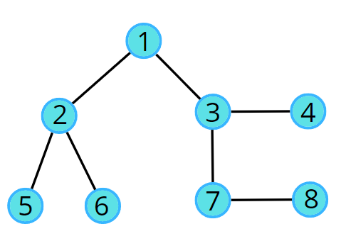
}

**17) Detect Cycle in an Undirected Graph**

**Example 1:**

**Input:**

V = 8, E = 7



**Output:** false

**Explanation:** No cycle in the given graph.

**Example 2:**

**Input:**

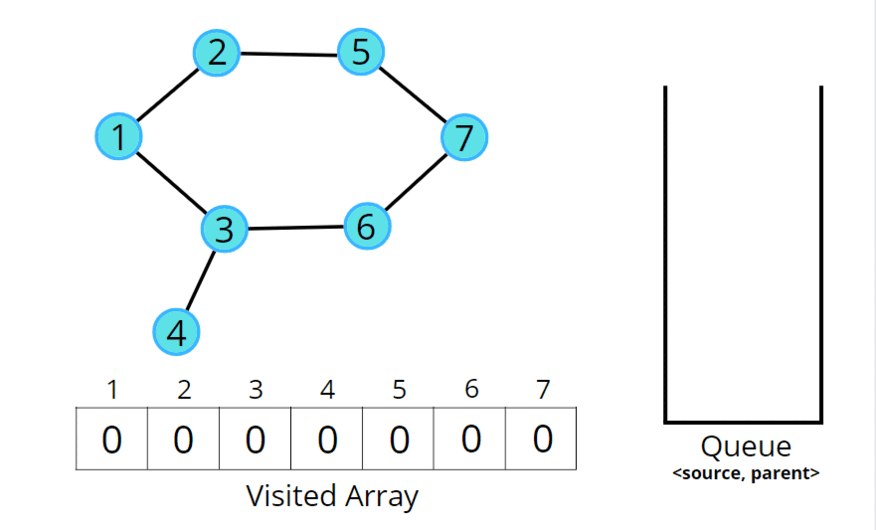
V = 8, E = 6



**Output:** true

**Explanation:** 4->5->6->4 is a cycle.

**Solution : (Using Bfs)**



static boolean checkForCycle(ArrayList<ArrayList<Integer>> adj, int s,boolean vis[])

{

Queue<Node> q = new LinkedList<>(); *//BFS*

q.add(new Node(s, -1));

vis[s] =true;

*// until the queue is empty*

while(!q.isEmpty())

{

*// source node and its parent node*

int node = q.peek().first;

int par = q.peek().second;

q.remove();

*// go to all the adjacent nodes*

for(Integer it: adj.get(node))

{

if(vis[it]==false)

{

q.add(new Node(it, node)); //it = adj node and node == parent

vis[it] = true;

}

*// if adjacent node is visited and is not its own parent node*

else if(par != it) return true;

}

}

return false;

}

**Using DFS: (approach same as Bfs)**

public boolean checkForCycle(int node, int parent, boolean vis[], ArrayList < ArrayList

< Integer >> adj) {

vis[node] = true;

for (Integer it: adj.get(node)) {

if (vis[it] == false) {

if (checkForCycle(it, node, vis, adj) == true)

return true;

} else if (it != parent)

return true;

}

return false;

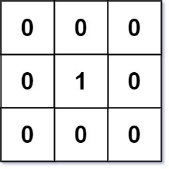
}

**18) 01 Matrix (Leetcode – 542)**

Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.

The distance between two adjacent cells is 1.

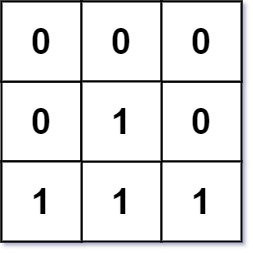
Example 1:



Input: mat = [[0,0,0],[0,1,0],[0,0,0]]

Output: [[0,0,0],[0,1,0],[0,0,0]]

Example 2:



Input: mat = [[0,0,0],[0,1,0],[1,1,1]]

Output: [[0,0,0],[0,1,0],[1,2,1]]

Constraints:

m == mat.length

n == mat[i].length

1 <= m, n <= 104

1 <= m \* n <= 104

mat[i][j] is either 0 or 1.

There is at least one 0 in mat.

**Solution : (Multi Source BFS)**

**//push all zeros to queue and traverse and increase distance 1**

class Solution {

    public int[][] updateMatrix(int[][] mat) {

        Queue<int[]> queue = new LinkedList<>();

        for(int i=0;i<mat.length;i++)

        {

            for(int j=0;j<mat[0].length;j++)

            {

                if(mat[i][j]==0)

                {

                    queue.add(new int[]{i,j});

                }

                else

                {

                    //for checking visited or not

                    mat[i][j]=-1;

                }

            }

        }

        while(!queue.isEmpty())

        {

            int[] cur = queue.poll();

            int i = cur[0],j=cur[1];

            int[][] dirs = {{0,-1},{-1,0},{0,1},{1,0}};

            for(int[] dir : dirs)

            {

                int x=i+dir[0];

                int y=j+dir[1];

                if(x>=0 && x<mat.length && y>=0 && y<mat[0].length && mat[x][y]==-1)

                {

                    mat[x][y]=mat[i][j]+1;//current node distance + 1

                    queue.add(new int[]{x,y});

                }

            }

        }

        return mat;

    }

}

**19) Surrounded Regions (Leetcode - 130)**

You are given an m x n matrix board containing letters 'X' and 'O', capture regions that are surrounded:

Connect: A cell is connected to adjacent cells horizontally or vertically.

Region: To form a region connect every 'O' cell.

Surround: The region is surrounded with 'X' cells if you can connect the region with 'X' cells and none of the region cells are on the edge of the board.

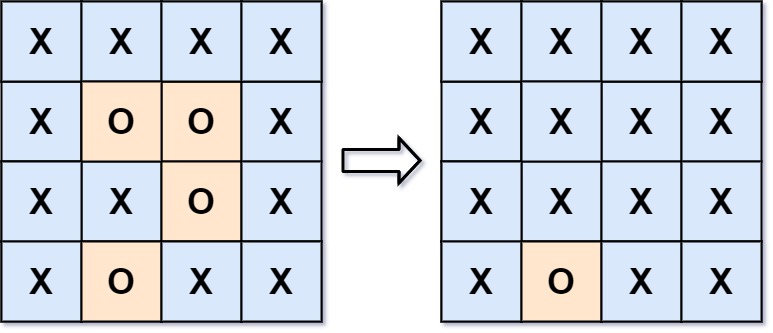
A surrounded region is captured by replacing all 'O's with 'X's in the input matrix board.

Example 1:

Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]

Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

Explanation:



In the above diagram, the bottom region is not captured because it is on the edge of the board and cannot be surrounded.

Example 2:

Input: board = [["X"]]

Output: [["X"]]

Constraints:

m == board.length

n == board[i].length

1 <= m, n <= 200

board[i][j] is 'X' or 'O'.

**Solution :**

class Solution {

    public void dfs(int i,int j,char[][] board)

    {

        board[i][j]='V';

        int[][] dirs={{0,-1},{-1,0},{1,0},{0,1}};

        for(int[] dir:dirs)

        {

            int x=dir[0]+i,y=dir[1]+j;

            if(x>=0 && x<board.length && y>=0 && y<board[0].length && board[x][y]=='O'){

                dfs(x,y,board);

            }

        }

    }

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

        for(int i=0;i<board.length;i++)

        {

            if(board[i][0]=='O')

            {

                dfs(i,0,board);

            }

            if(board[i][board[0].length-1]=='O')

            {

                dfs(i,board[0].length-1,board);

            }

        }

        for(int i=0;i<board[0].length;i++)

        {

            if(board[0][i]=='O')

            {

                dfs(0,i,board);

            }

            if(board[board.length-1][i]=='O')

            {

                dfs(board.length-1,i,board);

            }

        }

        for(int i=0;i<board.length;i++)

        {

            for(int j=0;j<board[0].length;j++)

            {

                if(board[i][j]=='O') board[i][j]='X';

                if(board[i][j]=='V') board[i][j]='O';

            }

        }

    }

}

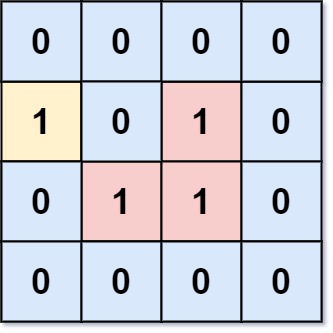
**20) Number of Enclaves (Leetcode - 1020)**

You are given an m x n binary matrix grid, where 0 represents a sea cell and 1 represents a land cell.

A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid.

Return the number of land cells in grid for which we cannot walk off the boundary of the grid in any number of moves.

Example 1:

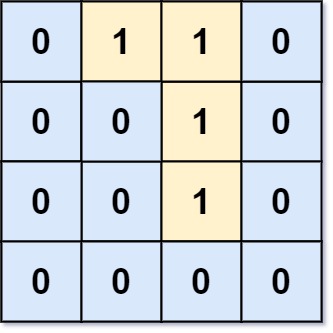


Input: grid = [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]

Output: 3

Explanation: There are three 1s that are enclosed by 0s, and one 1 that is not enclosed because its on the boundary.

Example 2:



Input: grid = [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]

Output: 0

Explanation: All 1s are either on the boundary or can reach the boundary.

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 500

grid[i][j] is either 0 or 1.

**Solution : (from borders if cell==1 call dfs )**

class Solution {

    public void dfs(int i,int j,int[][] board)

    {

        board[i][j]=-1;

        int[][] dirs={{0,-1},{-1,0},{1,0},{0,1}};

        for(int[] dir:dirs)

        {

            int x=dir[0]+i,y=dir[1]+j;

            if(x>=0 && x<board.length && y>=0 && y<board[0].length && board[x][y]==1){

                dfs(x,y,board);

            }

        }

    }

    public int numEnclaves(int[][] board) {

        for(int i=0;i<board.length;i++)

        {

            if(board[i][0]==1)

            {

                dfs(i,0,board);

            }

            if(board[i][board[0].length-1]==1)

            {

                dfs(i,board[0].length-1,board);

            }

        }

        for(int i=0;i<board[0].length;i++)

        {

            if(board[0][i]==1)

            {

                dfs(0,i,board);

            }

            if(board[board.length-1][i]==1)

            {

                dfs(board.length-1,i,board);

            }

        }

        int count=0;

        for(int i=0;i<board.length;i++)

        {

            for(int j=0;j<board[0].length;j++)

            {

                if(board[i][j]==1)

                {

                    count++;

                }

            }

        }

        return count;

    }

}

**21) Number of Distinct Islands (GFG)**

Given a boolean 2D matrix grid of size n \* m. You have to find the number of distinct islands where a group of connected 1s (horizontally or vertically) forms an island. Two islands are considered to be distinct if and only if one island is not equal to another (not rotated or reflected).

Example 1:

Input:

grid[][] = {{1, 1, 0, 0, 0},

{1, 1, 0, 0, 0},

{0, 0, 0, 1, 1},

{0, 0, 0, 1, 1}}

Output:

1

Explanation:

grid[][] = {{1, 1, 0, 0, 0},

{1, 1, 0, 0, 0},

{0, 0, 0, 1, 1},

{0, 0, 0, 1, 1}}

Same colored islands are equal.

We have 2 equal islands, so we

have only 1 distinct island.

Example 2:

Input:

grid[][] = {{1, 1, 0, 1, 1},

{1, 0, 0, 0, 0},

{0, 0, 0, 0, 1},

{1, 1, 0, 1, 1}}

Output:

3

Explanation:

grid[][] = {{1, 1, 0, 1, 1},

{1, 0, 0, 0, 0},

{0, 0, 0, 0, 1},

{1, 1, 0, 1, 1}}

Same colored islands are equal.

We have 4 islands, but 2 of them

are equal, So we have 3 distinct islands.

Your Task:

You don't need to read or print anything. Your task is to complete the function countDistinctIslands() which takes the grid as an input parameter and returns the total number of distinct islands.

Expected Time Complexity: O(n \* m)

Expected Space Complexity: O(n \* m)

Constraints:

1 ≤ n, m ≤ 500

grid[i][j] == 0 or grid[i][j] == 1

**Solution : (using set)**

class Solution {

public String dfs(int i,int j,int[][] board)

{

String s=Integer.toString(board[i][j]);

board[i][j]=-1;

int[][] dirs={{0,-1},{-1,0},{1,0},{0,1}};

for(int[] dir:dirs)

{

int x=dir[0]+i,y=dir[1]+j;

if(x>=0 && x<board.length && y>=0 && y<board[0].length && board[x][y]==1){

s+=dfs(x,y,board);

}

else

{

s+="0";

}

}

return s;

}

int countDistinctIslands(int[][] board) {

Set<String> set = new HashSet<>();

for(int i=0;i<board.length;i++)

{

for(int j=0;j<board[0].length;j++)

{

if(board[i][j]==1)

{

set.add(dfs(i,j,board));

}

}

}

return set.size();

}

}

**22) Is Graph Bipartite? (Leetcode - 785)**

There is an undirected graph with n nodes, where each node is numbered between 0 and n - 1. You are given a 2D array graph, where graph[u] is an array of nodes that node u is adjacent to. More formally, for each v in graph[u], there is an undirected edge between node u and node v. The graph has the following properties:

There are no self-edges (graph[u] does not contain u).

There are no parallel edges (graph[u] does not contain duplicate values).

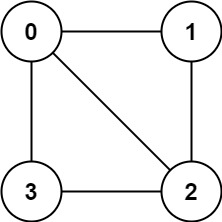
If v is in graph[u], then u is in graph[v] (the graph is undirected).

The graph may not be connected, meaning there may be two nodes u and v such that there is no path between them.

A graph is bipartite if the nodes can be partitioned into two independent sets A and B such that every edge in the graph connects a node in set A and a node in set B.

Return true if and only if it is bipartite.

Example 1:

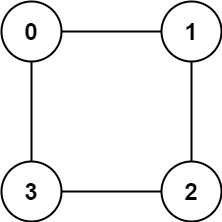


Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

Output: false

Explanation: There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

Example 2:



Input: graph = [[1,3],[0,2],[1,3],[0,2]]

Output: true

Explanation: We can partition the nodes into two sets: {0, 2} and {1, 3}.

Constraints:

graph.length == n

1 <= n <= 100

0 <= graph[u].length < n

0 <= graph[u][i] <= n - 1

graph[u] does not contain u.

All the values of graph[u] are unique.

If graph[u] contains v, then graph[v] contains u.

**Solution : (Graph Coluring) (BFS)**

class Solution {

    public boolean bfs(int[][] graph,int c,int[] color)

    {

        Queue<Integer> queue = new LinkedList<>();

        queue.offer(c);

        color[c]=0;

        while(!queue.isEmpty())

        {

            int cur = queue.poll();

            for(int i:graph[cur])

            {

// If the destination node has not been colored, color it with the opposite color

                if(color[i]==-1)

                {

                    queue.add(i);

                    color[i]=1 - color[cur];

                }

// If the destination node has the same color as the current node, the graph is not bipartite

                else if(color[i]==color[cur])

                {

                    return false;

                }

            }

        }

        return true;

    }

    public boolean isBipartite(int[][] graph) {

        int color[] = new int[graph.length];

        Arrays.fill(color,-1);

        for(int i=0;i<graph.length;i++)

        {

            // if there is more than one component i.e more than one graph

            // like [[1,3],[0,2],[1,3],[0,2],[5,6],[4,6],[4,5]]

            if(color[i]==-1)

            {

                if(!bfs(graph,i,color))

                {

                    return false;

                }

            }

        }

        return true;

    }

}

**Using DFS :**

    public boolean dfs(int[][] graph,int cur,int col,int[] color)

    {

        color[cur]=col;

        for(int i:graph[cur])

        {

            if(color[i]==-1)

            {

                if(!dfs(graph,i,1-col,color))

                {

                    return false;

                }

            }

            else if(color[i]==col)

            {

                return false;

            }

        }

        return true;

    }

    public boolean isBipartite(int[][] graph) {

        int color[] = new int[graph.length];

        Arrays.fill(color,-1);

        for(int i=0;i<graph.length;i++)

        {

            // if there is more than one component i.e more than one graph

            // like [[1,3],[0,2],[1,3],[0,2],[5,6],[4,6],[4,5]]

            if(color[i]==-1)

            {

                if(!dfs(graph,i,0,color))

                {

                    return false;

                }

            }

        }

        return true;

    }

**23) Course Schedule (Leetcode - 237)**

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]

Output: true

Explanation: There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

Example 2:

Input: numCourses = 2, prerequisites = [[1,0],[0,1]]

Output: false

Explanation: There are a total of 2 courses to take.

To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Constraints:

1 <= numCourses <= 2000

0 <= prerequisites.length <= 5000

prerequisites[i].length == 2

0 <= ai, bi < numCourses

All the pairs prerequisites[i] are unique.

