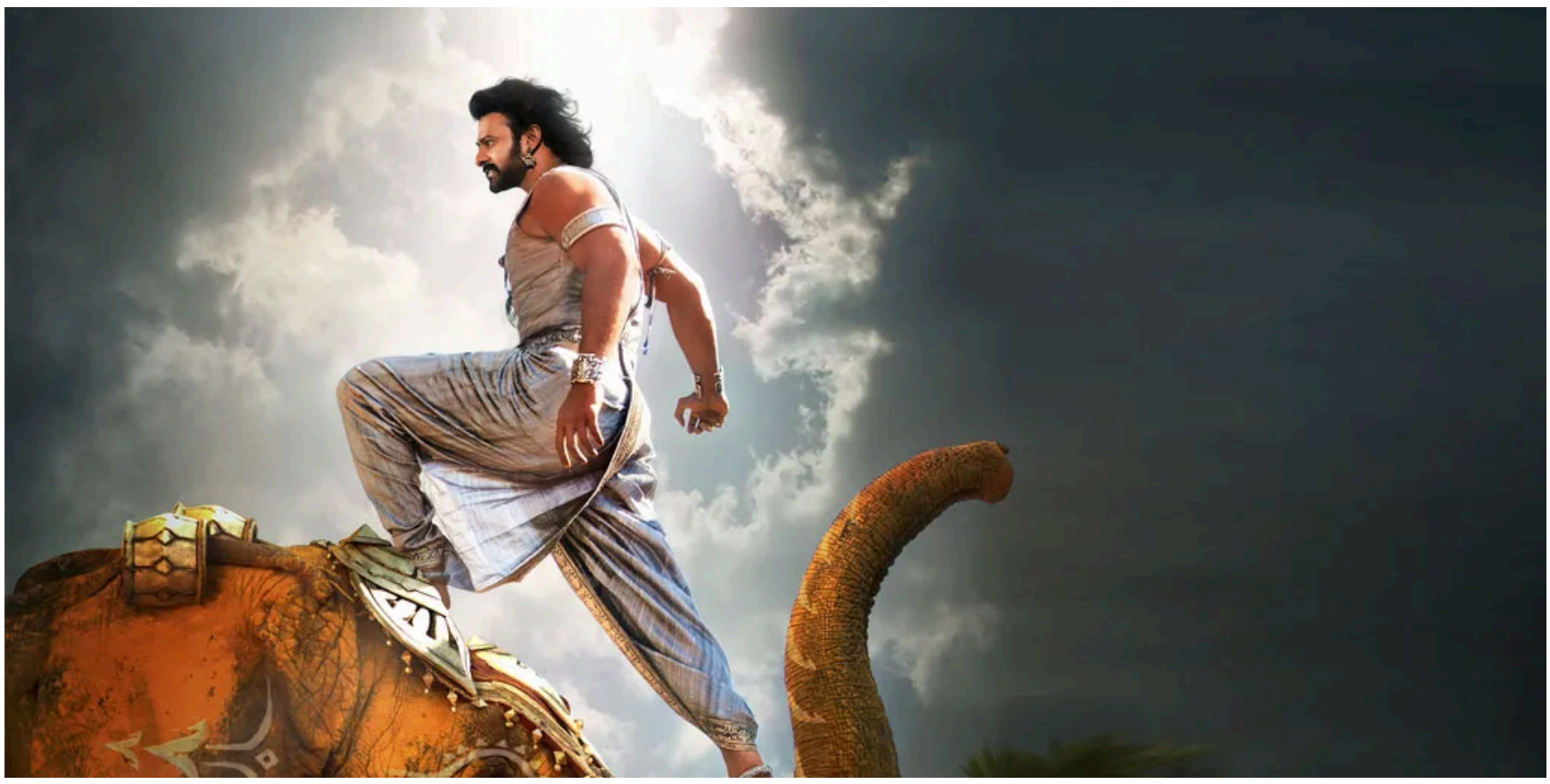

FAT

🔥 CAT 1

▼ Loop Detection

```
import java.util.Scanner;

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();
        }
        int target = sc.nextInt();
        // Check if target exists in the array
        boolean found = false;
        for (int num : arr) {
            if (num == target) {
                found = true;
                break;
            }
        }
        System.out.println(found);
    }
}
```

