1. Loop Detection

```
import java.util.Scanner;

class node {
    int data;
    node next;


    node(int data) {
        this.data = data;
        this.next = null;
    }
}

public class Main {
    static node head = null;
    static node ptr;


    static void addnode(int data) {
        node newnode = new node(data);
        if (head == null) {
            head = newnode;
            ptr = head;
        } else {
            ptr.next = newnode;
            ptr = ptr.next;
        }
    }


    static boolean detectloop(node head) {
        node slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) return true;
        }
        return false;
    }
```

```
    static void printll(){
        node temp = head;
        while(temp != null){
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }


    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        addnode(20);
        addnode(4);
        addnode(15);
        addnode(10);
        addnode(0);
        if (head != null && head.next != null && head.next.next != null){
            ptr.next = head; // points the last node to the head to create the loop
        }



        boolean ans = detectloop(head);
        System.out.println("Has loop: " + ans);
        printll();
        sc.close();
    }
}
```

SHORTCUT:

```
import java.util.*;

public class Main{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for(int i = 0; i<n; i++){
            arr[i] = sc.nextInt();
        }
        Set<Integer> seen = new HashSet<>();
        boolean flag = false;
        for (int num : arr) {
            if (!seen.add(num)) {
```

```
            flag = true;
            break;
          }
        }
        if(flag) System.out.println("Loop detected");
        else System.out.println("No loop");
    }
}
```

2. Sort the bitonic DLL

```
import java.util.*;

public class Main{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for(int i = 0; i<n; i++) arr[i] = sc.nextInt();
        Arrays.sort(arr);
        for(int i = 0; i<n; i++) System.out.print(arr[i] + " ");
      }
    }
```

3. Merge sort for DLL

4. Sort without extra Space

5. Segregate even & odd nodes in a LLL

```
import java.util.*;

public class Main{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for(int i = 0; i<n; i++) arr[i] = sc.nextInt();

        LinkedList<Integer> res = new LinkedList<>();
        for(int num : arr){
            if(num%2 == 0) res.add(num);
```

```
        }
        for(int num : arr){
            if(num%2 != 0) res.add(num);
        }
        for(int i : res) System.out.print( i + " ");
    }
}
```

6. Minimum Stack

```
import java.util.*;

public class Main{
    static Stack<Integer> mainStack = new Stack<>();
    static Stack<Integer> minStack = new Stack<>();
    public static void push(int x){
        mainStack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    public static void pop(){
        if (!mainStack.isEmpty()) {
            int poppedValue = mainStack.pop();
            if (!minStack.isEmpty() && poppedValue == minStack.peek()) {
                minStack.pop();
            }
        }
    }

    static public int top() {
        if (!mainStack.isEmpty())
            return mainStack.peek();
        return -1; // or throw an error
    }

    static public int getMin() {
        if (!minStack.isEmpty())
            return minStack.peek();
        return -1; // or throw an error
    }

    public static void main(String args[]){
        push(3);
        push(5);
        push(2);
        push(1);
```

```
        System.out.println("Top of stack: " + top());      // Output: 1
        System.out.println("Min value: " + getMin());  // Output: 1

        pop();
        System.out.println("Top of stack: " + top());      // Output: 2
        System.out.println("Min value: " + getMin());  // Output: 2

    }
}
```

7. The Celebrity problem

```
import java.util.*;

public class Main{
    public static int findCelebrity(int[][] m, int n){
        int candidate = 0;
        for (int i = 1; i < n; i++) {
            if (m[candidate][i] == 1) {
                candidate = i;
            }
        }
        for (int i = 0; i < n; i++) {
            if (i != candidate && (m[candidate][i] == 1 || m[i][candidate] == 0)) {
                return -1;
            }
        }
        return candidate;
    }
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] m = new int[n][n];
        for(int i = 0; i<n; i++){
            for(int j = 0; j<n; j++){
                m[i][j] = sc.nextInt();
            }
        }

        int celebid = findCelebrity(m, n);
        if (celebid != -1) System.out.println("Celebrity id: " + celebid);
        else System.out.println("No Celebrity");
    }
}
```