**Solution :( Detect a cycle in directed graph)**

class Solution {

    public boolean dfs(ArrayList<Integer>[] graph,int cur,int[] vis,int[] pathVis)

    {

        vis[cur]=1;

        pathVis[cur]=1;

        for(int i:graph[cur])

        {

            if(vis[i]!=1)

            {

               if(!dfs(graph,i,vis,pathVis))

               {

                    return false;

               }

            }

            //if it is visited in vis and it is also visited in pathVis then it has cycle

            else if(pathVis[i]==1)

            {

                return false;

            }

        }

        pathVis[cur]=0;

        return true;

    }

    public boolean canFinish(int numCourses, int[][] prerequisites) {

        ArrayList<Integer>[] graph = new ArrayList[numCourses];

        for(int i=0;i<numCourses;i++)

        {

            graph[i]=new ArrayList<>();

        }

        //converting prequisites into directed graph

        for(int[] i:prerequisites)

        {

            int u=i[0],v=i[1];

            graph[u].add(v);

        }

        System.out.println(Arrays.toString(graph));

        int[] vis = new int[numCourses];

        int[] pathVis = new int[numCourses];

        for(int i=0;i<numCourses;i++)

        {

            if(vis[i]!=1)

            {

               if(!dfs(graph,i,vis,pathVis))

               {

                    return false;

               }

            }

        }

        return true;

    }

}

**24) Course Schedule II (Leetcode - 210)**

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]

Output: [0,1]

Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

Example 2:

Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

Output: [0,2,1,3]

Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

Example 3:

Input: numCourses = 1, prerequisites = []

Output: [0]

Constraints:

1 <= numCourses <= 2000

0 <= prerequisites.length <= numCourses \* (numCourses - 1)

prerequisites[i].length == 2

0 <= ai, bi < numCourses

ai != bi

All the pairs [ai, bi] are distinct.

**Solution : (Using Topological Sort)**

class Solution {

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

        ArrayList<Integer>[] graph = new ArrayList[numCourses];

        int[] inDegree = new int[numCourses];

        for(int i=0;i<numCourses;i++)

        {

            graph[i]=new ArrayList<Integer>();

        }

        //converting in to directed graph and calculating Indegree

        for(int[] i : prerequisites)

        {

            graph[i[1]].add(i[0]);

            inDegree[i[0]]++;

        }

        int[] res = new int[numCourses];

        Queue<Integer> queue = new LinkedList<>();

        for(int i=0;i<numCourses;i++)

        {

            if(inDegree[i]==0)

            {

                queue.offer(i);

            }

        }

        int ind=0;

        while(!queue.isEmpty())

        {

            int cur = queue.poll();

            res[ind]=cur;

            ind++;

            for(int i:graph[cur])

            {

                inDegree[i]--;

                if(inDegree[i]==0)

                {

                    queue.offer(i);

                }

            }

        }

        return (ind==numCourses)?res:new int[0];

    }

}

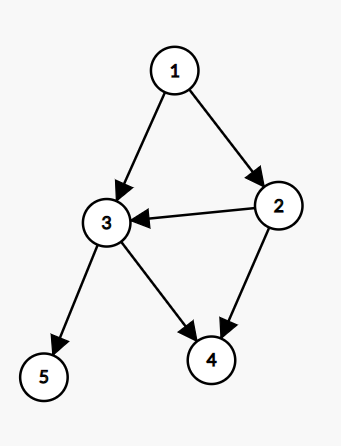
**25) Topological Sort Using DFS**

**Problem Statement**: Given a DAG( Directed Acyclic Graph ), print all the vertex of the graph in a topologically sorted order. If there are multiple solutions, print any.

**Pre-req**: DFS traversal, Graphs, Stack data structure.

**Example 1:**

**Input:**

****

**Output:**

One of the solutions is 1,2,3,5,4

**Solution :**

import java.util.\*;

class Solution {

static void findTopoSort(int node, int vis[], ArrayList<ArrayList<Integer>> adj, Stack<Integer> st) {

vis[node] = 1;

for(Integer it: adj.get(node)) {

if(vis[it] == 0) {

findTopoSort(it, vis, adj, st);

}

}

st.push(node);

}

static int[] topoSort(int N, ArrayList<ArrayList<Integer>> adj) {

Stack<Integer> st = new Stack<Integer>();

int vis[] = new int[N];

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

if(vis[i] == 0) {

findTopoSort(i, vis, adj, st);

}

}

int topo[] = new int[N];

int ind = 0;

while(!st.isEmpty()) {

topo[ind++] = st.pop();

}

*// for(int i = 0;i<N;i++) System.out.println(topo[i] + " ");*

return topo;

}

public static void main(String args[])

{

ArrayList<ArrayList<Integer>> adj=new ArrayList<>();

int n=6;

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

{

ArrayList<Integer> arr=new ArrayList<>();

adj.add(arr);

}

adj.get(5).add(2);

adj.get(5).add(0);

adj.get(4).add(0);

adj.get(4).add(1);

adj.get(2).add(3);

adj.get(3).add(1);

int res[] = topoSort(6, adj);

System.out.println("Toposort of the given graph is:" );

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

System.out.print(res[i]+" ");

}

}

}

**Detect a Cycle in Directed Graph | Topological Sort | Kahn's Algorithm**

* Since we know topological sorting is only possible for ***directed acyclic graphs(DAGs)*** if we apply Kahn’s algorithm in a directed cyclic graph(A directed graph that contains a cycle), it will fail to find the topological sorting(i.e. The final sorting will not contain all the nodes or vertices).
* So, finally, we will check the sorting to see if it contains all V vertices or not. If the result does not include all V vertices, we can conclude that there is a cycle.

**26) Alien Dictionary - Topological Sort: G-26**

**Problem Statement:** Given a sorted dictionary of an alien language having N words and k starting alphabets of a standard dictionary. Find the order of characters in the alien language.

**Note:** Many orders may be possible for a particular test case, thus you may return any valid order.

**Examples:**

**Example 1:**

**Input:** N = 5, K = 4

dict = {"baa","abcd","abca","cab","cad"}

**Output**: b d a c

**Explanation:**

We will analyze every consecutive pair to find out the order of the characters.

The pair “baa” and “abcd” suggests ‘b’ appears before ‘a’ in the alien dictionary.

The pair “abcd” and “abca” suggests ‘d’ appears before ‘a’ in the alien dictionary.

The pair “abca” and “cab” suggests ‘a’ appears before ‘c’ in the alien dictionary.

The pair “cab” and “cad” suggests ‘b’ appears before ‘d’ in the alien dictionary.

So, [‘b’, ‘d’, ‘a’, ‘c’] is a valid ordering.

**Example 2:**

**Input:** N = 3, K = 3

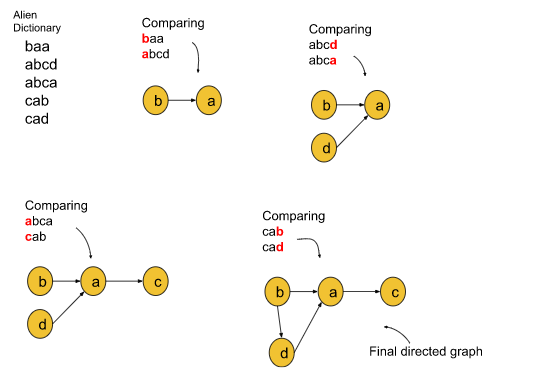
dict = {"caa","aaa","aab"}

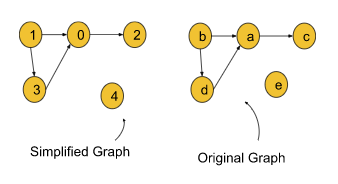
**Output**: c a b

**Explanation:** Similarly, if we analyze the consecutive pair

for this example, we will figure out [‘c’, ‘a’, ‘b’] is

a valid ordering.





**Solution:**

import java.util.\*;

class Solution {

private List<Integer> topoSort(int V, List<List<Integer>> adj) {

int indegree[] = new int[V];

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

for (int it : adj.get(i)) {

indegree[it]++;

}

}

Queue<Integer> q = new LinkedList<>();

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

if (indegree[i] == 0) {

q.add(i);

}

}

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

while (!q.isEmpty()) {

int node = q.peek();

q.remove();

topo.add(node);

*// node is in your topo sort*

*// so please remove it from the indegree*

for (int it : adj.get(node)) {

indegree[it]--;

if (indegree[it] == 0) q.add(it);

}

}

return topo;

}

public String findOrder(String [] dict, int N, int K) {

//converting to direct graph

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

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

adj.add(new ArrayList<>());

}

for (int i = 0; i < N - 1; i++) {

String s1 = dict[i];

String s2 = dict[i + 1];

int len = Math.min(s1.length(), s2.length());

for (int ptr = 0; ptr < len; ptr++) {

if (s1.charAt(ptr) != s2.charAt(ptr)) {

adj.get(s1.charAt(ptr) - 'a').add(s2.charAt(ptr) - 'a');

break;

}

}

}

List<Integer> topo = topoSort(K, adj);

String ans = "";

for (int it : topo) {

ans = ans + (char)(it + (int)('a'));

}

return ans;

}

}

public class tUf {

public static void main(String[] args) {

int N = 5, K = 4;

String[] dict = {"baa", "abcd", "abca", "cab", "cad"};

Solution obj = new Solution();

String ans = obj.findOrder(dict, N, K);

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

System.out.print(ans.charAt(i) + " ");

}

System.out.println("");

}

}

**27) Shortest path in Directed Acyclic Graph (GFG)**

Given a Directed Acyclic Graph of n vertices from 0 to n-1 and a 2D Integer array(or vector) edges[ ][ ] of length m, where there is a directed edge from edge[i][0] to edge[i][1] with a distance of edge[i][2] for all i.

Find the shortest path from src(0) vertex to all the vertices and if it is impossible to reach any vertex, then return -1 for that vertex.

Examples :

Input: n = 4, m = 2, edge = [[0,1,2],[0,2,1]]

Output: 0 2 1 -1

Explanation: Shortest path from 0 to 1 is 0->1 with edge weight 2. Shortest path from 0 to 2 is 0->2 with edge weight 1. There is no way we can reach 3, so it's -1 for 3.\

Input: n = 6, n = 7, edge = [[0,1,2],[0,4,1],[4,5,4],[4,2,2],[1,2,3],[2,3,6],[5,3,1]]

Output: 0 2 3 6 1 5

Explanation: Shortest path from 0 to 1 is 0->1 with edge weight 2. Shortest path from 0 to 2 is 0->4->2 with edge weight 1+2=3. Shortest path from 0 to 3 is 0->4->5->3 with edge weight 1+4+1=6. Shortest path from 0 to 4 is 0->4 with edge weight 1.Shortest path from 0 to 5 is 0->4->5 with edge weight 1+4=5.

Your Task:You don't need to print or input anything. Complete the function shortest path() which takes an integer N as number of vertices, an integer M as number of edges and a 2D Integer array(or vector) edges as the input parameters and returns an integer array(or vector), denoting the list of distance from src to all nodes.

Expected Time Complexity: O(n+m), where n is the number of nodes and m is edges

Expected Space Complexity: O(n)

Constraint:

1 <= n <= 100

1 <= m <= min((N\*(N-1))/2,4000)

0 <= edgei,0,edgei,1 < n

0 <= edgei,2 <=105

**Solution : (Using toposort)**

class Solution {

// Define a Pair class to store an edge and its corresponding weight

class Pair {

int edge, weight; // 'edge' is the destination node, 'weight' is the cost

Pair(int edge, int weight) {

this.edge = edge;

this.weight = weight;

}

}

// Function to perform topological sort using DFS

public void topoSort(int cur, Stack<Integer> stack, int[] vis, ArrayList<Pair>[] graph) {

vis[cur] = 1; // Mark the current node as visited

// Explore all neighbors (connected edges) of the current node

for (Pair i : graph[cur]) {

if (vis[i.edge] == 0) { // If the neighboring node is not visited, recursively visit it

topoSort(i.edge, stack, vis, graph);

}

}

// After all neighbors are processed, push the current node onto the stack

stack.push(cur);

}

// Main function to find the shortest path in a Directed Acyclic Graph (DAG)

public int[] shortestPath(int N, int M, int[][] edges) {

// Initialize the adjacency list representation of the graph

ArrayList<Pair>[] graph = new ArrayList[N];

int vis[] = new int[N]; // Visited array to mark nodes during topological sort

Stack<Integer> stack = new Stack<>(); // Stack to store topologically sorted nodes

// Initialize each node's adjacency list

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

graph[i] = new ArrayList<>();

}

// Populate the graph with edges from the input (directed edges with weights)

for (int[] i : edges) {

graph[i[0]].add(new Pair(i[1], i[2])); // Add edge with destination and weight

}

// Perform topological sort on all nodes

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

if (vis[i] == 0) { // If the node hasn't been visited yet

topoSort(i, stack, vis, graph); // Call topoSort to process the node

}

}

// Initialize the distance array with a large value (infinity) for all nodes

int dis[] = new int[N];

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

dis[i] = (int) (1e9); // 1e9 represents infinity (unreachable nodes)

}

dis[0] = 0; // Distance to the starting node (node 0) is set to 0

// Process nodes in topological order to find the shortest path

while (!stack.isEmpty()) {

int cur = stack.pop(); // Pop the next node from the stack (topological order)

// Relax all the edges of the current node

for (Pair pair : graph[cur]) {

if (dis[pair.edge] > dis[cur] + pair.weight) {

dis[pair.edge] = dis[cur] + pair.weight; // Update the shortest distance

}

}

}

// Replace distances of unreachable nodes (still marked as infinity) with -1

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

if (dis[i] == (1e9)) dis[i] = -1;

}

return dis; // Return the array of shortest distances

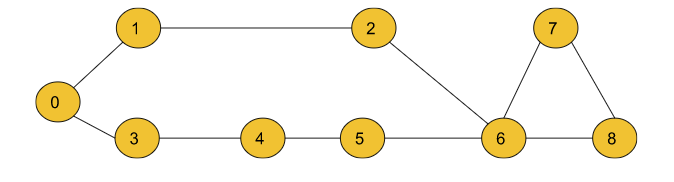
}

}

**27) Shortest Path in Undirected Graph with unit distance: G-28**

Given an Undirected Graph having **unit weight**, find the shortest path from the source to all other nodes in this graph. In this problem statement, we have assumed the source vertex to be ‘0’. If a vertex is unreachable from the source node, then return -1 for that vertex.

**Example 1:**



**Input**:

n = 9, m = 10

edges = [[0,1],[0,3],[3,4],[4 ,5],[5, 6],[1,2],[2,6],[6,7],[7,8],[6,8]]

src=0

**Output:** 0 1 2 1 2 3 3 4 4

**Explanation:**

The above output array shows the shortest path to all

the nodes from the source vertex (0), Dist[0] = 0,

Dist[1] = 1 , Dist[2] = 2 , …. Dist[8] = 4

Where Dist[node] is the shortest path between src and

the node. For a node, if the value of Dist[node]= -1,

then we conclude that the node is unreachable from

the src node.

**Solution :( Using only Queue(BFS) )**

class Solution {

public int[] shortestPath(int[][] edges,int n,int m ,int src) {

//Create an adjacency list of size N for storing the undirected graph.

ArrayList<ArrayList<Integer>> adj = new ArrayList<>();

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

adj.add(new ArrayList<>());

}

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

adj.get(edges[i][0]).add(edges[i][1]);

adj.get(edges[i][1]).add(edges[i][0]);

}

//A dist array of size N initialised with a large number to

//indicate that initially all the nodes are untraversed.

int dist[] = new int[n];

for(int i = 0;i<n;i++) dist[i] = (int)1e9;

dist[src] = 0;

// BFS Implementation

Queue<Integer> q = new LinkedList<>();

q.add(src);

while(!q.isEmpty()) {

int node = q.peek();

q.remove();

for(int it : adj.get(node)) {

if(dist[node] + 1 < dist[it]) {

dist[it] = 1 + dist[node];

q.add(it);

}

}

}

// Updated shortest distances are stored in the resultant array ‘ans’.

// Unreachable nodes are marked as -1.

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

if(dist[i] == 1e9) {

dist[i] = -1;

}

}

return dist;

}

}

**28) Word Ladder (Leetcode - 127)**

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.

Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.

sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return the number of words in the shortest transformation sequence from beginWord to endWord, or 0 if no such sequence exists.

Example 1:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

Example 2:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:

1 <= beginWord.length <= 10

endWord.length == beginWord.length

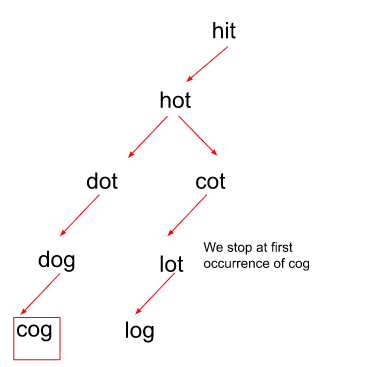
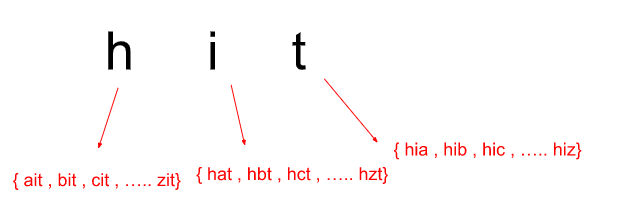
1 <= wordList.length <= 5000

wordList[i].length == beginWord.length

beginWord, endWord, and wordList[i] consist of lowercase English letters.

beginWord != endWord

All the words in wordList are unique.



**Solution :**

class Solution {

    class Pair

    {

        String word;

        int level;

        Pair(String word,int level)

        {

            this.word=word;

            this.level=level;

        }

    }

    public int ladderLength(String beginWord, String endWord, List<String> wordList) {

        Queue<Pair> queue = new LinkedList<>();

        queue.add(new Pair(beginWord,1));

        Set<String> set = new HashSet(wordList);

        while(!queue.isEmpty())

        {

            Pair cur = queue.poll();

            if(cur.word.equals(endWord)) return cur.level;

            for(int i=0;i<cur.word.length();i++)

            {

                for(char ch='a';ch<='z';ch++)

                {

                    char replace[] =  cur.word.toCharArray();

                    replace[i]=ch;

                    String replaceWord = new String(replace);

                    if(set.contains(replaceWord))

                    {

                        set.remove(replaceWord);

                        queue.add(new Pair(replaceWord,cur.level+1));

                    }

                }

            }

        }

        return 0;

    }

}

**30) Word Ladder II (Leetcode - 126)**

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.

Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.

sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

Example 1:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

Explanation: There are 2 shortest transformation sequences:

"hit" -> "hot" -> "dot" -> "dog" -> "cog"

"hit" -> "hot" -> "lot" -> "log" -> "cog"

Example 2:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

Output: []

Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:

1 <= beginWord.length <= 5

endWord.length == beginWord.length

1 <= wordList.length <= 500

wordList[i].length == beginWord.length

beginWord, endWord, and wordList[i] consist of lowercase English letters.

beginWord != endWord

All the words in wordList are unique.

The sum of all shortest transformation sequences does not exceed 10^5.



Each level may have multiple paths having same (like pot).so we using level

**Solution :( visit video striver graph 31 for another approach)**

class Solution {

    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

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

        Queue<List<String>> queue = new LinkedList<>();

        List<String> list = new ArrayList<>();

        list.add(beginWord);

        queue.add(list);

        Set<String> set = new HashSet(wordList);

        if (!set.contains(endWord)) {

            return res; // Early exit if endWord is not in wordList

        }

        boolean found=false;

        while(!queue.isEmpty() && !found)

        {

            int size = queue.size();//level by level because if endWord is found then it is last level check only that level

//store words in current level and after the level we remove so “pot” removed from set after the level completion.

            Set<String> wordsInThisLevel = new HashSet<>();

            for(int level=0;level<size;level++)

            {

                List<String> cur = queue.poll();

                String word=cur.get(cur.size()-1);

                for(int i=0;i<word.length();i++)

                {

                    for(char ch='a';ch<='z';ch++)

                    {

                        char replace[] =  word.toCharArray();

                        replace[i]=ch;

                        String replaceWord = new String(replace);

                        if(replaceWord.equals(endWord))

                        {

                            List<String> newList = new ArrayList<String>(cur);

                            newList.add(endWord);

                            res.add(newList);

                            found=true;

                        }

                       else if(set.contains(replaceWord))

                        {

                            wordsInThisLevel.add(replaceWord);

                            List<String> newList = new ArrayList<String>(cur);

                            newList.add(replaceWord);

                            queue.add(newList);

                        }

                    }

                }

            }

            set.removeAll(wordsInThisLevel);

        }

        return res;

    }

}

//The time complexity of the Word Ladder-II algorithm is not predictable because the number of transformations needed to go from one word to another can vary depending on the example. This means that there is no fixed range of time or space in which the program will run for all test cases.

**31) Print Shortest Path - Dijkstra’s Algorithm**

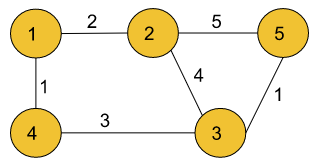
**Problem Statement**:

You are given a weighted undirected graph having **n+1** vertices numbered from 0 to n and **m** edges describing there are edges between a to b with some weight, find the shortest path between the vertex 1 and the vertex n, and if the path does not exist then return a list consisting of only**-1**.

**Note:** Please read the [G-32](https://takeuforward.org/data-structure/dijkstras-algorithm-using-priority-queue-g-32/) and the [G-33](https://takeuforward.org/data-structure/dijkstras-algorithm-using-set-g-33/) article before reading this article to get a clear understanding of Dijkstra’s Algorithm will form the base for this particular problem.

**Examples:**

**Example 1:**



**Input:**

n = 5, m= 6

edges = [[1,2,2], [2,5,5], [2,3,4], [1,4,1],[4,3,3],[3,5,1]]

**Output:**

1 4 3 5

**Explanation:**

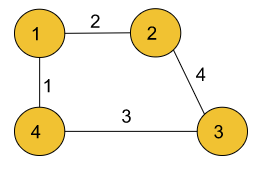
The source vertex is 1. Hence, the shortest distance path

of node 5 from the source will be 1->4->3->5 as this is

the path with a minimum sum of edge weights from source

to destination.

**Example 2:**



**Input:**

V = 4, E = 4

edges = [[1,2,2], [2,3,4], [1,4,1],[4,3,3]]

**Output:** 1 4

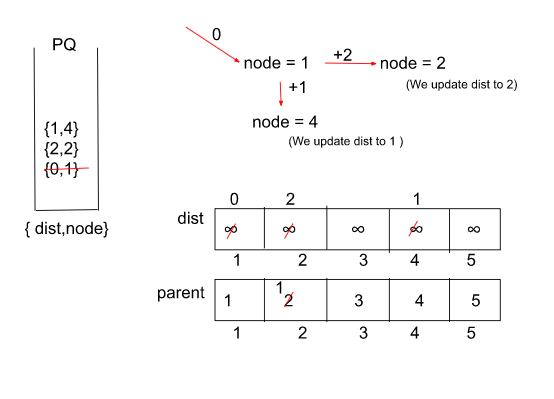
**Explanation:**

The source vertex is 1. Hence, the shortest distance

path of node 4 from the source will be 1->4 as this is

the path with the minimum sum of edge weights from

source to destination.



**Solution :**

import java.util.\*;

class Pair{

int first;

int second;

public Pair(int first,int second){

this.first = first;

this.second = second;

}

}

class Solution {

public static List<Integer> shortestPath(int n, int m, int edges[][]) {

*// Create an adjacency list of pairs of the form node1 -> {node2, edge weight}*

*// where the edge weight is the weight of the edge from node1 to node2.*

ArrayList<ArrayList<Pair>> adj = new ArrayList<>();

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

adj.add(new ArrayList<>());

}

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

adj.get(edges[i][0]).add(new Pair(edges[i][1], edges[i][2]));

adj.get(edges[i][1]).add(new Pair(edges[i][0], edges[i][2]));

}

*// Create a priority queue for storing the nodes along with distances*

*// in the form of a pair { dist, node }.*

PriorityQueue<Pair> pq =

new PriorityQueue<Pair>((x,y) -> x.first - y.first);

*// Create a dist array for storing the updated distances and a parent array*

*//for storing the nodes from where the current nodes represented by indices of*

*// the parent array came from.*

int[] dist = new int[n+1];

int[] parent =new int[n+1];

for(int i = 1;i<=n;i++) {

dist[i] = (int)(1e9);

parent[i] = i;

}

dist[1] = 0;

*// Push the source node to the queue.*

pq.add(new Pair(0, 1));

while(pq.size() != 0) {

*// Topmost element of the priority queue is with minimum distance value.*

Pair it = pq.peek();

int node = it.second;

int dis = it.first;

pq.remove();

*// Iterate through the adjacent nodes of the current popped node.*

for(Pair iter : adj.get(node)) {

int adjNode = iter.first;

int edW = iter.second;

*// Check if the previously stored distance value is*

*// greater than the current computed value or not,*

*// if yes then update the distance value.*

if(dis + edW < dist[adjNode]) {

dist[adjNode] = dis + edW;

pq.add(new Pair(dis + edW, adjNode));

*// Update the parent of the adjNode to the recent*

*// node where it came from.*

parent[adjNode] = node;

}

}

}

*// Store the final path in the ‘path’ array.*

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

*// If distance to a node could not be found, return an array containing -1.*

if(dist[n] == 1e9) {

path.add(-1);

return path;

}

int node = n;

*// o(N)*

while(parent[node] != node) {

path.add(node);

node = parent[node];

}

path.add(1);

*// Since the path stored is in a reverse order, we reverse the array*

*// to get the final answer and then return the array.*

Collections.reverse(path);

return path;

}

}

class tuf {

public static void main(String[] args) {

int V = 5, E = 6;

int[][] edges = {{1,2,2},{2,5,5},{2,3,4},{1,4,1},{4,3,3},{3,5,1}};

Solution obj = new Solution();

List < Integer > path = obj.shortestPath(V, E, edges);

for (int i = 0; i < path.size(); i++) {

System.out.print(path.get(i) + " ");

}

System.out.println();

}

}

**32) Shortest Path in Binary Matrix (Leetcode - 1091)**

Given an n x n binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

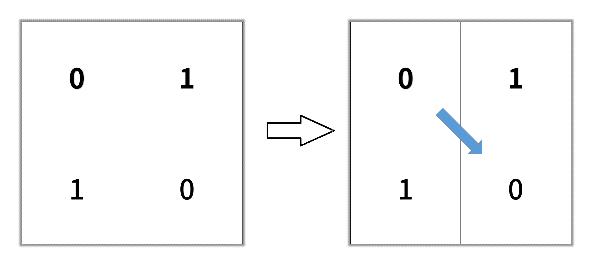
A clear path in a binary matrix is a path from the top-left cell (i.e., (0, 0)) to the bottom-right cell (i.e., (n - 1, n - 1)) such that:

All the visited cells of the path are 0.

All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).

The length of a clear path is the number of visited cells of this path.

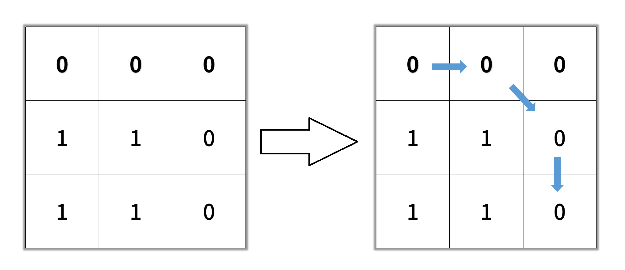
Example 1:



Input: grid = [[0,1],[1,0]]

Output: 2

Example 2:



Input: grid = [[0,0,0],[1,1,0],[1,1,0]]

Output: 4

Example 3:

Input: grid = [[1,0,0],[1,1,0],[1,1,0]]

Output: -1

Constraints:

n == grid.length

n == grid[i].length

1 <= n <= 100

grid[i][j] is 0 or 1

**Solution : (it also solvable using dp but it need to check 8 directions so BFS is good)**class Solution {

    class Pair

    {

        int x;

        int y;

        int level;

        Pair(int x,int y,int level)

        {

            this.x=x;

            this.y=y;

            this.level=level;

        }

    }

    public int shortestPathBinaryMatrix(int[][] grid) {

        int n = grid.length;

        int[][] directions = {

        {1, 0},   // Down

        {0, 1},   // Right

        {1, 1},   // Down-Right

        {-1, 0},  // Up

        {0, -1},  // Left

        {-1, -1}, // Up-Left

        {1, -1},  // Down-Left

        {-1, 1}   // Up-Right

        };

        Queue<Pair> queue = new LinkedList<>();

        if(grid[0][0]==1) return -1;

        queue.add(new Pair(0,0,1));

        while(!queue.isEmpty())

        {

            Pair cur = queue.poll();

            if(cur.x==n-1 && cur.y==n-1) return cur.level;

            for(int[] i : directions)

            {

                int x = cur.x+i[0];

                int y = cur.y+i[1];

                if(x>=0 && x<n && y>=0 && y<n && grid[x][y]==0)

                {

                    queue.offer(new Pair(x,y,cur.level+1));

                    grid[x][y]=-1;

                }

            }

        }

        return -1;

    }

}

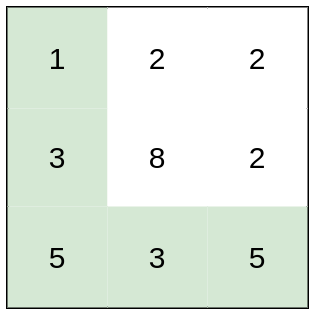
**33) Path With Minimum Effort (Leetcode - 1631)**

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., 0-indexed). You can move up, down, left, or right, and you wish to find a route that requires the minimum effort.

A route's effort is the maximum absolute difference in heights between two consecutive cells of the route.

Return the minimum effort required to travel from the top-left cell to the bottom-right cell.

Example 1:



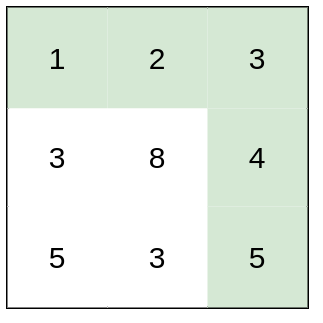
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]

Output: 2

Explanation: The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.

This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

Example 2:

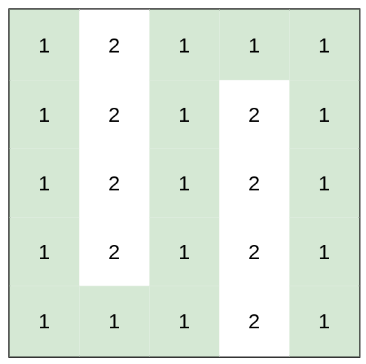


Input: heights = [[1,2,3],[3,8,4],[5,3,5]]

Output: 1

Explanation: The route of [1,2,3,4,5] has a maximum absolute difference of 1 in consecutive cells, which is better than route [1,3,5,3,5].

Example 3:



Input: heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]

Output: 0

Explanation: This route does not require any effort.

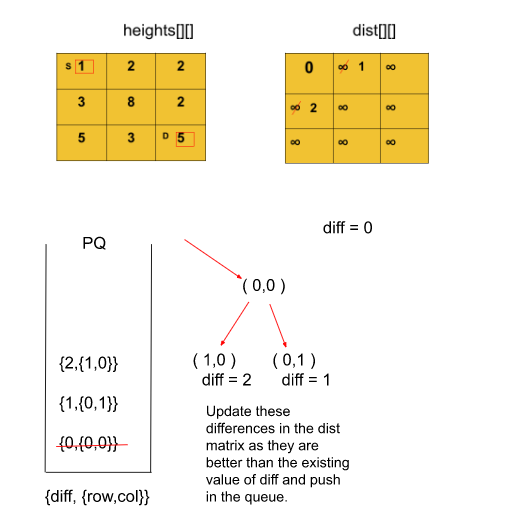
Constraints:

rows == heights.length

columns == heights[i].length

1 <= rows, columns <= 100

1 <= heights[i][j] <= 106



**Solution :**

class Solution {

    class Pair

    {

        int x;

        int y;

        int effort;

        Pair(int x,int y,int effort)

        {

            this.x=x;

            this.y=y;

            this.effort=effort;

        }

    }

    public int minimumEffortPath(int[][] heights) {

        int[][] effort = new int[heights.length][heights[0].length];

        for(int i[]:effort)

        {

            Arrays.fill(i,Integer.MAX\_VALUE);

        }

        effort[0][0]=0;

        PriorityQueue<Pair> pq = new PriorityQueue<>((a,b)->a.effort-b.effort);

        pq.offer(new Pair(0,0,0));

        int[][] directions = {{0,1},{0,-1},{-1,0},{1,0}};

        int max=Integer.MIN\_VALUE;

        while(!pq.isEmpty())

        {

            Pair cur = pq.poll();

             max=Math.max(cur.effort,max);

            if(cur.x== heights.length-1 && cur.y==heights[0].length-1) return max;

            for(int i[]:directions)

            {

                int x = cur.x+i[0];

                int y = cur.y+i[1];

                if(x>=0 && x<heights.length && y>=0 && y<heights[0].length && effort[x][y]>Math.abs(heights[cur.x][cur.y]-heights[x][y]))

                {

                    effort[x][y]=Math.abs(heights[cur.x][cur.y]-heights[x][y]);

                    pq.offer(new Pair(x,y,effort[x][y]));

                }

            }

        }

        return max;

    }

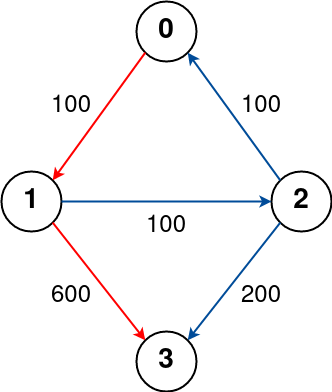
}

**34) Cheapest Flights Within K Stops (Leetcode - 787)**

There are n cities connected by some number of flights. You are given an array flights where flights[i] = [fromi, toi, pricei] indicates that there is a flight from city fromi to city toi with cost pricei.

You are also given three integers src, dst, and k, return the cheapest price from src to dst with at most k stops. If there is no such route, return -1.

Example 1:



Input: n = 4, flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], src = 0, dst = 3, k = 1

Output: 700

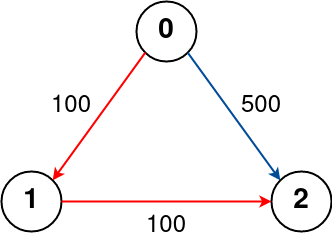
Explanation:

The graph is shown above.

The optimal path with at most 1 stop from city 0 to 3 is marked in red and has cost 100 + 600 = 700.

Note that the path through cities [0,1,2,3] is cheaper but is invalid because it uses 2 stops.

Example 2:



Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1

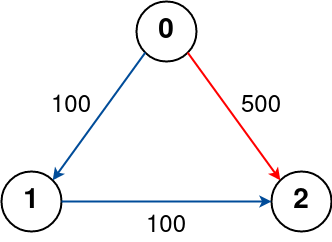
Output: 200

Explanation:

The graph is shown above.

The optimal path with at most 1 stop from city 0 to 2 is marked in red and has cost 100 + 100 = 200.

Example 3:



Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 0

Output: 500

Explanation:

The graph is shown above.

The optimal path with no stops from city 0 to 2 is marked in red and has cost 500.

Constraints:

1 <= n <= 100

0 <= flights.length <= (n \* (n - 1) / 2)

flights[i].length == 3

0 <= fromi, toi < n

fromi != toi

1 <= pricei <= 104

There will not be any multiple flights between two cities.

0 <= src, dst, k < n

src != dst

ex :

 dst = 2,k=2;

Cosider this to know why we using dist in step class also we have dist array.

Dist[4]=5 //0-3-1-4 but it completes k but doesn’t reach dest.so we don’t consider distance of dist array in calculating new distance.

For calculating new distance we use the current distance i.e step.dist.

Considering dist in step , 0-1-4-2 to reach dest using k. we consider step.dist of 4 have 6 in this path.But in the dist[4] array it has 5.

//dist array contains only minimum distance but we need dist of each path.