🔥
```
3
1 2 3
4
```

🔥 false

▼ Segregate Even & Odd nodes in a LL

```
import java.util.*;

class Main {
    public static void segregateEvenOddValues(LinkedList<Integer> list) {
        LinkedList<Integer> evenList = new LinkedList<>();
```

```
    LinkedList<Integer> oddList = new LinkedList<>();

    for (int i = 0; i < list.size(); i++) {
        if (list.get(i) % 2 == 0) {
            evenList.add(list.get(i));
        } else {
            oddList.add(list.get(i));
        }
    }

    list.clear();
    list.addAll(evenList);
    list.addAll(oddList);
}

public static void displayList(LinkedList<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.print(list.get(i) + "→");
    }
    System.out.println("null");
}

public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        LinkedList<Integer> list = new LinkedList<>();
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }
        displayList(list);
        segregateEvenOddValues(list);
        displayList(list);
    }
}
```

🔥
```
4
1 2 3 4
```

🔥
```
1→2→3→4→null
2→4→1→3→null
```

▼ Sort the Bitonic DLL

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }
        // Display the original list in DLL-like format
        displayList(list);
        // Sort the bitonic list
        Collections.sort(list);
        // Display the sorted list in DLL-like format
        displayList(list);
    }
    private static void displayList(List<Integer> list) {
        for (int i = 0; i < list.size(); i++) {
            System.out.print(list.get(i)+"⟷");
        }
        System.out.println("null");
    }
}
```

🔥
```
5
1 3 5 4 2
```

🔥
1⟷2⟷3⟷4⟷5⟷null

🔥

Sequence	Bitonic?	Reason
`1 2 3 4 5`	✅	Increasing only
`5 4 3 2 1`	✅	Decreasing only
`1 3 5 4 2`	✅	Inc → Dec (normal case)
`5 3 1 2 4`	❌	Dec → Inc → Not bitonic ❌

▼ Merge sort using DLL

```
import java.util.*;
class Main {
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        ArrayList<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(sw.nextInt());
        }
        display(list);
        Collections.sort(list);
        display(list);
    }

    // Method to display the list
    static void display(ArrayList<Integer> list) {
        for (int data : list) {
            System.out.print(data + "⟷");
        }
        System.out.println("null");
    }
}
```

🔥 5

3 1 4 2 5

🔥 1⟷2⟷3⟷4⟷5⟷null

▼ Minimum Stack

```
import java.util.*;
class Main{
    static Stack<Integer> st = new Stack<>();
    static Stack<Integer> mst = new Stack<>();
    static void push(int n){
        if(st.isEmpty()){
            st.push(n);
            mst.push(n);
        }
        else{
            st.push(n);
            if(n<=mst.peek()) mst.push(n);
        }
    }
    static void pop(){
        int ele=st.pop();
        if(ele==mst.peek()) mst.pop();
    }
    static void getmin(){
        if(mst.isEmpty()) System.out.print("Stack is Empty");
        else System.out.print(mst.peek());
    }
    public static void main(String ar[]){
        Scanner sw = new Scanner(System.in);
        int n=sw.nextInt();
        for(int i=0;i<n;i++) push(sw.nextInt());
        getmin();
```

```
    }
}
```

🔥
```
5
3 5 2 1 4
```

🔥
```
1
```

▼ The Celebrity Problem

```
import java.util.*;

public class Main {
    static int findCel(int n, int[][] mat) {
        int[] KnowMe = new int[n];
        int[] IKnow = new int[n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (mat[i][j] == 1) {
                    KnowMe[j]++;
                    IKnow[i]++;
                }
            }
        }
        for (int i = 0; i < n; i++) {
            if (KnowMe[i] == n - 1 && IKnow[i] == 0) {
                return i;
            }
        }
        return -1;
    }

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

        // Read number of people
        int n = sc.nextInt();
        int[][] matrix = new int[n][n];

        // Read the relationship matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }

        // Find and print the celebrity
        int id = findCel(n, matrix);
        if (id == -1) {
            System.out.println("No celebrity found");
        } else {
            System.out.println("celebrity is: " + id);
        }
    }
}
```

🔥
```
4
0 0 1 0
0 0 1 0
0 0 0 0
0 0 1 0
```

🔥
```
celebrity is: 2
```

▼ Towers of Hanoi

```
import java.util.Scanner;

class Main {
    static void moveDisks(int n, char from, char to, char aux) {
        if (n == 1) {
            System.out.println("The value 1 is moved from " + from + " to " + to);
            return;
```

```
        }
        moveDisks(n - 1, from, aux, to);
        System.out.println("The value " + n + " is moved from " + from + " to " + to);
        moveDisks(n - 1, aux, to, from);
    }