8. Iterative Tower of Hanoi

```
import java.util.*;
public class Main{
    public static void hanoi(int n, char src, char aux, char dest){
        if(n == 1){
            System.out.println("Move disk from" + src + "-> " + dest);
            return;
        }
        hanoi(n-1, src,dest,aux);
        hanoi(1,src,aux,dest);
        hanoi(n-1,aux,src,dest);
    }
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        hanoi(n,'A','B','C');
    }
}
```

9. Stock Span problem

```
import java.util.*;

public class Main{
    public static void main(String args[]){
        int[] stock = {100, 80, 60, 70, 60, 75, 85};
        int n = stock.length;
        int[] span = new int[n];

        for(int i = 0; i<n; i++){
            span[i] = 1;
            for(int j = i-1; j>=0; j--){
                if(stock[i] >= stock[j]){
                    span[i]++;
                }
                else{
                    break;
                }
            }
        }

        for(int i = 0; i<n; i++){
            System.out.print(span[i] + " ");
```

```
        }
    }
}
```

10. Priority Queue using DLL

```
import java.util.*;
public class Main{

    List<PriorityItem> items = new ArrayList<>();

    public void enqueue(String value, int priority) {
        PriorityItem newItem = new PriorityItem(value, priority);
        for (int i = 0; i < items.size(); i++) {
            if (newItem.priority > items.get(i).priority) {
                break;
            }
        }
        items.add(i, newItem); // adds in sorted position
    }

    public String dequeue() {
        if (isEmpty()) {
            return null; // Or throw an exception
        }
        return items.remove(0).value; // Remove and return the item at the front
    }

    public String peek() {
        if (isEmpty()) {
            return null;
        }
        return items.get(0).value; // return element at front
    }

    public boolean isEmpty() {
        return items.isEmpty();
    }

    private class PriorityItem {
        String value;
        int priority;

        public PriorityItem(String value, int priority) {
            this.value = value;
            this.priority = priority;
        }
    }

    public static void main(String[] args) {
        Main pq = new Main(); // create instance of the Main class
```

```
        pq.enqueue("Task C", 3);
        pq.enqueue("Task A", 1);
        pq.enqueue("Task B", 2);
        pq.enqueue("Task D", 1);

        System.out.println("Peek: " + pq.peek());   // Output: Task A
        System.out.println("Dequeue: " + pq.dequeue()); // Output: Task A
        System.out.println("Dequeue: " + pq.dequeue()); // Output: Task D
        System.out.println("Dequeue: " + pq.dequeue()); // Output: Task B
        System.out.println("Peek: " + pq.peek());   // Output: Task C
    }
}
```

11. Max Sliding Window

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        int n = nums.length;
        int[] result = new int[n - k + 1];
        for (int i = 0; i <= n - k; i++) {
            int max = Integer.MIN_VALUE;
            for (int j = i; j < i + k; j++) {
                max = Math.max(max, nums[j]);
            }
            result[i] = max;
        }
        System.out.println("Maximums in sliding window : " + Arrays.toString(result));
    }
}
```

12. Stack permutations

```
import java.util.Stack;

public class Main {

    public static boolean isStackPermutation(int[] original, int[] target) {
        Stack<Integer> stack = new Stack<>();
        int j = 0;
        for (int element : original) {
            stack.push(element);
```

```
            while (!stack.isEmpty() && j < target.length && stack.peek() == target[j]) {
                stack.pop();
                j++;
            }
        }
        return j == target.length && stack.isEmpty();
    }

    public static void main(String[] args) {
        int[] original = {1, 2, 3};
        int[] target = {2, 1, 3};
        System.out.println("Is it a stack permutation? " + isStackPermutation(original, target));
    }
}
```

1. **Recover BST.**

```java
import java.util.*;

//Tree Structure.
class TreeNode{
    int val;
    TreeNode left , right;
    TreeNode(int x){
        val = x;
    }
}

public class Main{

    public TreeNode buildtree_levelorder(String inp){
        String nodes[] = inp.split(",");
        if(nodes.length == 0 || nodes[0].equals("null")) return null;