**Solution :**

import java.util.\*;

class Solution {

    class Pair {

        int node;

        int weight;

        Pair(int node, int weight) {

            this.node = node;

            this.weight = weight;

        }

    }

    class Step {

        int step;

        int node;

        int dist;

        Step(int step, int node,int dist) {

            this.step = step;

            this.node = node;

            this.dist=dist;

        }

    }

    public int bfs(int k, int src, int dst, ArrayList<Pair>[] graph, int n) {

        Queue<Step> queue = new LinkedList<>();

        int[] dist = new int[n];

        Arrays.fill(dist, Integer.MAX\_VALUE);

        dist[src] = 0;

        queue.offer(new Step(0, src,0));

        while (!queue.isEmpty()) {

            Step cur = queue.poll();

            // If we've exceeded the allowed number of steps, skip processing

            if (cur.step > k) {

                continue;

            }

            for (Pair p : graph[cur.node]) {

                // Calculate the new distance to this node

                int newDist = cur.dist + p.weight;

                // If the new distance is cheaper, update and add to the queue

                if (newDist < dist[p.node] && cur.step<=k) {

                    dist[p.node] = newDist;

                    queue.offer(new Step(cur.step + 1, p.node,dist[p.node]));

                }

            }

        }

        return dist[dst] == Integer.MAX\_VALUE ? -1 : dist[dst];

    }

    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {

        // Convert to graph

        ArrayList<Pair>[] graph = new ArrayList[n];

        for (int i = 0; i < n; i++) {

            graph[i] = new ArrayList<>();

        }

        for (int[] flight : flights) {

            graph[flight[0]].add(new Pair(flight[1], flight[2]));

        }

        return bfs(k, src, dst, graph, n);

    }

}

**35) Minimum Multiplications to reach End (GFG)**

Given start, end and an array arr of n numbers. At each step, start is multiplied with any number in the array and then mod operation with 100000 is done to get the new start.

Your task is to find the minimum steps in which end can be achieved starting from start. If it is not possible to reach end, then return -1.

Example 1:

Input:

arr[] = {2, 5, 7}

start = 3, end = 30

Output:

2

Explanation:

Step 1: 3\*2 = 6 % 100000 = 6

Step 2: 6\*5 = 30 % 100000 = 30

Example 2:

Input:

arr[] = {3, 4, 65}

start = 7, end = 66175

Output:

4

Explanation:

Step 1: 7\*3 = 21 % 100000 = 21

Step 2: 21\*3 = 63 % 100000 = 63

Step 3: 63\*65 = 4095 % 100000 = 4095

Step 4: 4095\*65 = 266175 % 100000 = 66175

Your Task:

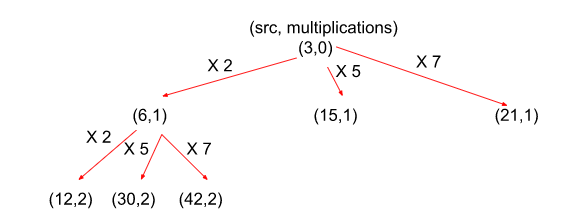
You don't need to print or input anything. Complete the function minimumMultiplications() which takes an integer array arr, an integer start and an integer end as the input parameters and returns an integer, denoting the minumum steps to reach in which end can be achieved starting from start.

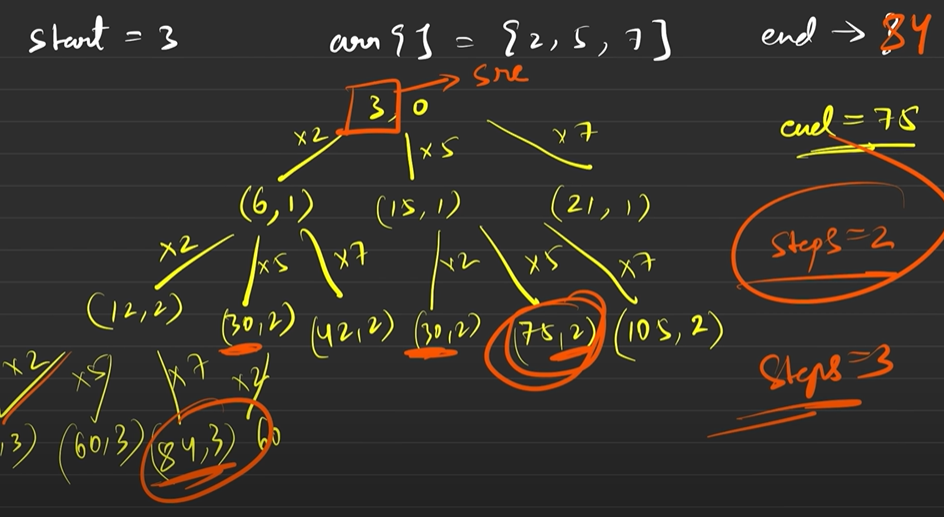
Constraints:

1 <= n <= 104

1 <= arr[i] <= 104

1 <= start, end < 10^5





**Solution : (This can be solved with out dist array only queue but it calculate two 30’s differently )**

**Using Dijkstra, in above 30 is twice but it only calculated once.**

**Dijkstra’s approach Solution:**

class Solution {

class Pair {

int first, second;

Pair(int first, int second) {

this.first = first;

this.second = second;

}

}

int minimumMultiplications(int[] arr, int start, int end) {

// Create a queue for storing the numbers as a result of multiplication

// of the numbers in the array and the start number.

Queue<Pair> q = new LinkedList<>();

q.add(new Pair(start, 0));

// Create a dist array to store the no. of multiplications to reach

// a particular number from the start number.

int[] dist = new int[100000];

for(int i = 0;i<100000;i++) dist[i] = (int)(1e9);

dist[start] = 0;

int mod = 100000;

int n = arr.length;

if(start==end) return 0;

// O(100000 \* N)

// Multiply the start no. with each of numbers in the arr

// until we get the end no.

while(!q.isEmpty()) {

int node = q.peek().first;

int steps = q.peek().second;

q.remove();

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

int num = (arr[i] \* node) % mod;

// If the no. of multiplications are less than before

// in order to reach a number, we update the dist array.

if(steps + 1 < dist[num]) {

dist[num] = steps + 1;

// Whenever we reach the end number

// return the calculated steps

if(num == end) return steps + 1;

q.add(new Pair(num, steps + 1));

}

}

}

// If the end no. is unattainable.

return -1;

}

}

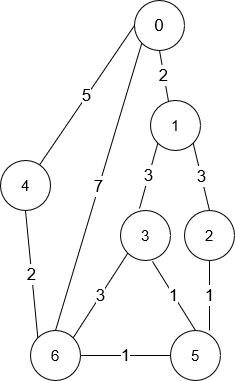
**36) Number of Ways to Arrive at Destination (Leetcode - 1976)**

You are in a city that consists of n intersections numbered from 0 to n - 1 with bi-directional roads between some intersections. The inputs are generated such that you can reach any intersection from any other intersection and that there is at most one road between any two intersections.

You are given an integer n and a 2D integer array roads where roads[i] = [ui, vi, timei] means that there is a road between intersections ui and vi that takes timei minutes to travel. You want to know in how many ways you can travel from intersection 0 to intersection n - 1 in the shortest amount of time.

Return the number of ways you can arrive at your destination in the shortest amount of time. Since the answer may be large, return it modulo 109 + 7.

Example 1:



Input: n = 7, roads = [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],[3,5,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]

Output: 4

Explanation: The shortest amount of time it takes to go from intersection 0 to intersection 6 is 7 minutes.

The four ways to get there in 7 minutes are:

- 0 ➝ 6

- 0 ➝ 4 ➝ 6

- 0 ➝ 1 ➝ 2 ➝ 5 ➝ 6

- 0 ➝ 1 ➝ 3 ➝ 5 ➝ 6

Example 2:

Input: n = 2, roads = [[1,0,10]]

Output: 1

Explanation: There is only one way to go from intersection 0 to intersection 1, and it takes 10 minutes.

Constraints:

1 <= n <= 200

n - 1 <= roads.length <= n \* (n - 1) / 2

roads[i].length == 3

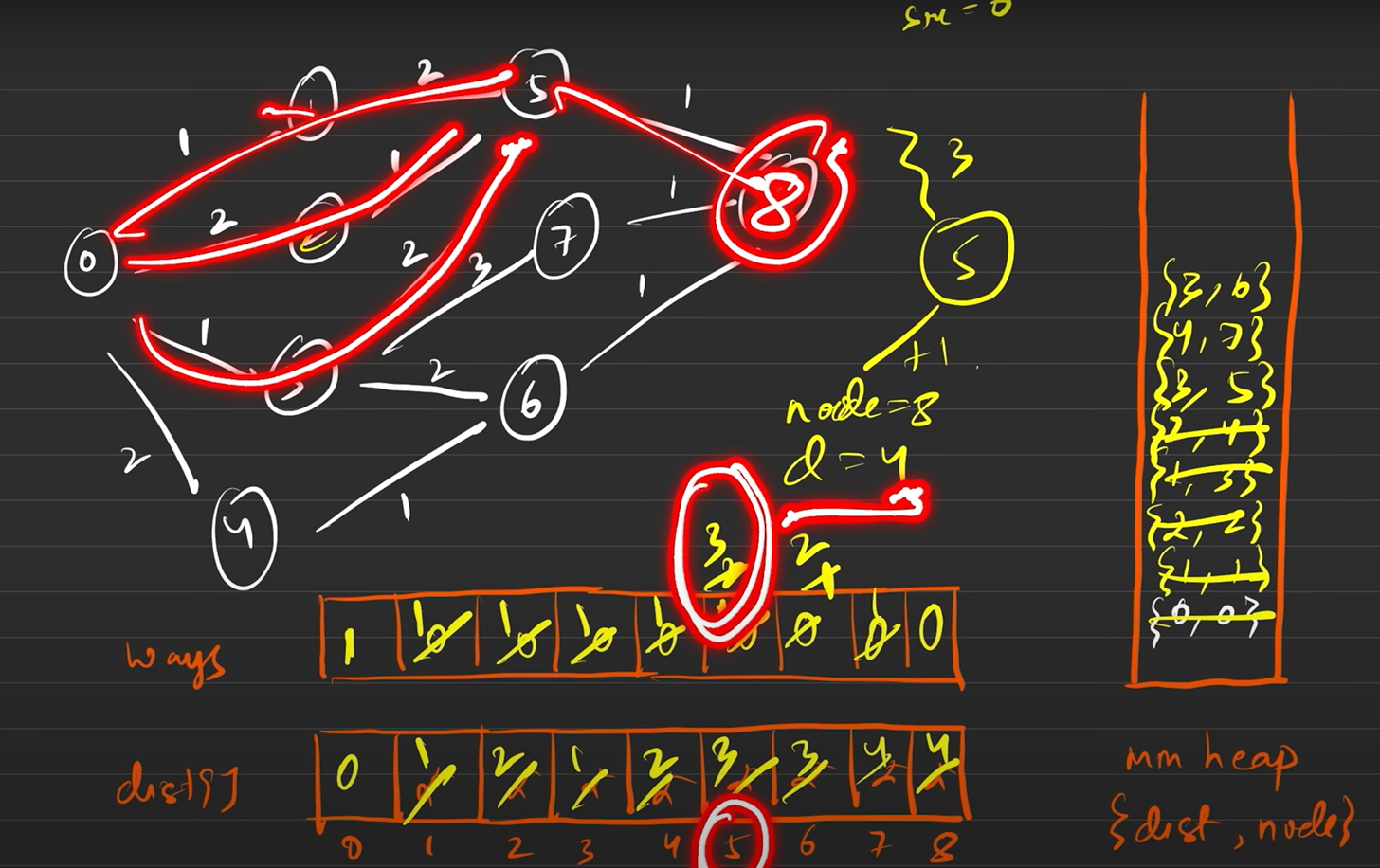
0 <= ui, vi <= n - 1

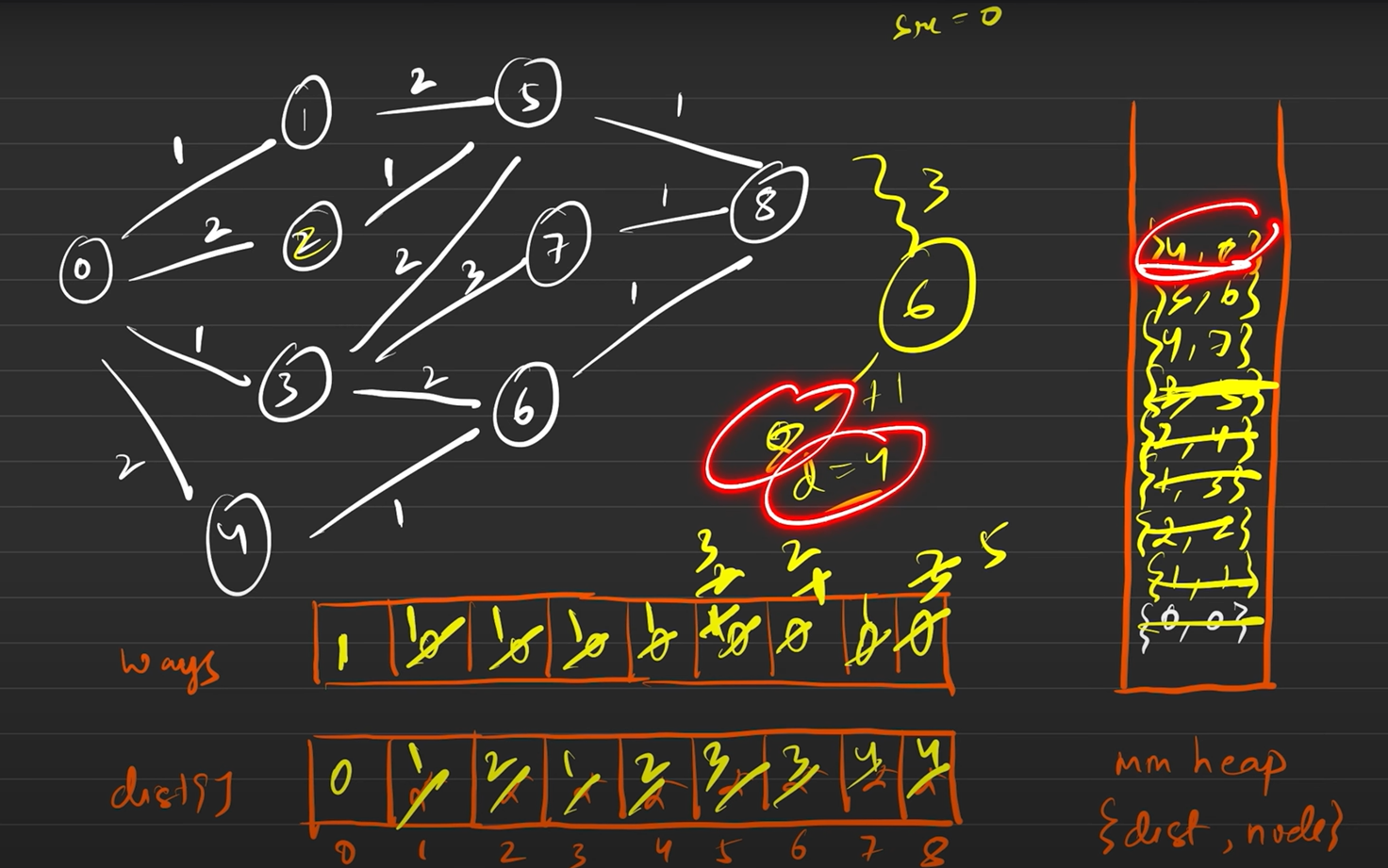
1 <= timei <= 109

ui != vi

There is at most one road connecting any two intersections.

You can reach any intersection from any other intersection.

****

****

**Solution :**

class Solution {

    class Pair

    {

        long dist;

        int node;

        Pair(long dist,int node)

        {

            this.dist=dist;

            this.node=node;

        }

        public String toString()

        {

            return this.dist+" "+this.node;

        }

    }

    class Node

    {

        int node;

        int weight;

        Node(int node,int weight)

        {

            this.node=node;

            this.weight=weight;

        }

    }

    public int dijkstra(ArrayList<Node>[] graph,int n)

    {

        PriorityQueue<Pair> queue = new PriorityQueue<>((a,b)-> Long.compare(a.dist, b.dist));

        queue.add(new Pair(0,0));

        long[] dist = new long[n];

        Arrays.fill(dist,Long.MAX\_VALUE);

        dist[0]=0;

        int[] ways = new int[n];

        ways[0]=1;

        while(!queue.isEmpty())

        {

            Pair cur = queue.poll();

            for(Node i : graph[cur.node])

            {

                if(dist[i.node]>cur.dist+i.weight)

                {

                    dist[i.node]=cur.dist+i.weight;

                    queue.offer(new Pair(dist[i.node],i.node));

//new then ways[i.node]=ways[cur.node]

                    ways[i.node]=ways[cur.node];

                }

// if same then ways = ways[i.node]+ways[cur.node]

                else if(dist[i.node]==cur.dist+i.weight)

                {

                    ways[i.node]=(ways[i.node]+ways[cur.node]);

                }

            }

        }

        return ways[n-1]%(int)(1e9+7);

    }

    public int countPaths(int n, int[][] roads) {

        //convert to graph

        ArrayList<Node>[] graph = new ArrayList[n];

        for(int i=0;i<n;i++)

        {

            graph[i]=new ArrayList<>();

        }

        for(int i[]:roads)

        {

            graph[i[0]].add(new Node(i[1],i[2]));

            graph[i[1]].add(new Node(i[0],i[2]));

        }

        return dijkstra(graph, n);

    }

}

**37) Bellman-Ford Algorithm - Shortest Distance with Negative Edge**

Problem Statement: Given a weighted directed graph with negative edge weights with n nodes and m edges. Nodes are labeled from 0 to n-1, the task is to find the shortest distance from the source node to all other nodes. Output “-1” if there exists a negative edge weight cycle in the graph.