    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        char s = 'S', a = 'A', d = 'D';
        if (n % 2 == 0) {
            char temp = a;
            a = d;
            d = temp;
        }
        moveDisks(n, s, d, a);
    }
}
```

🔥 3

🔥 The disk 1 is moved from S to D
The disk 2 is moved from S to A
The disk 1 is moved from D to A
The disk 3 is moved from S to D
The disk 1 is moved from A to S
The disk 2 is moved from A to D
The disk 1 is moved from S to D

▼ Stock Span Problem

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] prices = new int[n];
        for (int i = 0; i < n; i++) {
            prices[i] = sc.nextInt();
        }
        int[] span = calculateSpan(prices, n);
        for (int value : span) {
            System.out.print(value + " ");
        }
    }
    public static int[] calculateSpan(int[] prices, int n) {
        int[] span = new int[n];

        // For each price, calculate its span
        for (int i = 0; i < n; i++) {
            int count = 1; // Default span for the current day
            for (int j = i - 1; j >= 0; j--) {
                if(prices[j] <= prices[i]) count++;
            }
            span[i] = count;
        }
        return span;
    }
}
```

🔥 5
10 4 5 90 120

🔥 1 1 2 4 5

▼ Priority Queue using DLL

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
```

```
        ArrayList<Integer> data = new ArrayList<>();
        ArrayList<Integer> priority = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            data.add(sw.nextInt());
            priority.add(sw.nextInt());
        }

        // Create a list of indexes 0 to n-1
        ArrayList<Integer> indexList = new ArrayList<>();
        for (int i = 0; i < n; i++) indexList.add(i);

        // Sort indexes based on priority values
        Collections.sort(indexList, (i, j) -> priority.get(i) - priority.get(j));

        // Print data and priority based on sorted indexes
        for (int i : indexList) {
            System.out.println(data.get(i) + " " + priority.get(i));
        }
    }
}
```

🔥
```
3
10 2
20 1
30 3
```

🔥
```
20 1
10 2
30 3
```

▼ Sort without extra Space

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        List<Integer> list = new ArrayList<>();

        // Input elements into the ArrayList
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }

        Collections.sort(list);

        // Print the sorted elements
        for (int num : list) {
            System.out.print(num + " ");
        }
    }
}
```

🔥
```
5
4 3 1 2 5
```

🔥
```
1 2 3 4 5
```

▼ Stack permutations

```
import java.util.*;

class Main {
    public static void main(String ar[]) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        Queue<Integer> q1 = new LinkedList<>();
        Queue<Integer> q2 = new LinkedList<>();

        for (int i = 0; i < n; i++) q1.add(sw.nextInt());
        for (int i = 0; i < n; i++) q2.add(sw.nextInt());
```

```
    Stack<Integer> st = new Stack<>();

    while (!q1.isEmpty()) {
        int ele = q1.poll();
        if (ele == q2.peek()) {
            q2.poll();
            while (!st.isEmpty()) {
                if (st.peek() == q2.peek()) {
                    st.pop();
                    q2.poll();
                } else {
                    break;
                }
            }
        } else {
            st.push(ele);
        }
    }

    if (q1.isEmpty() && st.isEmpty()) {
        System.out.print("Yes");
    } else {
        System.out.print("No");
    }
}
```

🔥 3
1 2 3
2 1 3

🔥 Yes

🔥 CAT 2

▼ Max Sliding Window

```
import java.util.*;

class Main {
    static void maxCal(int[] arr, int l, int r, int[] ret, int index) {
        int temp = Integer.MIN_VALUE;
        for (int i = l; i <= r; i++) {
            temp = Math.max(arr[i], temp);
        }
        ret[index] = temp;
    }

    static int[] maxWin(int[] arr, int k) {
        int n = arr.length;
        int[] ret = new int[n - k + 1];
        int l = 0, r = k - 1, index = 0;

        while (r < n) {
            maxCal(arr, l, r, ret, index);
            l++;
            r++;
            index++;
        }
        return ret;
    }

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

        int n = sc.nextInt();
        if (n <= 0) {
            System.out.println("0");
```

```
            return;
        }

        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        int k = sc.nextInt();
        if (k <= 0 || k > n) {
            System.out.println("0");
            return;
        }

        int[] res = maxWin(arr, k);
        for (int x : res) {
            System.out.print(x + " ");
        }

        sc.close();
    }
}
```