        Queue<TreeNode> q = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
        q.offer(root);

        int i = 1;
        while(i < nodes.length){
            TreeNode curr = q.poll();
            if(!nodes[i].equals("null")){
                //first add the left onto the tree.
                curr.left = new TreeNode(Integer.parseInt(nodes[i]));
                q.offer(curr.left);
            }
            i++;

            if(i < nodes.length && !nodes[i].equals("null")){
                curr.right = new TreeNode(Integer.parseInt(nodes[i]));
                q.offer(curr.right);
            }
            i++;
        }
        return root;
    }

    //get the inorder way of it.
```

```
    public int[] inorder(TreeNode root){
        List<Integer> list = new ArrayList<>();
        inorder_helper(root,list);
        return list.stream().mapToInt(i -> i).toArray();
    }

    //get the inorder from the given Tree that we constructed rn.
    private void inorder_helper(TreeNode root , List<Integer> list){
        if(root == null){
            return;
        }

        inorder_helper(root.left , list);
        list.add(root.val);
        inorder_helper(root.right , list);
    }

    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        //the string of the level order will be given as the input.
        Main rec = new Main();
        String s = sc.nextLine();
        //have a Builder Function to Build the given Tree.
        TreeNode root = rec.buildtree_levelorder(s);
        //get the inorder to recover the BST.
        int[] inorder = rec.inorder(root);
        Arrays.sort(inorder);
        for(int i = 0; i < inorder.length ; i++){
            System.out.print(inorder[i] + " ");
        }
        //recover the proper BST from it.
        //TreeNode bstroot = rec.recoverbst(inorder);
        //rec.printInorder(bstroot);
    }
}
```

(Just to directly print it from the Obtained Ouput of the given Inorder Traversals).

2. **Views of The Tree.**
   (The Input Order will be Level Ordered)

```java
import java.util.*;

class TreeNode{
    int val;
    TreeNode left , right;
    TreeNode(int x){
        val = x;
    }
}

public class Main{
    //helped in Building Tree
    public TreeNode build_tree(String inp){
        String[] nodes = inp.split(",");
        if(nodes.length == 0 || nodes[0].equals("null")) return null;

        Queue<TreeNode> queue = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
        queue.offer(root);

        int i = 1;
        while(i < nodes.length){
            TreeNode curr = queue.poll();
            if(!nodes[i].equals("null")){
                //add in the left.
                curr.left = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(curr.left);
            }
            i++;

            if(i < nodes.length && !nodes[i].equals("null")){
                //add in the right;
                curr.right = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(curr.right);
            }
            i++;
        }
        return root;
    }
```

```java
    public void top_view(TreeNode root){
        if(root == null) return ;
        Map<Integer,Integer> map = new TreeMap<>();
        Queue<Pair<TreeNode , Integer>> queue = new LinkedList<>();

        //we add in the map only if its not there hence preserves the first
obatined case.
        queue.offer(new Pair<>(root , 0));
        //left --> -1 and right --> +1
        while(!queue.isEmpty()){
            Pair<TreeNode , Integer> curr = queue.poll();
            TreeNode node = curr.getKey();
            int dist = curr.getValue();

            //since we need to sort it based on the L->R the Key will be
the distance.
            if(!map.containsKey(dist)){
                map.put(dist,node.val);
            }

            //check for left and right and put it inside.
            if(node.left != null){
                queue.offer(new Pair<>(node.left , dist - 1));
            }
            if(node.right != null){
                queue.offer(new Pair<>(node.right , dist + 1));
            }
        }

        //print the Keys as they will be sorted in order.
        System.out.println("Top View:");
        for(Integer val : map.values()){
            System.out.print(val + " ");
        }
        System.out.println();
    }


    public void bottom_view(TreeNode root){
        if(root == null) return ;
        Map<Integer,Integer> mp = new TreeMap<>();
        Queue<Pair<TreeNode,Integer>> queue = new LinkedList<>();
```

```
        queue.offer(new Pair<>(root , 0));
        while(!queue.isEmpty()){
            Pair<TreeNode,Integer> curr = queue.poll();
            TreeNode node = curr.getKey();
            int dist = curr.getValue();