Note: edges[i] is defined as u, v and weight.

Example 1:

Input:

Edges : [[0,1,5],[1,5,-3],[1,2,-2],[2,4,3],[3,2,6],[3,4,-2],[5,3,1]]

Output: [0,5,3,3,1,2]

Explanation: The shortest distance is calculated pretty much like the Djisktra’s Algorithm except that we have negative edge weights year.

For node pair 0-4 the possible paths were 0->1->2->4(cost = 6) and 0->1->5->3->4(cost = 1). We choose the minimum of these two of cost = 1. In the same manner, the costs for other nodes are calculated.

Note that since no negative edge weight cycle exists, our output is an array with minimum distance to all other nodes from source node 0.

Example 2:

Input:

Edges : [[0,1,1],[1,2,-1],[2,3,-1],[3,0,-1]]

Output: -1

Explanation: Notice that the graph has a negative edge weight cycle. 0->1->2->3(cost = -2). Hence, we output -1 to indicate the same.

**Solution :**

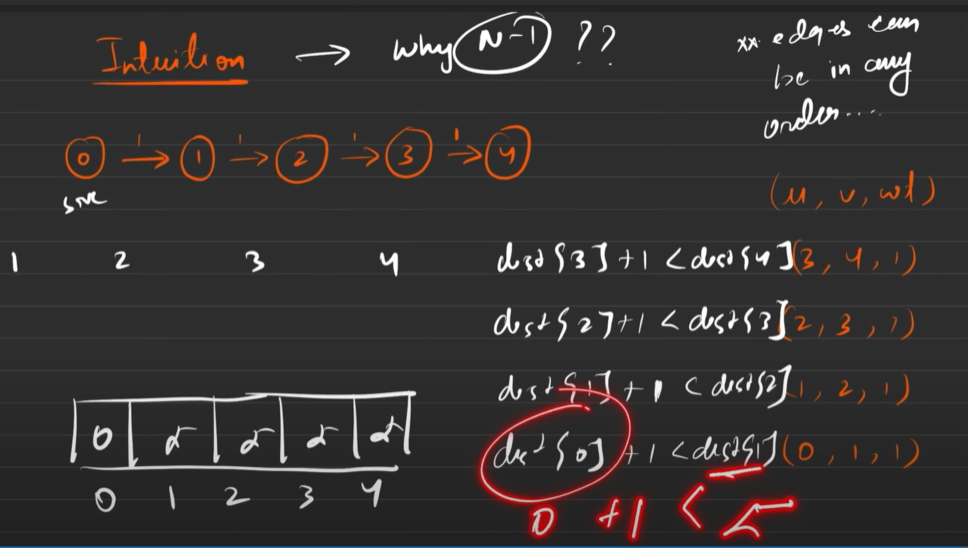
**It is used in directed graph , it is undirected graph represent as directed (0 – 1) = (0 -> 1)**

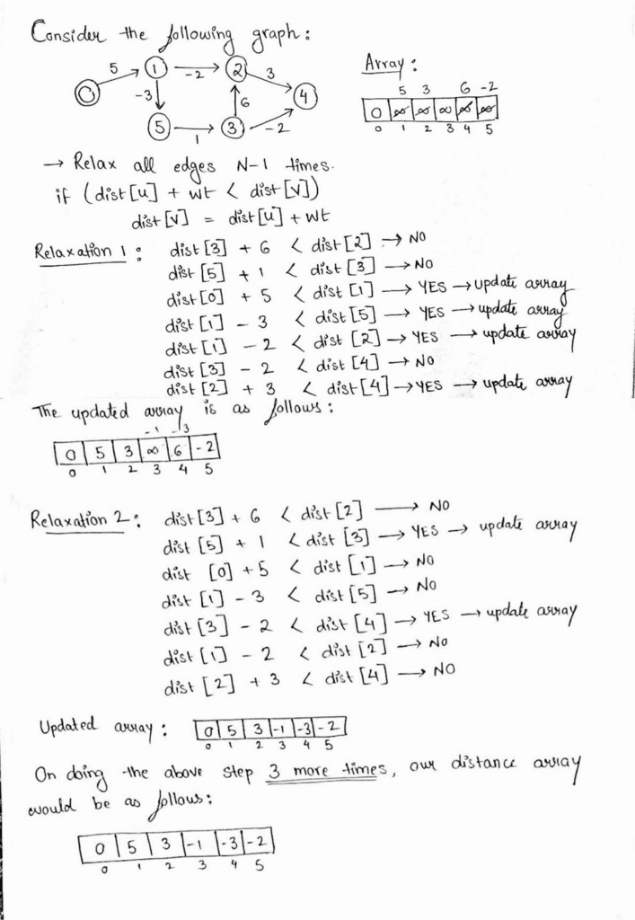
**(1 -> 0)**

**Recall that in Dijkstra's algorithm, we update the distance array every time we find a better solution which was a lesser distance. With the presence of negative edge weights, our algorithm would continue to update the distance array with lesser and lesser values and we might end up in time limit exceeded or segmentation fault error.**

**Now why N-1? our adjacency list might be in such a manner that only one node is updated in a single pass. Thus, to try out all nodes, we would require atleast N-1 iterations.The following is the demonstration of the same:**

**It will be any order based on given (3,4,1) , (2,3,10 ) ……**

****



**Check for Negative Cycles:** After completing the V−1 relaxations, you perform one more iteration over all edges. If you can still relax any edge (i.e., if the distance array is still updated), then there is a negative cycle in the graph.

**Time Complexity: O(N\*E). We check E edges N times**

**Space Complexity: O(N). Distance Array**

import java.util.\*;

class Node {

private int u;

private int v;

private int weight;

Node(int \_u, int \_v, int \_w) {

u = \_u;

v = \_v;

weight = \_w;

}

Node() {}

int getV() { return v; }

int getU() { return u; }

int getWeight() { return weight; }

}

class Main {

void bellmanFord(ArrayList<Node> edges, int N, int src) {

int dist[] = new int[N];

for (int i = 0; i < N; i++)

dist[i] = 10000000;

dist[src] = 0;

for (int i = 1; i <= N - 1; i++) {

for (Node node : edges) {

if (dist[node.getU()] + node.getWeight() < dist[node.getV()]) {

dist[node.getV()] = dist[node.getU()] + node.getWeight();

}

}

}

int fl = 0;

for (Node node : edges) {

if (dist[node.getU()] + node.getWeight() < dist[node.getV()]) {

fl = 1;

System.out.println("Negative Cycle");

break;

}

}

if (fl == 0) {

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

System.out.print(dist[i] + " ");

}

}

}

public static void main(String args[]) {

int n = 6;

ArrayList<Node> adj = new ArrayList<Node>();

adj.add(new Node(3, 2, 6));

adj.add(new Node(5, 3, 1));

adj.add(new Node(0, 1, 5));

adj.add(new Node(1, 5, -3));

adj.add(new Node(1, 2, -2));

adj.add(new Node(3, 4, -2));

adj.add(new Node(2, 4, 3));

Main obj = new Main();

obj.bellmanFord(adj, n, 0);

}

}

**38) Floyd Warshall Algorithm: G-42**

Problem Statement: The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed graph. The graph is represented as an adjacency matrix of size n\*n. Matrix[i][j] denotes the weight of the edge from i to j. If Matrix[i][j]=-1, it means there is no edge from i to j.

Do it in place.

Example 1:

Input Format:

matrix[][] = { {0, 2, -1, -1},

{1, 0, 3, -1},{-1, -1, 0, -1},{3, 5, 4, 0} }

Result:

0 2 5 -1

1 0 3 -1

-1 -1 0 -1

3 5 4 0

Explanation: In this example, the final matrix

is storing the shortest distances. For example, matrix[i][j] is

storing the shortest distance from node i to j.

Example 2:

Input Format:

matrix[][] = {{0,25},

{-1,0}}

Result:

0 25

-1 0﻿

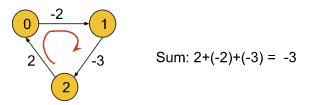
Explanation: In this example, the shortest distance

is already given (if it exists).

**Solution :**

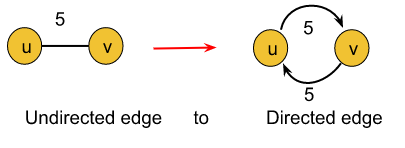
* ***How to detect a negative cycle using the Floyd Warshall algorithm?***

***Negative Cycle:****A cycle is called a negative cycle if the sum of all its weights becomes negative. The following illustration is an example of a negative cycle:*



* *We have previously said that the cost of reaching a node from itself must be 0. But in the above graph, if we try to reach node 0 from itself we can follow the path: 0->1->2->0. In this case, the cost to reach node 0 from itself becomes -3 which is less than 0. This is only possible if the graph contains a negative cycle.*
* *So, if we find that the cost of reaching any node from itself is less than 0, we can conclude that the graph has a negative cycle.*

In order to apply this algorithm to an undirected graph, we just need to convert the undirected edges into directed edges like the following:



import java.util.\*;

*//User function template for JAVA*

class Solution {

public void shortest\_distance(int[][] matrix) {

int n = matrix.length;

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

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

if (matrix[i][j] == -1) {

matrix[i][j] = (int)(1e9);

}

if (i == j) matrix[i][j] = 0;

}

}

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

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

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

matrix[i][j] = Math.min(matrix[i][j],

matrix[i][k] + matrix[k][j]);

}

}

}

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

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

if (matrix[i][j] == (int)(1e9)) {

matrix[i][j] = -1;

}

}

}

}

}

public class tUf {

public static void main(String[] args) {

int V = 4;

int[][] matrix = new int[V][V];

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

for (int j = 0; j < V; j++) {

matrix[i][j] = -1;

}

}

matrix[0][1] = 2;

matrix[1][0] = 1;

matrix[1][2] = 3;

matrix[3][0] = 3;

matrix[3][1] = 5;

matrix[3][2] = 4;

Solution obj = new Solution();

obj.shortest\_distance(matrix);

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

for (int j = 0; j < V; j++) {

System.out.print(matrix[i][j] + " ");

}

System.out.println("");

}

}

}

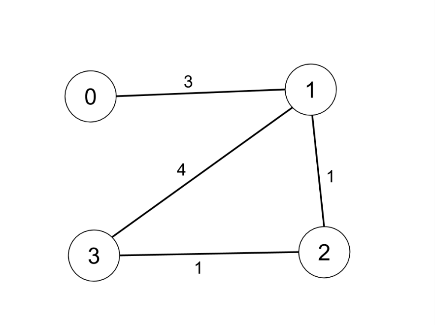
**39) Find the City With the Smallest Number of Neighbors at a Threshold Distance (Leetcode - 1334)**

There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold.

Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number.

Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path.

Example 1:



Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4

Output: 3

Explanation: The figure above describes the graph.

The neighboring cities at a distanceThreshold = 4 for each city are:

City 0 -> [City 1, City 2]

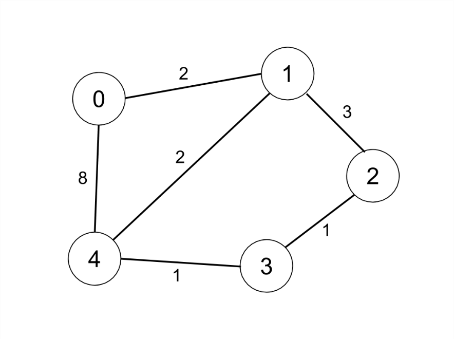
City 1 -> [City 0, City 2, City 3]

City 2 -> [City 0, City 1, City 3]

City 3 -> [City 1, City 2]

Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number.

Example 2:



Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distanceThreshold = 2

Output: 0

Explanation: The figure above describes the graph.

The neighboring cities at a distanceThreshold = 2 for each city are:

City 0 -> [City 1]

City 1 -> [City 0, City 4]

City 2 -> [City 3, City 4]

City 3 -> [City 2, City 4]

City 4 -> [City 1, City 2, City 3]

The city 0 has 1 neighboring city at a distanceThreshold = 2.

Constraints:

2 <= n <= 100

1 <= edges.length <= n \* (n - 1) / 2

edges[i].length == 3

0 <= fromi < toi < n

1 <= weighti, distanceThreshold <= 10^4

All pairs (fromi, toi) are distinct.

**Solution :**

class Solution {

    public void floydWarshall(int[][] dist,int n)

    {

        for(int i=0;i<n;i++)

        {

            for(int u=0;u<n;u++)

            {

                for(int v=0;v<n;v++)

                {

                    dist[u][v]=Math.min(dist[u][v],dist[u][i]+dist[i][v]);

                }

            }

        }

    }

    public int findTheCity(int n, int[][] edges, int distanceThreshold) {

        int[][] dist = new int[n][n];

        for(int[] i : edges)

        {

            dist[i[0]][i[1]]=i[2];

            dist[i[1]][i[0]]=i[2];

        }

        for (int i = 0; i < n; i++) {

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

                if (i == j) dist[i][j] = 0;

                else if (dist[i][j] == 0) {

                    dist[i][j] = (int)(1e9);

                }

            }

        }

        floydWarshall(dist,n);

        //now in dist matrix check no of cities with distance <= threshold for each city . Return the city with the smallest number of cities that are reachable, If there are multiple such cities, return the city with the greatest number.

        int min = Integer.MAX\_VALUE;

        int city=0;

        for (int i = 0; i < n; i++) {

            int cities = -1;

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

                if(dist[i][j]<=distanceThreshold)

                {

                    cities++;

                }

            }

            if(min >= cities)

            {

                min=cities;

                city = i;

            }

        }

        return city;

    }

}

**40) Minimum Spanning Tree - MST using Prim’s Algo**

Problem Statement: Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.

Definition: A minimum spanning tree consists of N nodes and N-1 edges connecting all the nodes which have the minimum cost(sum of edge weights).

Note: It is known as a tree since a tree doesn’t have cycles involved. A graph with N nodes and N-1 edges is equivalent to a tree.

**Solution:**

class Pair implements Comparable<Pair>{

int wt;

int v;

Pair(int v, int wt){

this.v = v;

this.wt = wt;

}

public int compareTo(Pair that){

return this.wt - that.wt;

}

}

class Solution {

static int spanningTree(int V, int E, List<List<int[]>> adj) {

// Code Here.

boolean[] vis = new boolean[V];

PriorityQueue<Pair> q = new PriorityQueue<>();

q.add(new Pair(0, 0));

int ans = 0;

while(!q.isEmpty()){

Pair cur = q.remove();

int u = cur.v;

if(vis[u]) continue;

ans += cur.wt;

vis[u] = true;

List<int[]> neigh = adj.get(u);

for(int[] list: neigh){

int vertex = list[0];

int wt = list[1];

if(vis[vertex] == false){

q.add(new Pair(vertex, wt));

}

}

}

return ans;

}

}

**41) Disjoint Set | Union by Rank | Union by Size | Path Compression: G-46**

In this article, we will discuss the Disjoint Set data structure which is a very important topic in the entire graph series. Let’s first understand why we need a Disjoint Set data structure using the below question:

Question: Given two components of an undirected graph

The question is whether node 1 and node 5 are in the same component or not.

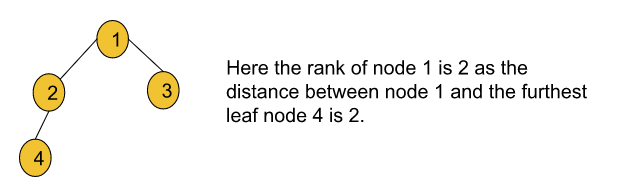
Approach:

Now, in order to solve this question we can use either the DFS or BFS traversal technique like if we traverse the components of the graph we can find that node 1 and node 5 are not in the same component. This is actually the brute force approach whose time complexity is O(N+E)(N = no. of nodes, E = no. of edges). But using a Disjoint Set data structure we can solve this same problem in constant time.

The disjoint Set data structure is generally used for dynamic graphs.

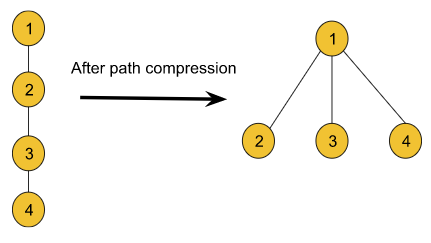
**Rank:**

The rank of a node generally refers to the distance (the number of nodes including the leaf node) between the furthest leaf node and the current node. Basically rank includes all the nodes beneath the current node.