🔥
```
8
1 3 -1 -3 5 3 6 7
3
```

🔥
```
3 3 5 5 6 7
```

▼ Recover the BST

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int value) {
            data = value;
        }
    }

    public static Node createBST(String[] input) {
        if (input.length == 0 || input[0].equals("-1")) return null;

        Node root = new Node(Integer.parseInt(input[0]));
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);

        int i = 1;
        while (!queue.isEmpty() && i < input.length) {
            Node current = queue.poll();

            if (i < input.length) {
                String leftVal = input[i];
                if (!leftVal.equals("-1")) {
                    current.left = new Node(Integer.parseInt(leftVal));
                    queue.add(current.left);
                }
                i++;
            }

            if (i < input.length) {
                String rightVal = input[i];
                if (!rightVal.equals("-1")) {
                    current.right = new Node(Integer.parseInt(rightVal));
                    queue.add(current.right);
                }
                i++;
            }
        }
        return root;
    }
```

```
static Node first, middle, last, prev;

public static void correctBSTUtil(Node root) {
    if (root == null) return;

    correctBSTUtil(root.left);

    if (prev != null && root.data < prev.data) {
        if (first == null) {
            first = prev;
            middle = root;
        } else {
            last = root;
        }
    }

    prev = root;
    correctBSTUtil(root.right);
}

public static void correctBST(Node root) {
    first = middle = last = null;
    prev = new Node(Integer.MIN_VALUE);
    correctBSTUtil(root);

    if (first != null && last != null)
        swap(first, last);
    else if (first != null && middle != null)
        swap(first, middle);
}

public static void swap(Node a, Node b) {
    int temp = a.data;
    a.data = b.data;
    b.data = temp;
}

public static void printInorder(Node node) {
    if (node == null) return;
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = createBST(input);
    printInorder(root);  // Before fixing
    System.out.println();

    correctBST(root);    // Fix the tree

        printInorder(root);  // After fixing
        System.out.println();

        sc.close();
    }
}
```

🔥 10 5 8 2 20 -1 -1 -1 -1 -1 -1

🔥 2 5 20 10 8
2 5 8 10 20

▼ Right View

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int value) {
```

```
        data = value;
    }
}

static Node createBST(String[] input) {
    if (input.length == 0 || input[0].equals("-1")) return null;

    Queue<Node> queue = new LinkedList<>();
    Node root = new Node(Integer.parseInt(input[0]));
    queue.add(root);

    int i = 1;
    while (!queue.isEmpty() && i < input.length) {
        Node cur = queue.poll();

        if (i < input.length) {
            String lV = input[i];
            if (!lV.equals("-1")) {
                cur.left = new Node(Integer.parseInt(lV));
                queue.add(cur.left);
            }
            i++;
        }

        if (i < input.length) {
            String rV = input[i];
            if (!rV.equals("-1")) {
                cur.right = new Node(Integer.parseInt(rV));
                queue.add(cur.right);
            }
            i++;
        }
    }

    return root;
}

public static void rightView(Node root, List<Integer> view, int level) {
    if (root == null) return;
    if (level == view.size()) view.add(root.data);
    rightView(root.right, view, level + 1);
    rightView(root.left, view, level + 1);
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = createBST(input);

    List<Integer> view = new ArrayList<>();
    rightView(root, view, 0);

        for (int val : view) {
            System.out.print(val + " ");
        }
        System.out.println();
        sc.close();
    }
}
```

🔥 1 2 3 4 -1 -1 5 -1 -1 -1 -1

🔥 1 3 5

▼ Left View

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int value) {
            data = value;
```

```
    }
}

static Node createBST(String[] input) {
    if (input.length == 0 || input[0].equals("-1")) return null;

    Queue<Node> queue = new LinkedList<>();
    Node root = new Node(Integer.parseInt(input[0]));
    queue.add(root);

    int i = 1;
    while (!queue.isEmpty() && i < input.length) {
        Node cur = queue.poll();

        if (i < input.length) {
            String lV = input[i];
            if (!lV.equals("-1")) {
                cur.left = new Node(Integer.parseInt(lV));
                queue.add(cur.left);
            }
            i++;
        }

        if (i < input.length) {
            String rV = input[i];
            if (!rV.equals("-1")) {
                cur.right = new Node(Integer.parseInt(rV));
                queue.add(cur.right);
            }
            i++;
        }
    }

    return root;
}

public static void leftView(Node root, List<Integer> view, int level) {
    if (root == null) return;
    if (level == view.size()) view.add(root.data);
    leftView(root.left, view, level + 1);   // 👈 Left first
    leftView(root.right, view, level + 1);
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = createBST(input);