            //add it in until the very end.
            mp.put(dist , node.val);

            if(node.left != null){
                queue.offer(new Pair<>(node.left , dist - 1));
            }
            if(node.right != null){
                queue.offer(new Pair<>(node.right , dist + 1));
            }
        }

        System.out.println("Bottom View");
        for(Integer val : mp.values()){
            System.out.print(val + " ");
        }
        System.out.println();
    }

    public static void main(String[] args){
        Main tree = new Main();
        Scanner sc = new Scanner(System.in);
        String inp = sc.nextLine();
        //build the tree.
        TreeNode root = tree.build_tree(inp);
        //get the top and the bottom view as well.
        tree.top_view(root);
        tree.bottom_view(root);
    }
}

//have the Pair Class as well
class Pair<K,V>{
    private K key;
    private V value;

    public Pair(K key , V value){
        this.key = key;
```

```
        this.value = value;
    }

    public K getKey(){
        return key;
    }

    public V getValue(){
        return value;
    }
}
```

Left View :

```
// Left view function
    public void left_view(TreeNode root) {
        // Base case
        if (root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();
        List<Integer> res = new ArrayList<>();
        queue.offer(root); // Start with root node

        // Perform BFS
        while (!queue.isEmpty()) {
            int n = queue.size();

            for (int i = 0; i < n; i++) {
                TreeNode curr = queue.poll();

                // The first node at each level is part of the left view
                if (i == 0) {
                    res.add(curr.val);
                }

                // Add left and right children to queue
                if (curr.left != null) {
                    queue.offer(curr.left);
                }
                if (curr.right != null) {
                    queue.offer(curr.right);
                }
```

```
            }
        }

        // Print left view
        for (int i = 0; i < res.size(); i++) {
            if (i != 0) System.out.print(" "); // Add space between numbers
            System.out.print(res.get(i));
        }
        System.out.println(); // Move to next line after printing all left
view nodes
    }
```

RIght View :

```
public void right_view(TreeNode root) {
        // Base case
        if (root == null) return;

        Queue<TreeNode> queue = new LinkedList<>();
        List<Integer> res = new ArrayList<>();
        queue.offer(root); // Start with root node

        // Perform BFS
        while (!queue.isEmpty()) {
            int n = queue.size();

            for (int i = 0; i < n; i++) {
                TreeNode curr = queue.poll();

                // The first node at each level is part of the right view
                if (i == n-1) {
                    res.add(curr.val);
                }

                // Add left and right children to queue
                if (curr.left != null) {
                    queue.offer(curr.left);
                }
                if (curr.right != null) {
                    queue.offer(curr.right);
                }
            }
```

```
        }

        // Print left view
        for (int i = 0; i < res.size(); i++) {
            if (i != 0) System.out.print(" "); // Add space between numbers
            System.out.print(res.get(i));
        }
        System.out.println(); // Move to next line after printing all left
view nodes
    }
```

3. **Boundary Traversal :**
   First Go throught the Left View Nodes, then the Leaf Nodes and then the RIght side nodes have them in a ArrayList for Dynamic Entry so that it will be better to print it at the very end

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left, right;

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

public class Main {

    // Function to print the left boundary (excluding leaf nodes)
    public void printLeftBoundary(TreeNode root, List<Integer> res) {
        while (root != null) {
            if (root.left != null) {
                res.add(root.val);
                root = root.left;
            } else if (root.right != null) {
                res.add(root.val);
                root = root.right;
            } else {
                break;
```

```
            }
        }
    }

    // Function to print the right boundary (excluding leaf nodes) in
reverse order
    public void printRightBoundary(TreeNode root, List<Integer> res) {
        Stack<Integer> stack = new Stack<>();
        while (root != null) {
            if (root.right != null) {
                stack.push(root.val);
                root = root.right;
            } else if (root.left != null) {
                stack.push(root.val);
                root = root.left;
            } else {
                break;
            }
        }

        // Print the right boundary in reverse order
        while (!stack.isEmpty()) {
            res.add(stack.pop());
        }
    }

    // Function to print all the leaf nodes
    public void printLeaves(TreeNode root, List<Integer> res) {
        if (root == null) return;

        // If it's a leaf node, add it to the result
        if (root.left == null && root.right == null) {
            res.add(root.val);
            return;
        }

        // Recur for left and right children
        printLeaves(root.left, res);
        printLeaves(root.right, res);
    }

    // Function to perform the boundary traversal
    public void boundaryTraversal(TreeNode root) {
```