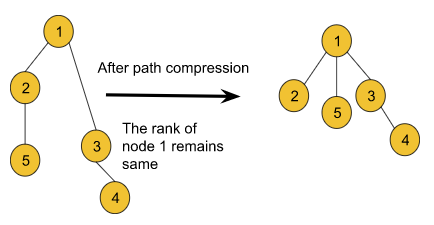


***What is path compression?***

Basically, connecting each node in a particular path to its ultimate parent refers to path compression. Let’s understand it using the following illustration:

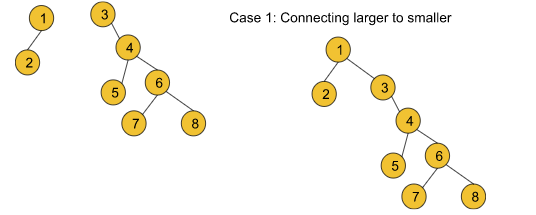


Though using the path compression technique it seems like the rank of the node is also changing, we cannot be sure about it. So, we will not make any changes to the rank array while applying path compression. The following example depicts an example:

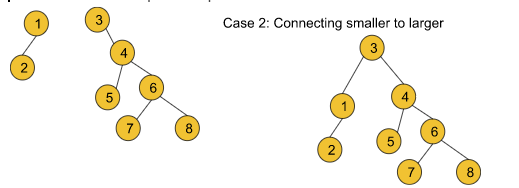


***In the union by rank method, why do we need to connect the smaller rank to the larger rank?***

* Let’s understand it using the following example:



In this case, the traversal time to find the ultimate parent for nodes 3, 4, 5, 6, 7, and 8 increases and so the path compression time also increases. But if we do the following



* the traversal time to find the ultimate parent for only nodes 1 and 2 increases. So the path compression time becomes relatively lesser than in the previous case. So, we can conclude that we should always connect a smaller rank to a larger one with the goal of
  + ***shrinking the height of the graph.***
  + ***reducing the time complexity as much as we can.***

**UNION BY SIZE :**

The algorithm steps are as follows:

* Firstly, the Union function requires two nodes(***let’s say u and v***) as arguments. Then we will find the ultimate parent (using the findPar() function discussed earlier) of u and v. Let’s consider the ultimate parent of u is ***pu***and the ultimate parent of v is ***pv***.
* After that, we will find the size of ***pu*** and ***pv*** i.e. size[pu] and size[pv].
* Finally, we will connect the ultimate parent with a smaller size to the other ultimate parent with a larger size. But if the size of the two is equal, we can connect any parent to the other parent.  
  While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

**Code:**

import java.io.\*;

import java.util.\*;

class DisjointSet {

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

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

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

public DisjointSet(int n) {

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

rank.add(0);

parent.add(i);

size.add(1);

}

}

public int findUPar(int node) {

if (node == parent.get(node)) {

return node;

}

int ulp = findUPar(parent.get(node));

parent.set(node, ulp);

return parent.get(node);

}

public void unionByRank(int u, int v) {

int ulp\_u = findUPar(u);

int ulp\_v = findUPar(v);

if (ulp\_u == ulp\_v) return;

if (rank.get(ulp\_u) < rank.get(ulp\_v)) {

parent.set(ulp\_u, ulp\_v);

} else if (rank.get(ulp\_v) < rank.get(ulp\_u)) {

parent.set(ulp\_v, ulp\_u);

} else {

parent.set(ulp\_v, ulp\_u);

int rankU = rank.get(ulp\_u);

rank.set(ulp\_u, rankU + 1);

}

}

public void unionBySize(int u, int v) {

int ulp\_u = findUPar(u);

int ulp\_v = findUPar(v);

if (ulp\_u == ulp\_v) return;

if (size.get(ulp\_u) < size.get(ulp\_v)) {

parent.set(ulp\_u, ulp\_v);

size.set(ulp\_v, size.get(ulp\_v) + size.get(ulp\_u));

} else {

parent.set(ulp\_v, ulp\_u);

size.set(ulp\_u, size.get(ulp\_u) + size.get(ulp\_v));

}

}

}

class Main {

public static void main (String[] args) {

DisjointSet ds = new DisjointSet(7);

ds.unionByRank(1, 2);

ds.unionByRank(2, 3);

ds.unionByRank(4, 5);

ds.unionByRank(6, 7);

ds.unionByRank(5, 6);

// if 3 and 7 same or not

if (ds.findUPar(3) == ds.findUPar(7)) {

System.out.println("Same");

} else

System.out.println("Not Same");

ds.unionByRank(3, 7);

if (ds.findUPar(3) == ds.findUPar(7)) {

System.out.println("Same");

} else

System.out.println("Not Same");

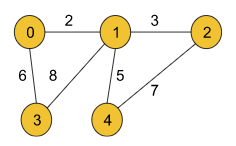
}

}

**42) Kruskal's Algorithm - Minimum Spanning Tree : G-47**

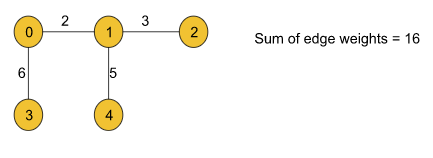
Problem Statement: Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.

Example 1:



Input Format:

V = 5, edges = { {0, 1, 2}, {0, 3, 6}, {1, 2, 3}, {1, 3, 8}, {1, 4, 5}, {4, 2, 7}}

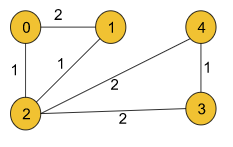


Result: 16

Explanation: The minimum spanning tree for the given graph is drawn below:

MST = {(0, 1), (0, 3), (1, 2), (1, 4)}

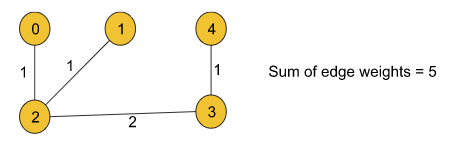
Example 2:



Input Format:

V = 5,

edges = { {0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}}



Result: 5

Explanation: The minimum spanning tree is drawn below:

MST = {(0, 2), (1, 2), (2, 3), (3, 4)}

**Solution :**

import java.io.\*;

import java.util.\*;

*// User function Template for Java*

class DisjointSet {

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

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

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

public DisjointSet(int n) {

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

rank.add(0);

parent.add(i);

size.add(1);

}

}

public int findUPar(int node) {

if (node == parent.get(node)) {

return node;

}

int ulp = findUPar(parent.get(node));

parent.set(node, ulp);

return parent.get(node);

}

public void unionByRank(int u, int v) {

int ulp\_u = findUPar(u);

int ulp\_v = findUPar(v);

if (ulp\_u == ulp\_v) return;

if (rank.get(ulp\_u) < rank.get(ulp\_v)) {

parent.set(ulp\_u, ulp\_v);

} else if (rank.get(ulp\_v) < rank.get(ulp\_u)) {

parent.set(ulp\_v, ulp\_u);

} else {

parent.set(ulp\_v, ulp\_u);

int rankU = rank.get(ulp\_u);

rank.set(ulp\_u, rankU + 1);

}

}

public void unionBySize(int u, int v) {

int ulp\_u = findUPar(u);

int ulp\_v = findUPar(v);

if (ulp\_u == ulp\_v) return;

if (size.get(ulp\_u) < size.get(ulp\_v)) {

parent.set(ulp\_u, ulp\_v);

size.set(ulp\_v, size.get(ulp\_v) + size.get(ulp\_u));

} else {

parent.set(ulp\_v, ulp\_u);

size.set(ulp\_u, size.get(ulp\_u) + size.get(ulp\_v));

}

}

}

class Edge implements Comparable<Edge> {

int src, dest, weight;

Edge(int \_src, int \_dest, int \_wt) {

this.src = \_src; this.dest = \_dest; this.weight = \_wt;

}

*// Comparator function used for*

*// sorting edgesbased on their weight*

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

};

class Solution {

*//Function to find sum of weights of edges of the Minimum Spanning Tree.*

static int spanningTree(int V,

ArrayList<ArrayList<ArrayList<Integer>>> adj) {

List<Edge> edges = new ArrayList<Edge>();

*// O(N + E)*

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

for (int j = 0; j < adj.get(i).size(); j++) {

int adjNode = adj.get(i).get(j).get(0);

int wt = adj.get(i).get(j).get(1);

int node = i;

Edge temp = new Edge(i, adjNode, wt);

edges.add(temp);

}

}

DisjointSet ds = new DisjointSet(V);

*// M log M*

Collections.sort(edges);

int mstWt = 0;

*// M x 4 x alpha x 2*

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

int wt = edges.get(i).weight;

int u = edges.get(i).src;

int v = edges.get(i).dest;

if (ds.findUPar(u) != ds.findUPar(v)) {

mstWt += wt;

ds.unionBySize(u, v);

}

}

return mstWt;

}

}

class Main {

public static void main (String[] args) {

int V = 5;

ArrayList<ArrayList<ArrayList<Integer>>> adj = new ArrayList<ArrayList<ArrayList<Integer>>>();

int[][] edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};

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

adj.add(new ArrayList<ArrayList<Integer>>());

}

for (int i = 0; i < 6; i++) {

int u = edges[i][0];

int v = edges[i][1];

int w = edges[i][2];

ArrayList<Integer> tmp1 = new ArrayList<Integer>();

ArrayList<Integer> tmp2 = new ArrayList<Integer>();

tmp1.add(v);

tmp1.add(w);

tmp2.add(u);

tmp2.add(w);

adj.get(u).add(tmp1);

adj.get(v).add(tmp2);

}

Solution obj = new Solution();

int mstWt = obj.spanningTree(V, adj);

System.out.println("The sum of all the edge weights: " + mstWt);

}

}

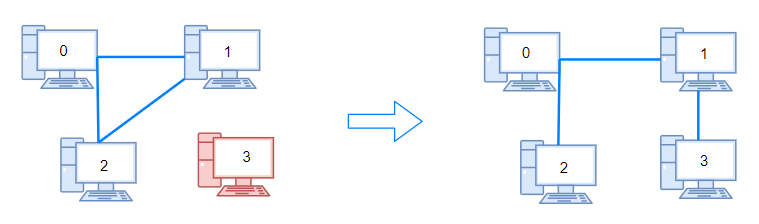
**43) Number of Operations to Make Network Connected (Leetcode - 1319)**

There are n computers numbered from 0 to n - 1 connected by ethernet cables connections forming a network where connections[i] = [ai, bi] represents a connection between computers ai and bi. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

Return the minimum number of times you need to do this in order to make all the computers connected. If it is not possible, return -1.

Example 1:

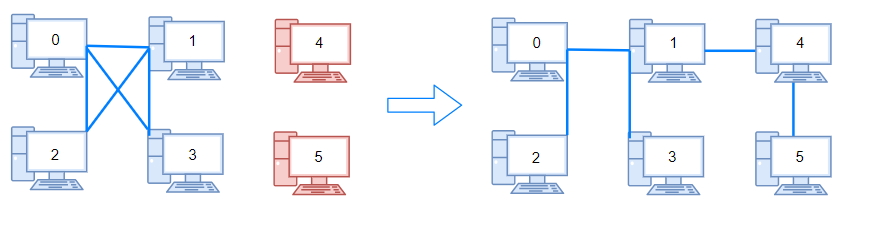


Input: n = 4, connections = [[0,1],[0,2],[1,2]]

Output: 1

Explanation: Remove cable between computer 1 and 2 and place between computers 1 and 3.

Example 2:



Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]

Output: 2

Example 3:

Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]

Output: -1

Explanation: There are not enough cables.

Constraints:

1 <= n <= 105

1 <= connections.length <= min(n \* (n - 1) / 2, 105)

connections[i].length == 2

0 <= ai, bi < n

ai != bi

There are no repeated connections.

No two computers are connected by more than one cable.

**Solution :**

class Solution {

    public int findParent(int[] parent, int i)

    {

        if(i==parent[i])

        {

            return i;

        }

        return parent[i]=findParent(parent,parent[i]);

    }

    public void unionByRank(int[] parent,int[] rank,int i,int j)

    {

        int u = findParent(parent,i);

        int v = findParent(parent,j);

        if(u == v) return ;

        if(rank[u]>rank[v])

        {

            parent[v]=u;

        }

        else if(rank[u]<rank[v])

        {

            parent[u]=v;

        }

        else

        {

            parent[v]=u;

            rank[u]++;

        }

    }

    public int makeConnected(int n, int[][] connections) {

        int rank[] = new int[n];

        int parent[] = new int[n];

        for(int i=0;i<n;i++)

        {

            parent[i]=i;

        }

        int extra = 0 ;

        for(int[] i : connections)

        {

            if(findParent(parent,i[0])==findParent(parent,i[1]))

            {

                extra++;

            }

            else

            {

                unionByRank(parent,rank,i[0],i[1]);

            }

        }

        int comp = 0;

        for(int i=0;i<n;i++)

        {

            if(i==parent[i])

            {

                comp++;

            }

        }

        if(extra >= comp-1)

        {

            return comp-1;

        }

        return -1;

    }

}

**44) Accounts Merge (Leetcode - 721)**

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Example 1:

Input: accounts = [["John","johnsmith@mail.com","john\_newyork@mail.com"],["John","johnsmith@mail.com","john00@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]

Output: [["John","john00@mail.com","john\_newyork@mail.com","johnsmith@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]

Explanation:

The first and second John's are the same person as they have the common email "johnsmith@mail.com".

The third John and Mary are different people as none of their email addresses are used by other accounts.

We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],

['John', 'john00@mail.com', 'john\_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.

Example 2:

Input: accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"],["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"],["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]]

Output: [["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"],["Gabe","Gabe0@m.co","Gabe1@m.co","Gabe3@m.co"],["Hanzo","Hanzo0@m.co","Hanzo1@m.co","Hanzo3@m.co"],["Kevin","Kevin0@m.co","Kevin3@m.co","Kevin5@m.co"],["Fern","Fern0@m.co","Fern1@m.co","Fern5@m.co"]]

Constraints:

1 <= accounts.length <= 1000

2 <= accounts[i].length <= 10

1 <= accounts[i][j].length <= 30

accounts[i][0] consists of English letters.

accounts[i][j] (for j > 0) is a valid email

ex:

Given: N = 6

accounts [ ] =

[["John","j1@com","j2@com","j3@com"],

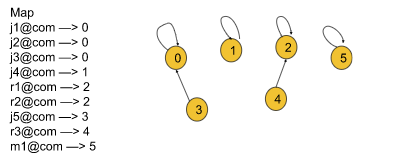
["John","j4@com"],

["Raj",”r1@com”, “r2@com”],

["John","j1@com","j5@com"],

["Raj",”r2@com”, “r3@com”],

["Mary","m1@com"]]



**Solution :**

class Solution {

    public int findParent(int[] parent, int i)

    {

        if(parent[i]==i)

        {

            return i;

        }

        return parent[i] = findParent(parent,parent[i]);

    }

    public void unionBySize(int[] parent,int[] size,int i,int j)

    {

        int u = findParent(parent,i);

        int v = findParent(parent,j);

        if(u==v) return;

        if(size[u]<size[v])

        {

            parent[u]=v;

            size[v]=size[u]+size[v];

        }

        else

        {

            parent[v]=u;

            size[u]=size[u]+size[v];

        }

    }

    public List<List<String>> accountsMerge(List<List<String>> accounts) {

        int n = accounts.size();

        int parent[] = new int[n];

        int size[] = new int[n];

        for(int i=0;i<n;i++)

        {

            parent[i]=i;

size[i] = 1;

        }

        Map<String,Integer> map = new HashMap<>();

        for(int i=0;i<n;i++)

        {

            for(int j=1;j<accounts.get(i).size();j++)

            {

                String email = accounts.get(i).get(j);

                if(!map.containsKey(email))

                {

                    map.put(email,i);

                }

                else

                {

                    unionBySize(parent,size,i,map.get(email));

                }

            }

        }

        //merge

        Map<Integer,List<String>> merge = new HashMap<>();

        for(Map.Entry<String,Integer> i : map.entrySet())

        {

            int p = findParent(parent,i.getValue());

            String email = i.getKey();

            if(!merge.containsKey(p))

            {

                merge.put(p,new ArrayList<>());

            }

           merge.get(p).add(email);

        }

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

        for(Map.Entry<Integer,List<String>> i : merge.entrySet())

        {

            List<String> temp = new ArrayList<>(i.getValue());

            temp.sort((a,b)-> a.compareTo(b));

            temp.add(0,accounts.get(i.getKey()).get(0));

            res.add(temp);

        }

        return res;

    }

}

Here *N* is the number of accounts and *K* is the maximum length of an account.

* Time complexity: *O*(*NK*log*NK*)

While merging we consider the size of each connected component and we always choose the representative of the larger component to be the new representative of the smaller component, also we have included the path compression so the time complexity for find/union operation is *α*(*N*) (Here, *α*(*N*) is the inverse Ackermann function that grows so slowly, that it doesn't exceed 4 for all reasonable *N* (approximately *N*<10600).

We find the representative of all the emails, hence it will take *O*(*NKα*(*N*)) time. We are also sorting the components and the worst case will be when all emails end up belonging to the same component this will cost *O*(*NK*(log*NK*)).

Hence the total time complexity is *O*(*NK*⋅log*NK*+*NK*⋅*α*(*N*)).

* Space complexity: *O*(*NK*)

List representative, size store information corresponding to each group so will take *O*(*N*) space. All emails get stored in emailGroup and component hence space used is *O*(*NK*).

The space complexity of the sorting algorithm depends on the implementation of each programming language. For instance, in Java, Collections.sort() dumps the specified list into an array this will take *O*(*NK*) space then Arrays.sort() for primitives is implemented as a variant of quicksort algorithm whose space complexity is *O*(log*NK*). In C++ sort() function provided by STL is a hybrid of Quick Sort, Heap Sort, and Insertion Sort with the worst-case space complexity of *O*(log*NK*).

**45) Number Of Islands (GFG)**

You are given a n,m which means the row and column of the 2D matrix and an array of size k denoting the number of operations. Matrix elements is 0 if there is water or 1 if there is land. Originally, the 2D matrix is all 0 which means there is no land in the matrix. The array has k operator(s) and each operator has two integer A[i][0], A[i][1] means that you can change the cell matrix[A[i][0]][A[i][1]] from sea to island. Return how many island are there in the matrix after each operation.You need to return an array of size k.

Note : An island means group of 1s such that they share a common side.

Example 1:

Input: n = 4

m = 5

k = 4

A = {{1,1},{0,1},{3,3},{3,4}}

Output: 1 1 2 2

Explanation:

0. 00000

00000

00000

00000

1. 00000

01000

00000

00000

2. 01000

01000

00000

00000

3. 01000

01000

00000

00010

4. 01000

01000

00000

00011

Example 2:

Input: n = 4

m = 5

k = 4

A = {{0,0},{1,1},{2,2},{3,3}}

Output: 1 2 3 4

Explanation:

0. 00000

00000

00000

00000

1. 10000

00000

00000

00000

2. 10000

01000

00000

00000

3. 10000

01000

00100

00000

4. 10000

01000

00100

00010

Your Task:

You don't need to read or print anything. Your task is to complete the function numOfIslands() which takes an integer n denoting no. of rows in the matrix, an integer m denoting the number of columns in the matrix and a 2D array of size k denoting the number of operators.