    List<Integer> view = new ArrayList<>();
    leftView(root, view, 0);

    for (int val : view) {
        System.out.print(val + " ");
    }
    System.out.println();
    sc.close();
  }
}
```

🔥 1 2 3 4 -1 -1 5 -1 -1 -1 -1

🔥 1 2 4

▼ Vertical Order Traversal

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int val) {
            data = val;
        }
```

```
    }

    static class Pair {
        Node node;
        int hd;
        Pair(Node node, int hd) {
            this.node = node;
            this.hd = hd;
        }
    }

    // Build tree from level-order input with "N" as null
    public static Node buildTree(String[] input) {
        if (input.length == 0 || input[0].equals("N")) return null;

        Node root = new Node(Integer.parseInt(input[0]));
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);

        int i = 1;
        while (!queue.isEmpty() && i < input.length) {
            Node current = queue.poll();

            // Process left child
            if (i < input.length) {
                String leftVal = input[i];
                if (!leftVal.equals("N")) {
                    current.left = new Node(Integer.parseInt(leftVal));
                    queue.add(current.left);
                }
                i = i + 1;
            }

            // Process right child
            if (i < input.length) {
                String rightVal = input[i];
                if (!rightVal.equals("N")) {
                    current.right = new Node(Integer.parseInt(rightVal));
                    queue.add(current.right);
                }
                i = i + 1;
            }
        }
        return root;
    }

    public static void verticalOrder(Node root) {
        if (root == null) return;

        TreeMap<Integer, ArrayList<Integer>> map = new TreeMap<>();
        Queue<Pair> queue = new LinkedList<>();
        queue.add(new Pair(root, 0));

        while (!queue.isEmpty()) {
            Pair temp = queue.poll();
            Node curr = temp.node;
            int hd = temp.hd;

            map.putIfAbsent(hd, new ArrayList<>());
            map.get(hd).add(curr.data);

            if (curr.left != null)
                queue.add(new Pair(curr.left, hd - 1));
            if (curr.right != null)
                queue.add(new Pair(curr.right, hd + 1));
        }

        for (ArrayList<Integer> list : map.values()) {
            for (int val : list)
                System.out.print(val + " ");
        }
    }

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

```
        String[] input = sc.nextLine().split(" ");
        Node root = buildTree(input);
        verticalOrder(root);
        sc.close();
    }
}
```

🔥 1 2 3 N N 4 5

🔥 2 1 4 3 5

▼ Top View

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int val) {
            data = val;
        }
    }

    static class Pair {
        Node node;
        int hd;
        Pair(Node node, int hd) {
            this.node = node;
            this.hd = hd;
        }
    }

    public static Node buildTree(String[] input) {
        if (input.length == 0 || input[0].equals("N")) return null;

        Node root = new Node(Integer.parseInt(input[0]));
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);

        int i = 1;
        while (!queue.isEmpty() && i < input.length) {
            Node current = queue.poll();

            if (i < input.length) {
                String leftVal = input[i];
                if (!leftVal.equals("N")) {
                    current.left = new Node(Integer.parseInt(leftVal));
                    queue.add(current.left);
                }
                i = i + 1;
            }

            if (i < input.length) {
                String rightVal = input[i];
                if (!rightVal.equals("N")) {
                    current.right = new Node(Integer.parseInt(rightVal));
                    queue.add(current.right);
                }
                i = i + 1;
            }
        }
        return root;
    }

    // Return Top View as list
    public static List<Integer> topView(Node root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        TreeMap<Integer, Integer> map = new TreeMap<>();
        Queue<Pair> queue = new LinkedList<>();
        queue.add(new Pair(root, 0));
```

```
    while (!queue.isEmpty()) {
        Pair temp = queue.poll();
        int hd = temp.hd;
        Node curr = temp.node;

        if (!map.containsKey(hd)) {
            map.put(hd, curr.data);
        }

        if (curr.left != null) queue.add(new Pair(curr.left, hd - 1));
        if (curr.right != null) queue.add(new Pair(curr.right, hd + 1));
    }

    result.addAll(map.values());
    return result;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = buildTree(input);
    System.out.println(topView(root));
    sc.close();
    }
}
```

🔥 1 2 3 N N 4 5

🔥 [2, 1, 3, 5]

▼ Bottom View

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int val) {
            data = val;
        }
    }

    static class Pair {
        Node node;
        int hd;
        Pair(Node node, int hd) {
            this.node = node;
            this.hd = hd;
        }
    }