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

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

        // Add root to the result (this will be the first node in the
boundary)
        res.add(root.val);

        // 1. Print left boundary (excluding the leaf nodes)
        printLeftBoundary(root.left, res);

        // 2. Print all the leaf nodes
        printLeaves(root, res);

        // 3. Print right boundary (excluding the leaf nodes)
        printRightBoundary(root.right, res);

        // Print the result list
        for (int i = 0; i < res.size(); i++) {
            if (i != 0) System.out.print(" ");  // Space between elements
            System.out.print(res.get(i));
        }
        System.out.println();
    }

    // Function to build a tree from level order input
    public TreeNode build_tree(String inp) {
        String[] nodes = inp.split(",");
        if (nodes.length == 0 || nodes[0].equals("null")) return null;

        Queue<TreeNode> queue = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
        queue.offer(root);

        int i = 1;
        while (i < nodes.length) {
            TreeNode curr = queue.poll();

            // Add left child
            if (!nodes[i].equals("null")) {
                curr.left = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(curr.left);
            }
```

```
            i++;

            // Add right child
            if (i < nodes.length && !nodes[i].equals("null")) {
                curr.right = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(curr.right);
            }
            i++;
        }
        return root;
    }

    public static void main(String[] args) {
        Main tree = new Main();
        Scanner sc = new Scanner(System.in);

        // Take input as a string (level-order format)
        String inp = sc.nextLine();

        // Build the tree
        TreeNode root = tree.build_tree(inp);

        // Perform boundary traversal
        tree.boundaryTraversal(root);

        sc.close(); // Close the scanner to avoid resource leak
    }
}
```

4. **BFS , DFS (based on what kinda traversal they are - pre, in , post)**

BFS :

```
 public void bfs(TreeNode root , List<Integer> res){
        if(root == null) return;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);

        while(!q.isEmpty()){
            int n = q.size();
            for(int i = 0 ; i < n ; i++){
```

```
                TreeNode curr = q.poll();
                res.add(curr.val);
                if(curr.left != null){
                    q.offer(curr.left);
                }
                if(curr.right != null){
                    q.offer(curr.right);
                }
            }
        }
    }
```

5. **Vertical Traversal :**

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left, right;

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

// Custom Pair class to hold TreeNode and its horizontal distance
class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
```

```java
}

public class Main {

    // Build tree from the input string (Level-order format)
    public TreeNode build_tree(String s) {
        String[] nodes = s.split(",");
        if (nodes.length == 0 || nodes[0].equals("null")) return null;

        Queue<TreeNode> queue = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
        queue.offer(root);

        int i = 1;
        while (i < nodes.length) {
            TreeNode curr = queue.poll();
            if (!nodes[i].equals("null")) {
                curr.left = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(curr.left);
            }
            i++;

            if (i < nodes.length && !nodes[i].equals("null")) {
                curr.right = new TreeNode(Integer.parseInt(nodes[i]));
                queue.offer(curr.right);
            }
            i++;
        }
        return root;
    }

    // Vertical Traversal using BFS with a custom Pair class
    public void vertical_traversal(TreeNode root) {
        if (root == null) return;

        Map<Integer, List<Integer>> map = new TreeMap<>(); // To store
nodes in vertical order
        Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();
        queue.offer(new Pair<>(root, 0)); // Root starts at HD = 0

        while (!queue.isEmpty()) {
            int n = queue.size();
            Map<Integer, List<Integer>> levelMap = new TreeMap<>(); //
```

```
Temporary map for each level

            for (int i = 0; i < n; i++) {
                Pair<TreeNode, Integer> pair = queue.poll();
                TreeNode curr = pair.getKey();
                Integer dist = pair.getValue();

                // Add the node value to the corresponding horizontal
distance
                levelMap.putIfAbsent(dist, new ArrayList<>());
                levelMap.get(dist).add(curr.val);

                // Add the left and right children to the queue with
updated HD
                if (curr.left != null) {
                    queue.offer(new Pair<>(curr.left, dist - 1));
                }
                if (curr.right != null) {
                    queue.offer(new Pair<>(curr.right, dist + 1));
                }
            }

            // After processing the level, update the final map with nodes
at each HD
            for (Map.Entry<Integer, List<Integer>> entry :
levelMap.entrySet()) {
                map.putIfAbsent(entry.getKey(), new ArrayList<>());
                map.get(entry.getKey()).addAll(entry.getValue());
            }
        }

        // Print the vertical order of nodes
        for (Map.Entry<Integer, List<Integer>> entry : map.entrySet()) {
            System.out.println(entry.getValue());
        }
    }

    public static void main(String[] args) {
        Main tree = new Main();
        Scanner sc = new Scanner(System.in);

        // Read the input string for the tree
        String s = sc.nextLine();
```

```
        // Build the tree from the input string
        TreeNode root = tree.build_tree(s);

        // Perform vertical traversal and print the result
        tree.vertical_traversal(root);
    }
```