Expected Time Complexity: O(m \* n)

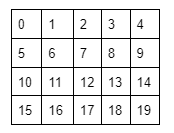
Expected Auxiliary Space: O(m \* n)

Constraints:

1 <= n,m <= 100

1 <= k <= 1000

For example, if a 5X4 matrix is given we will number the cell in the following way:



Now if we want to connect cells (1,0) and (2,0), we will just perform a union of 5 and 10. The number for representing each cell can be found using the following formula:  
number = (row of the current cell\*total number of columns)+column of the current cell for example, for the cell (2, 0) the number is = (2\*5) + 0 = 10.

**Solution :**

class Solution {

public int findParent(int[] parent, int i)

{

if(parent[i]==i)

{

return i;

}

return parent[i]=findParent(parent,parent[i]);

}

public void unionBySize(int[] parent,int size[], int i, int j)

{

int u = findParent(parent,i);

int v = findParent(parent,j);

if(u==v)

{

return ;

}

if(size[u] < size[v])

{

parent[u]=v;

size[v] = size[u]+size[v];

}

else

{

parent[v]=u;

size[u] = size[u] + size[v];

}

}

public List<Integer> numOfIslands(int rows, int cols, int[][] operators) {

int n = rows\*cols;

int parent[] = new int[n];

int size[] = new int[n];

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

{

parent[i]=i;

size[i]=1;

}

int[][] visited = new int[rows][cols];

int comp = 0;

int[][] dirs = {{0,-1},{0,1},{-1,0},{1,0}};

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

for(int[] i : operators)

{

int cur = cols \* i[0] + i[1] ;

//contains dublicate edges

if(visited[i[0]][i[1]]==1)

{

res.add(comp);

continue;

}

comp++;

visited[i[0]][i[1]]=1;

for(int[] dir : dirs )

{

int adjR = i[0] + dir[0];

int adjC = i[1] + dir[1];

if(adjR<rows && adjR >= 0 && adjC < cols && adjC >= 0 && visited[adjR][adjC]==1)

{

int adj = cols \* adjR + adjC;

if(findParent(parent,adj)!=findParent(parent,cur))

{

unionBySize(parent,size,adj,cur);

comp--;

}

}

}

res.add(comp);

}

return res;

}

}

**Time Complexity = O(rows\*cols)**

**46) Making A Large Island (Leetcode - 827)**

You are given an n x n binary matrix grid. You are allowed to change at most one 0 to be 1.

Return the size of the largest island in grid after applying this operation.

An island is a 4-directionally connected group of 1s.

Example 1:

Input: grid = [[1,0],[0,1]]

Output: 3

Explanation: Change one 0 to 1 and connect two 1s, then we get an island with area = 3.

Example 2:

Input: grid = [[1,1],[1,0]]

Output: 4

Explanation: Change the 0 to 1 and make the island bigger, only one island with area = 4.

Example 3:

Input: grid = [[1,1],[1,1]]

Output: 4

Explanation: Can't change any 0 to 1, only one island with area = 4.

Constraints:

n == grid.length

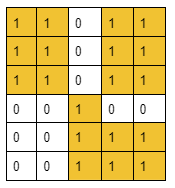
n == grid[i].length

1 <= n <= 500

grid[i][j] is either 0 or 1.

**Edge Case:**

Here is the edge case. Let’s understand it using the following example.



In this given grid, we will check for every cell with the value 0. When we come to cell (3,3), we will check all four adjacent cells to get the components’ sizes. Now it will first add the component of size 7 in our answer while checking the left cell and will again add the same component while checking the downward cell. This is where the answer gets incorrect. ***So, to avoid this edge case,*** ***instead of adding the component sizes to our answer we will store the ultimate parents in a set data structure***. This process will automatically discard the case of adding duplicate components. After that, to get the size of the ultimate parents we will just refer to the ultimate parent index of the size array inside the Disjoint Set data structure(size[ultimateParent]). Thus we will get the final answer.

**The algorithm steps are as follows (*step 3 is very important*):**

* Our first objective is to connect all the nodes that have formed groups. In order to do so, we will visit each cell of the grid and check if it contains the value 1.
  + If the value is 1, we will check all four adjacent cells of the current cell. If we find any adjacent cell with the same value 1, we will perform the union(***either unionBySize()*** of the two node numbers that represent those two cells i.e. the current cell and the adjacent cell.
  + Now, step 1 is completed.
* Then, we will again visit each cell of the grid and check if it contains the value 0.
  + If the value is 0, we will check all four adjacent cells of the current cell. If we found any cell with value 1, we will just insert the ultimate parent of that cell(using the ***findUPar()*** method) in the set data structure. This process will add the adjacent components to our answer.
  + After doing so for all the adjacent cells containing 1, we will iterate through the set data structure and add the size of each ultimate parent(*referring to the size array inside the Disjoint Set data structure*) to our answer. Finally, we will add an extra 1 to our answer for the current cell being included in the group.
  + Now, we will compare to get the maximum answer among all the previous answers we got for the previous cells with the value 0 and the current one.
* But if the matrix does not contain any cell with 0, step 2 will not be executed. For that reason, we will just run a loop from node number 0 to n\*n and for each node number, we will find the ultimate parent. After that, we will find the sizes of those ultimate parents and will take the size of the largest one.
* Thus we will get the maximum size of the group of connected 1s stored in our answer.

**Solution :**

class Solution {

    public int findParent(int[] parent, int i)

    {

        if(i==parent[i])

        {

            return i;

        }

        return parent[i]=findParent(parent, parent[i]);

    }

    public void unionBySize(int[] parent,int[] size,int i,int j)

    {

        int u = findParent(parent,i);

        int v = findParent(parent,j);

        if(u==v) return ;

        if(size[u] < size[v])

        {

            parent[u] = v;

            size[v] = size[u]+size[v];

        }

        else

        {

            parent[v] = u;

            size[u] = size[u]+size[v];

        }

    }

    public int largestIsland(int[][] grid) {

        int n = grid.length;

        int m = grid[0].length;

        int tot = n\*m;

        int size[] = new int[tot];

        int parent[] = new int[tot];

        Set<Integer> set = new HashSet<>();

        for(int i=0;i<tot;i++)

        {

            parent[i]=i;

            size[i]=1;

        }

        int[][] dirs = {{0,-1},{0,1},{-1,0},{1,0}};

        for(int i = 0;i<n;i++)

        {

            for(int j=0;j<m; j++)

            {

                if(grid[i][j]==1)

                {

                    int cur = n \* i + j;

                    for(int[] dir : dirs)

                    {

                        int row = i + dir[0];

                        int col = j + dir[1];

                        if(row >= 0 && row < n && col >= 0 && col < m && grid[row][col]==1)

                        {

                            int adj = row \* n + col;

                            unionBySize(parent,size,cur,adj);

                        }

                    }

                }

            }

        }

        int maxSize = 0;

        for(int i=0;i<n;i++)

        {

            for(int j=0;j<m;j++)

            {

                if(grid[i][j]==0)

                {

                    int s = 0;

                    for(int[] dir : dirs)

                    {

                        int row = i + dir[0];

                        int col = j + dir[1];

                        if(row >= 0 && row < n && col >= 0 && col < m && grid[row][col]==1)

                        {

                            int adj = row \* n + col;

                            int p = findParent(parent,adj);

                            if(!set.contains(p))

                            {

                                set.add(p);

                                s += size[p];

                            }

                        }

                    }

                    set.clear();

                    maxSize = Math.max(s+1,maxSize);

                }

            }

        }

        // for grid containint all 1's i.e if it contains all 1's maxSize is zero and size[0] contains total elements

        maxSize = Math.max(maxSize, size[0]);

        return maxSize;

    }

}

**47) Most Stones Removed with Same Row or Column (Leetcode - 947)**

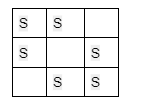
On a 2D plane, we place n stones at some integer coordinate points. Each coordinate point may have at most one stone.

A stone can be removed if it shares either the same row or the same column as another stone that has not been removed.

Given an array stones of length n where stones[i] = [xi, yi] represents the location of the ith stone, return the largest possible number of stones that can be removed.

Example 1:

Input: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]



Output: 5

Here no of comp = 1;

Explanation: One way to remove 5 stones is as follows:

1. Remove stone [2,2] because it shares the same row as [2,1].

2. Remove stone [2,1] because it shares the same column as [0,1].

3. Remove stone [1,2] because it shares the same row as [1,0].

4. Remove stone [1,0] because it shares the same column as [0,0].

5. Remove stone [0,1] because it shares the same row as [0,0].

Stone [0,0] cannot be removed since it does not share a row/column with another stone still on the plane.

Example 2:

Input: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]

Output: 3

Explanation: One way to make 3 moves is as follows:

1. Remove stone [2,2] because it shares the same row as [2,0].

2. Remove stone [2,0] because it shares the same column as [0,0].

3. Remove stone [0,2] because it shares the same row as [0,0].

Stones [0,0] and [1,1] cannot be removed since they do not share a row/column with another stone still on the plane.

Example 3:

Input: stones = [[0,0]]

Output: 0

Explanation: [0,0] is the only stone on the plane, so you cannot remove it.

Constraints:

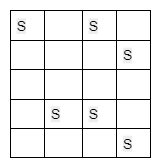
1 <= stones.length <= 1000

0 <= xi, yi <= 104

No two stones are at the same coordinate point.

**Solution:**

Let’s first understand the thought process that we will be using to solve this problem. In this problem, it is clearly stated that a stone can be removed if it shares either the same row or the same column as another stone that has not been removed. So, we can assume that these types of stones, sharing either the same row or column, are connected and belong to the same group. If we take example 2:



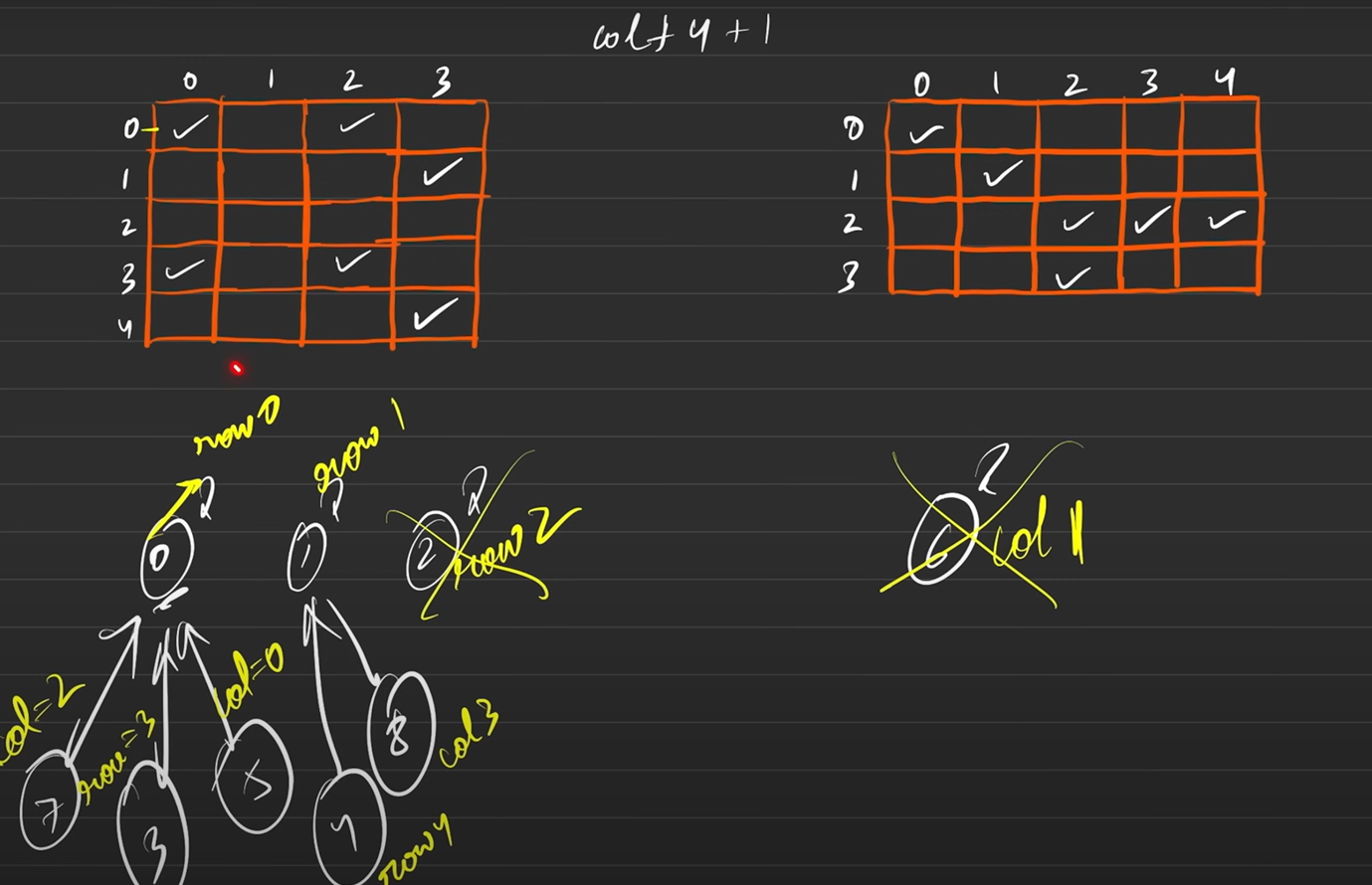
We can easily spot two different groups in this example. The first group includes the stones [0,0], [0,2], [3,2], and [3,1], and the second one includes [1,3] and [4,3].

If we carefully observe, for each group we can remove all the stones leaving one stone intact. So, we can conclude that at most we can remove (size of the group -1) no. of stones from a group as we need to leave one stone untouched for each group.

**Approach: (joining rows and columns)**

The algorithm steps are as follows:

* First, from the stone information, we will find out the maximum row and the maximum column number so that we can get an idea about the size of the 2D plane(i.e. nothing but a matrix).
* Then, we need to create a disjoint set of sizes (maximum row index+maximum column index+2)
* Now it’s time to connect the cells having a stone. For that we will loop through the given cell information array and for each cell we will extract the row and the column number and do the following:
  + First, we will convert column no. to (column no. + maximum row index +1).
  + We will perform the union(***either unionBySize() or unionByRank()***) of the row number and the converted column number.
* Now, it’s time to calculate the number of components and for that, we will count the number of ultimate parents. Here we will refer to the size array.
  + In this we check parent[i]==i and size[i]>1 i.e size[i] is not > 1 then it means the row or colums does n’t contains stone.



* Finally, we will subtract the no. of components(i.e. no. of ultimate parents) from the total no. of stones and we will get our answer.

**Solution :**

class Solution {

    public int findParent(int[] parent,int i)

    {

        if(i==parent[i])

        {

            return i;

        }

        return parent[i] = findParent(parent,parent[i]);

    }

    public void unionBySize(int[] parent,int size[], int i,int j)

    {

        int u = findParent(parent,i);

        int v = findParent(parent,j);

        if(u==v) return ;

        if(size[u] < size[v])

        {

            parent[u]=v;

            size[v]=size[u]+size[v];

        }

        else{

            parent[v] = u;

            size[u] = size[u] + size[v];

        }

    }

    public int removeStones(int[][] stones) {

        int maxRow = 0;

        int maxCol = 0;

        for(int[] i: stones)

        {

            maxRow = Math.max(maxRow,i[0]);

            maxCol = Math.max(maxCol,i[1]);

        }

        int n= maxRow + maxCol + 2;

        int parent[] = new int[n];

        int size[] = new int[n];

        for(int i=0;i<n;i++)

        {

            parent[i]=i;

            size[i]=1;

        }

        for(int[] i : stones)

        {

            int row = i[0];

            int col = maxRow + i[1] + 1;

            unionBySize(parent,size,row,col);

        }

        int comp = 0;

        for(int i=0;i<n;i++)

        {

            if(parent[i]==i && size[i]>1)

            {

                comp++;

            }

        }

        return stones.length - comp;

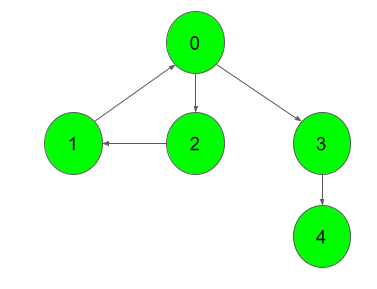
    }

}

**48) Strongly Connected Components (Kosaraju's Algo)**

Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, Find the number of strongly connected components in the graph.

Example 1:

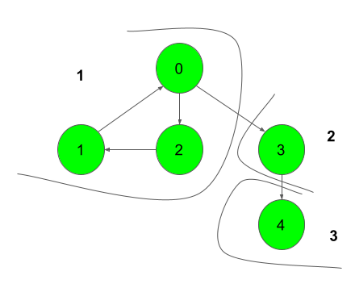


Input:

Output:

3

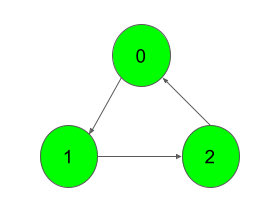
Explanation:



We can clearly see that there are 3 Strongly Connected Components in the Graph

Example 2:

Input:



Output:

1

Explanation:

All of the nodes are connected to each other.

So, there's only one SCC.

Your Task:

You don't need to read input or print anything. Your task is to complete the function kosaraju() which takes the number of vertices V and adjacency list of the graph of size V as inputs and returns an integer denoting the number of strongly connected components in the given graph.

Expected Time Complexity: O(V+E).

Expected Auxiliary Space: O(V+E).