6. **Winner Tree :**

They will give n sorted arrays we need to basically merge all of them and sort it

- **We will be adding all of them into an single array and then sorting it.**
- **We will use Inbuilt Functions to add them and sort it (Use Collections Sort in case of a List and Arrays.sort in case of an Array).**

```
import java.util.*;

public class Main{
    public static void main(String[] args){
        Scanner sc  = new Scanner(System.in);
        int n = sc.nextInt(); //to get the number of the arrays that we
have.
        sc.nextLine();
        List<Integer> res = new ArrayList<>();
        while(n > 0){
            String s = sc.nextLine();
            String[] inp = s.split(",");
            for(int i = 0 ; i < inp.length ; i++){
                res.add(Integer.parseInt(inp[i]));
            }
            n--;
        }
        Collections.sort(res);
        for(int i = 0 ; i < res.size() ; i++){
            if(i != 0 ) System.out.print(" ");
            System.out.print(res.get(i));
        }
    }
}
```

7. **Heap Sort:**
   Basically get the input and then do this (Both the array and the List Type is added)

Array Method :

```java
//heap sort.
import java.util.*;

public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for(int i = 0 ; i < n ; i++){
            arr[i] = sc.nextInt();
        }
        Arrays.sort(arr);
        for(Integer num : arr){
            System.out.print(num + " ");
        }
        sc.close();
    }
}
```

8.Binomial Heap : Under the hood we are basically Using Prioirty queue :

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        sc.nextLine();

        PriorityQueue<Integer> pq = new PriorityQueue<>();
        while(n > 0) {
            String s = sc.nextLine();
            String[] parts = s.split(",");
            for(String part : parts) {
                pq.add(Integer.parseInt(part));
            }
            n--;
```

```
        }

        while(!pq.isEmpty()) {
            System.out.print(pq.poll() + " ");
        }
        sc.close();
    }
}
```

(Min heap)

(To Convert it into Max Heap :)

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        sc.nextLine();

        PriorityQueue<Integer> pq = new
PriorityQueue<>(Collections.reverseOrder());
        while(n > 0) {
            String s = sc.nextLine();
            String[] parts = s.split(",");
            for(String part : parts) {
                pq.add(Integer.parseInt(part));
            }
            n--;
        }

        while(!pq.isEmpty()) {
            System.out.print(pq.poll() + " ");
        }
        sc.close();
    }
}
```