Constraints:

1 ≤ V ≤ 5000

0 ≤ E ≤ (V\*(V-1))

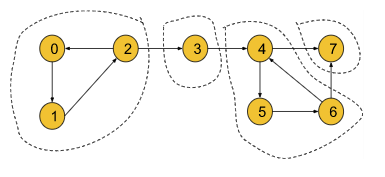
0 ≤ u, v ≤ V-1

Sum of E over all testcases will not exceed 25\*106

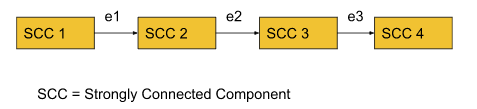
**Kosaraju’s Algorithm:**

To find the strongly connected components of a given directed graph, we are going to use Kosaraju’s Algorithm.

Before understanding the algorithm, we are going to discuss the thought process behind it. If we start DFS from node 0 for the following graph, we will end up visiting all the nodes. So, it is impossible to differentiate between different SCCs.

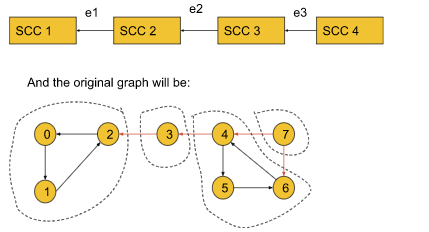


Now, we need to think in a different way. We can convert the above graph into the following illustration:



By definition, within each SCC, every node is reachable. So, if we start DFS from a node of SCC1 we can visit all the nodes in SCC1 and via edge e1 we can reach SCC2. Similarly, we can travel from SCC2 to SCC3 via e2 and SCC3 to SCC4 via e3. Thus all the nodes of the graph become reachable.

But if we reverse the edges e1, e2, and e3, the graph will look like the following:



Now in this graph, if we start DFS from node 0 it will visit only the nodes of SCC1. Similarly, if we start from node 3 it will visit only the nodes of SCC2. Thus, by reversing the SCC-connecting edges, the adjacent SCCs become unreachable. Now, the DFS will work in such a way, that in one DFS call we can only visit the nodes of a particular SCC. So,***the number of DFS calls will represent the number of SCCs***.

Until now, we have successfully found out the process of getting the number of SCCs. But here, comes a new problem i.e. if we do not know the SCCs, how the edges will be reversed? To solve this problem, we will simply try to reverse all the edges of the graph like the following:



If we carefully observe, the nodes within an SCC are reachable from each one to everyone even if we reverse the edges of the SCC. So, the SCCs will have no effect on reversing the edges. Thus we can fulfill our intention of reversing the SCC-connecting edge without affecting the SCCs.

Now, the question might be like, if node 0 is located in SCC4 and we start DFS from node 0, again we will visit all the SCCs at once even after reversing the edges. This is where ***the*** ***starting time and the finishing time*** concept will come in.

Now, we have a clear intuition about reversing edges before we move on to the starting and the finishing time concept in the algorithm part.

**Algorithm**:

The algorithm steps are as follows:

* ***Sort all the nodes according to their finishing time:***To sort all the nodes according to their finishing time, we will start DFS from node 0 and while backtracking in the DFS call we will store the nodes in a stack data structure. The nodes in the last SCC will finish first and will be stored in the last of the stack. After the DFS gets completed for all the nodes, the stack will be storing all the nodes in the sorted order of their finishing time.
* ***Reverse all the edges of the entire graph:***Now, we will create another adjacency list and store the information of the graph in a reversed manner.
* ***Perform the DFS and count the no. of different DFS calls to get the no. of SCC:***Now, we will start DFS from the node which is on the top of the stack and continue until the stack becomes empty. For each individual DFS call, we will increment the counter variable by 1. We will get the number of SCCs by just counting the number of individual DFS calls as in each individual DFS call, all the nodes of a particular SCC get visited.
* Finally, we will get the number of SCCs in the counter variable. If we want to store the SCCs as well, we need to store the nodes in some array during each individual DFS call in step 3.

**Solution :**

class Solution

{

//Function to find number of strongly connected components in the graph.

public void dfs(int node, int[] vis, ArrayList<ArrayList<Integer>> adj,Stack<Integer> stack)

{

vis[node]=1;

for(int i : adj.get(node))

{

if(vis[i]==0)

{

dfs(i,vis,adj,stack);

}

}

stack.push(node);

}

public void dfs(int node,int[] vis, ArrayList<ArrayList<Integer>> adj)

{

vis[node]=1;

for(int i : adj.get(node))

{

if(vis[i]==0)

{

dfs(i,vis,adj);

}

}

}

public int kosaraju(int V, ArrayList<ArrayList<Integer>> adj)

{

Stack<Integer> stack = new Stack<>();

int[] vis = new int[V];

//sorting the graph (toposort)

for(int i=0;i<V;i++)

{

if(vis[i]==0)

{

dfs(i,vis,adj,stack);

}

}

ArrayList<ArrayList<Integer>> rev = new ArrayList<>();

for(int i=0;i<V;i++)

{

rev.add(new ArrayList<>());

}

//reversing the graph

for(int i=0;i<V;i++)

{

vis[i]=0;

for(int j:adj.get(i))

{

rev.get(j).add(i);

}

}

int comp = 0;

while(!stack.isEmpty())

{

int cur =stack.pop();

if(vis[cur]==0)

{

dfs(cur,vis,rev);

comp++;

}

}

return comp;

}

}

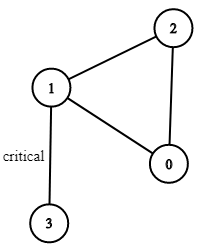
**49) Critical Connections in a Network (Leetcode - 1192)**

There are n servers numbered from 0 to n - 1 connected by undirected server-to-server connections forming a network where connections[i] = [ai, bi] represents a connection between servers ai and bi. Any server can reach other servers directly or indirectly through the network.

A critical connection is a connection that, if removed, will make some servers unable to reach some other server.

Return all critical connections in the network in any order.

Example 1:



Input: n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]

Output: [[1,3]]

Explanation: [[3,1]] is also accepted.

Example 2:

Input: n = 2, connections = [[0,1]]

Output: [[0,1]]

Constraints:

2 <= n <= 105

n - 1 <= connections.length <= 105

0 <= ai, bi <= n - 1

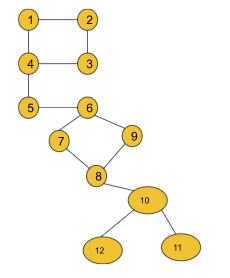
ai != bi

There are no repeated connections.

**Bridge:**

Any edge in a component of a graph is called a bridge when the component is divided into 2 or more components if we remove that particular edge.

**Example:**



If in this graph, we remove the edge (5,6), the component gets divided into 2 components. So, it is a bridge. But if we remove the edge (2,3) the component remains connected. So, this is not a bridge. In this graph, we have a total of 3 bridges i.e. (4,5), (5,6), and (10, 8).

In order to find all the bridges of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let’s discuss the algorithm in detail. Before that, we will discuss two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion.***

* **Time of insertion:**Dring the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 1 it will visit node 1 first then node 2, node 3, node 4, and so on. So, the time of insertion for node 1 will be 1, node 2 will be 2, node 3 will be 3 and it will continue like this. ***To store the time of insertion for each node, we will use a time array***.
* **Lowest time of insertion:**In this case, the current node refers to all its adjacent nodes ***except the parent***and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another ‘low’ array.

***The logical modification of the DFS algorithm is discussed below***:

After the DFS for any adjacent node gets completed, we will just check if the edge, whose starting point is the current node and ending point is that adjacent node, is a bridge. For that, we will just check if any other path from the current node to the adjacent node exists if we remove that particular edge. If any other alternative path exists, this edge is not a bridge. Otherwise, it can be considered a valid bridge.

**Approach**:

The algorithm steps are as follows:

* First, we need to create the adjacency list for the given graph from the edge information(***If not already given***). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node.
* Then we will start DFS from node 0(assuming the graph contains a single component otherwise, we will call DFS for every component) with parent -1.
  + Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. The timer may be initialized to 0 or 1.
  + Now, it’s time to visit the adjacent nodes.
    - ***If the adjacent node is the parent itself***, we will just continue to the next node.
    - ***If the adjacent node is not visited***, we will call DFS for the adjacent node with the current node as the parent.  
      After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.  
      Now, we will check if the lowest time of insertion of the adjacent node is greater than the time of insertion of the current node.  
      If it is, then we will store the adjacent node and the current node in our answer array as they are representing the bridge.
    - ***If the adjacent node is already visited***, we will just compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.
* Finally, our answer array will store all the bridges.

**Solution :**

class Solution {

    int time = 1;

    public void dfs(int node,int parent,int[] timeIn,int[] low,List<Integer>[] graph,List<List<Integer>> bridges)

    {

        timeIn[node]=low[node]=time++;

        for(int i:graph[node])

        {

            //to check visited or not

            if(timeIn[i]==0)

            {

                dfs(i,node,timeIn,low,graph,bridges);

                low[node]=Math.min(low[i],low[node]);

                if(low[i] > timeIn[node])

                {

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

                    res.add(node);

                    res.add(i);

                    bridges.add(res);

                }

            }

            else if(i!=parent)

            {

                low[node]=Math.min(low[i],low[node]);

            }

        }

    }

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        //convert to graph

        List<Integer>[] graph = new ArrayList[n];

        for(int i=0;i<n;i++)

        {

            graph[i]=new ArrayList<>();

        }

        for(List<Integer> i:connections)

        {

            graph[i.get(0)].add(i.get(1));

            graph[i.get(1)].add(i.get(0));

        }

        int timeIn[] = new int[n];

        int low[] = new int[n];

        int parent = -1;

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

        dfs(0,parent,timeIn,low,graph,bridges);

        return bridges;

    }

}

**50) Articulation Point in Graph: G-56**

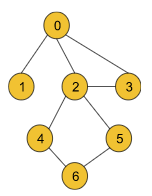
**Problem Statement:** Given an undirected connected graph with V vertices and adjacency list adj. You are required to find all the vertices removing which (and edges through it) disconnect the graph into 2 or more components.

Note: Indexing is zero-based i.e nodes numbering from (0 to V-1). There might be loops present in the graph.

**Pre-requisite:**[**Bridges in Graph**](https://www.youtube.com/watch?v=qrAub5z8FeA&list=PLgUwDviBIf0oE3gA41TKO2H5bHpPd7fzn&index=55) problem & [**DFS algorithm**](https://takeuforward.org/data-structure/depth-first-search-dfs/).

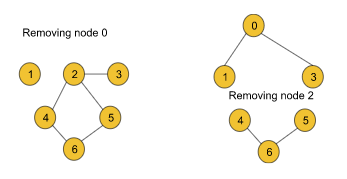
**Example 1**:

**Input Format:**



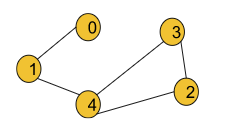
**Result:** {0, 2}

**Explanation:** If we remove node 0 or node 2, the graph will be divided into 2 or more components.



**Example 2**:

**Input Format:**



**Result:** {1, 4}

**Explanation:** If we remove either node 1 or node 4, the graph breaks into multiple components.

**Solution**

***Disclaimer***: *Don't jump directly to the solution, try it out yourself first.*

[*Problem Link*](https://practice.geeksforgeeks.org/problems/articulation-point-1/1?utm_source=youtube&utm_medium=collab_striver_ytdescription&utm_campaign=articulation-point)*.*

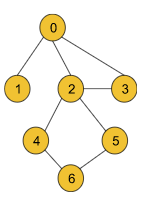
**Solution:**

Before moving on to the solution, we need to understand the definition of the articulation point of a graph.

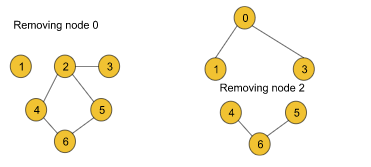
**Articulation Point:**

Articulation Points of a graph are the nodes on whose removal, the graph breaks into multiple components.

Example:



For the above graph node 0 and node, 2 are the articulation points. If we remove either of the two nodes, the graph breaks into multiple components like the following:



But node 3 is not an articulation point as this node’s removal does not break the graph into multiple components.

In order to find all the articulation points of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let’s discuss the algorithm in detail. Before that, we will discuss the two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion.***

* **Time of insertion:**Dring the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 0 it will visit node 1 first then node 2, node 3, and so on. So, the time of insertion for node 0 will be 1, node 1 will be 2, node 2 will be 3 and it will continue like this. ***We will use a time array to store the insertion time for each node***.  
  This definition remains the same as it was during the bridge problem.
* **Lowest time of insertion:**In this case, the current node refers to all its adjacent nodes ***except the parent and the visited nodes***and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another ‘***low***’ array.  
  ***The difference in finding the lowest time of insertion in this problem is that in the bridgealgorithm, we only excluded the parent node but in this algorithm, we are excluding the visited nodes along with the parent node.***

***The logical modification of the DFS algorithm is discussed below***:

To find out the bridges in the bridge problem, we checked inside the DFS, if there exists any alternative path from the adjacent node to the current node.  
But here we cannot do so as in this case, we are trying to remove the current node along with all the edges linked to it. For that reason, here we will check if there exists any path from the adjacent node to the previous node of the current node. ***In addition to that***, we must ensure that the current node we are trying to remove must not be the starting node.

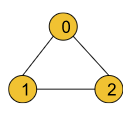
The check conditions for this case will change like the following:  
if(low[it] > tin[node])  converts to if(low[it] >= tin[node] && parent  != -1)

For the starting node, we will apply different logic.

**The logic for the starting node:**

If the node is a starting point we will check the number of children of the node. If the starting node has more than 1 child(The children must not be connected), it will definitely be one of the articulation points.

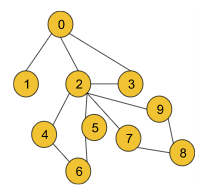
To find the number of children, we will generally count the number of adjacent nodes. But there is a point to notice. In the following graph, the starting node 0 has two adjacent nodes, but it is not an articulation point.



To avoid this edge case, we will increment the number of children only if the adjacent node is not previously visited(*i.e. child++ will be inside the not visited if statement*).

**We can get a single node as an articulation point multiple times:**

If we carefully observe, we can easily notice that we can get a single node as the articulation point multiple times. For example, consider the following graph:



While checking for node 2, we will get the node as the articulation point once for the first component that contains nodes 4, 5, and 6 and we will again get the same node 2 for the second component that includes the nodes 7, 8, and 9.

***To avoid the storing of duplicate nodes, we will store the nodes in a hash array(i.e. mark array used in the code) instead of directly inserting them in a simple array.***

**Approach:**

The algorithm steps are as follows:

* First, we need to create the adjacency list for the given graph from the edge information(***If not already given***). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node. The timer may be initialized to 0 or 1 accordingly.
* Then we will perform DFS for each component. For each component, the starting node will carry -1 as its parent.
  + Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. We will declare a child variable to implement the logic for starting node.
  + Now, it’s time to visit the adjacent nodes.
    - ***If the adjacent node is the parent itself***, we will just continue to the next node.
    - ***If the adjacent node is not visited***, we will call DFS for the adjacent node with the current node as the parent.  
      After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum.  
      Now, we will check if the lowest time of insertion of the adjacent node is greater or equal to the time of insertion of the current node and also ensure that the current node is not the starting node(checking parent not equal -1).  
      If the condition matches, then we will mark the current node in our hash array as one of our answers as it is one of the articulation points of the graph.  
      Then we will increment the child variable by 1.
    - ***If the adjacent node is visited***, we will just compare the lowest time of insertion of the current node and the time of insertion of the adjacent node and take the minimum.
  + Finally, we will check if the child value is greater than 1 and if the current node is the starting node. If it is then we will keep the starting node marked in our hash array as the starting node is also an articulation point in this case.
* Finally, our answer array will store all the bridges.

*.*

**Solution:**

import java.io.\*;

import java.util.\*;

class Solution {

private int timer = 1;

private void dfs(int node, int parent, int[] vis,

int tin[], int low[], int[] mark,

ArrayList<ArrayList<Integer>> adj) {

vis[node] = 1;

tin[node] = low[node] = timer;

timer++;

int child = 0;

for (Integer it : adj.get(node)) {

if (it == parent) continue;

if (vis[it] == 0) {

dfs(it, node, vis, tin, low, mark, adj);

low[node] = Math.min(low[node], low[it]);

*// node --- it*

if (low[it] >= tin[node] && parent != -1) {

mark[node] = 1;

}

child++;

} else {

low[node] = Math.min(low[node], tin[it]);

}

}

if (child > 1 && parent == -1) {

mark[node] = 1;

}

}

*//Function to return Breadth First Traversal of given graph.*

public ArrayList<Integer> articulationPoints(int n,

ArrayList<ArrayList<Integer>> adj) {

int[] vis = new int[n];

int[] tin = new int[n];

int[] low = new int[n];

int[] mark = new int[n];

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

if (vis[i] == 0) {

dfs(i, -1, vis, tin, low, mark, adj);

}

}

ArrayList<Integer> ans = new ArrayList<>();

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

if (mark[i] == 1) {

ans.add(i);

}

}

if (ans.size() == 0) {

ans.add(-1);

}

return ans;

}

}

class Main {

public static void main (String[] args) {

int n = 5;

int[][] edges = {

{0, 1}, {1, 4},

{2, 4}, {2, 3}, {3, 4}

};

ArrayList<ArrayList<Integer>> adj = new ArrayList<>();

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

adj.add(new ArrayList<Integer>());

}

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

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

adj.get(u).add(v);

adj.get(v).add(u);

}

Solution obj = new Solution();

ArrayList<Integer> nodes = obj.articulationPoints(n, adj);

int size = nodes.size();

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

int node = nodes.get(i);

System.out.print(node + " ");

}

System.out.println("");

}

}