10. Topological Sort : Using Kahns Algorithm :

import java.util.*;

```
//implementation of Kahns Algorithm.

public class Main{
    static int[] toposort(int V , List<List<Integer>> adj){
        //get an indegree vector.
        int[] indegree = new int[V];

        //fill in those based on adjacency lists.
        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);
            }
        }

        int[] topo = new int[V];
        int i = 0;
        while(!q.isEmpty()){
            int node = q.peek();
            q.remove();
            topo[i++] = node;

            //process the node.
            for(int it : adj.get(node)){
                indegree[it]--;
                if(indegree[it] == 0){
                    q.add(it);
                }
            }
        }
        return topo;
    }
    public static void main(String[] args){
```

```
        Scanner sc = new Scanner(System.in);
        int V = sc.nextInt();
        int E = sc.nextInt();

        List<List<Integer>> adj = new ArrayList<>();
        //add in the empty array list
        for(int i = 0 ; i < V ; i++){
            adj.add(new ArrayList<>());
        }

        //adding edges
        for(int i = 0 ; i < E ; i++){
            int u = sc.nextInt();
            int v = sc.nextInt();
            adj.get(u).add(v);
        }

        //adjaceny list is ready.
        int[] res = toposort(V,adj);
        for(int i = 0 ; i < V ; i++){
            System.out.print(res[i] + " ");
        }
    }
}
```

11. Bellman Ford Algorithm :

```
//BellMan Ford to calculate the distance
import java.util.*;

public class Main{
    static int[] bellman_ford(int V , List<List<Integer>> edges , int S){
        int[] dist = new int[V];

        //assign everything to Infinite now.
        for(int i = 0 ; i < V ; i++){
            dist[i] = (int)(1e8);
        }

        dist[S] = 0;
        for(int i = 0 ; i < V-1; i ++){
```

```java
            //get the edges data from this.
            for(List<Integer> it : edges){
                int u = it.get(0);
                int v = it.get(1);
                int w = it.get(2);
                if(dist[u] != (int)1e8 && dist[u] + w < dist[v]){
                    dist[v] = dist[u] + w;
                }
            }
        }
        //to relax the negative edges at the end.
        for(List<Integer> it : edges){
            int u = it.get(0);
            int v = it.get(1);
            int w = it.get(2);
            if(dist[u] != (int)1e8 && dist[u] + w < dist[v]){
                return new int[]{-1};
            }
        }
        return dist;

    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int V = sc.nextInt();
        int E = sc.nextInt();
        int S = sc.nextInt();

        List<List<Integer>> edges = new ArrayList<>();
        for(int i = 0 ; i < E ; i++){
            int u = sc.nextInt();
            int v = sc.nextInt();
            int w = sc.nextInt();
            edges.add(new ArrayList<>(Arrays.asList(u,v,w)));
        }

        int[] dist = bellman_ford(V,edges,S);
        for(int i = 0 ; i < V ; i++){
            System.out.print(dist[i] + " ");
        }
        sc.close();
    }
}
```

12. Dials also same :

```java
import java.util.*;

public class Main{
    static int[] bellman_ford(int V , List<List<Integer>> edges , int S){
        int[] dist = new int[V];

        // assign everything to Infinite now.
        for(int i = 0 ; i < V ; i++){
            dist[i] = Integer.MAX_VALUE;  // Fixed this line
        }

        dist[S] = 0;
        // Relax edges V-1 times
        for(int i = 0 ; i < V-1; i++){
            // get the edges data from this.
            for(List<Integer> it : edges){
                int u = it.get(0);
                int v = it.get(1);
                int w = it.get(2);
                if(dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]){
                    dist[v] = dist[u] + w;
                }
            }
        }

        // to relax the negative edges at the end.
        for(List<Integer> it : edges){
            int u = it.get(0);
            int v = it.get(1);
            int w = it.get(2);
            if(dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]){
                return new int[]{-1};  // If negative cycle detected
            }
        }

        return dist;
    }

    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int V = sc.nextInt();  // Number of vertices
        int E = sc.nextInt();  // Number of edges
```

```
        int S = sc.nextInt();  // Source vertex

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

        // Reading edges from input
        for(int i = 0 ; i < E ; i++){
            int u = sc.nextInt();
            int v = sc.nextInt();
            int w = sc.nextInt();
            edges.add(new ArrayList<>(Arrays.asList(u, v, w)));
        }

        int[] dist = bellman_ford(V, edges, S);

        // Print the distances from source
        for(int i = 0 ; i < V ; i++){
            if (dist[i] == Integer.MAX_VALUE) {
                System.out.print("INF ");  // unreachable node
            } else {
                System.out.print(dist[i] + " ");
            }
        }

        sc.close();
    }
}
```