// Build tree from level-order input with "N" as null
public static Node buildTree(String[] input) {
    if (input.length == 0 || input[0].equals("N")) return null;

    Node root = new Node(Integer.parseInt(input[0]));
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);

    int i = 1;
    while (!queue.isEmpty() && i < input.length) {
        Node current = queue.poll();

        if (i < input.length) {
            String leftVal = input[i];
            if (!leftVal.equals("N")) {
                current.left = new Node(Integer.parseInt(leftVal));
                queue.add(current.left);
            }
            i = i + 1;
        }

        if (i < input.length) {
```

```
            String rightVal = input[i];
            if (!rightVal.equals("N")) {
                current.right = new Node(Integer.parseInt(rightVal));
                queue.add(current.right);
            }
            i = i + 1;
        }
    }
    return root;
}

// Bottom view logic
public static List<Integer> bottomView(Node root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    TreeMap<Integer, Integer> map = new TreeMap<>();
    Queue<Pair> queue = new LinkedList<>();
    queue.add(new Pair(root, 0));

    while (!queue.isEmpty()) {
        Pair temp = queue.poll();
        int hd = temp.hd;
        Node curr = temp.node;

        // Overwrite at this horizontal distance to get bottom view
        map.put(hd, curr.data);

        if (curr.left != null) queue.add(new Pair(curr.left, hd - 1));
        if (curr.right != null) queue.add(new Pair(curr.right, hd + 1));
    }

    result.addAll(map.values());
    return result;
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = buildTree(input);
    System.out.println(bottomView(root));
    sc.close();
  }
}
```

🔥 20 8 22 5 3 N 25 N N 10 14

🔥 [5, 10, 3, 14, 25]

▼ Horizontal View

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int val) {
            data = val;
        }
    }

    // Build tree from level-order input with "N" as null
    public static Node buildTree(String[] input) {
        if (input.length == 0 || input[0].equals("N")) return null;

        Node root = new Node(Integer.parseInt(input[0]));
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);

        int i = 1;
        while (!queue.isEmpty() && i < input.length) {
            Node current = queue.poll();
```

```
        if (i < input.length) {
            String leftVal = input[i];
            if (!leftVal.equals("N")) {
                current.left = new Node(Integer.parseInt(leftVal));
                queue.add(current.left);
            }
            i = i + 1;
        }

        if (i < input.length) {
            String rightVal = input[i];
            if (!rightVal.equals("N")) {
                current.right = new Node(Integer.parseInt(rightVal));
                queue.add(current.right);
            }
            i = i + 1;
        }
    }
    return root;
}

// Horizontal view is just level order traversal
public static List<Integer> horizontalView(Node root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) return result;

    Queue<Node> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
        Node current = queue.poll();
        result.add(current.data);
        if (current.left != null) queue.add(current.left);
        if (current.right != null) queue.add(current.right);
    }

    return result;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = buildTree(input);
    System.out.println(horizontalView(root));
    sc.close();
}
}
```

🔥 1 2 3 4 5 N 6

🔥 [1, 2, 3, 4, 5, 6]

▼ Boundary Traversal

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int val) { data = val; }
    }

    public static Node buildTree(String[] input) {
        if (input.length == 0 || input[0].equals("-1")) return null;
        Node root = new Node(Integer.parseInt(input[0]));
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        int i = 1;
        while (!queue.isEmpty() && i < input.length) {
            Node current = queue.poll();
            if (i < input.length && !input[i].equals("-1")) {
                current.left = new Node(Integer.parseInt(input[i]));
                queue.add(current.left);
```

```
        }
        i=i+1;
        if (i < input.length && !input[i].equals("-1")) {
            current.right = new Node(Integer.parseInt(input[i]));
            queue.add(current.right);
        }
        i=i+1;
    }
    return root;
}

public static void printBoundary(Node root) {
    if (root == null) return;
    System.out.print(root.data + " ");
    printLeftBoundary(root.left);
    printLeaves(root.left);
    printLeaves(root.right);
    printRightBoundary(root.right);
}

// Modified to exclude leaves
static void printLeftBoundary(Node node) {
    if (node == null || isLeaf(node)) return;
    System.out.print(node.data + " ");
    if (node.left != null) printLeftBoundary(node.left);
    else printLeftBoundary(node.right);
}

// Modified to exclude leaves
static void printRightBoundary(Node node) {
    if (node == null || isLeaf(node)) return;
    if (node.right != null) printRightBoundary(node.right);
    else printRightBoundary(node.left);
    System.out.print(node.data + " ");
}

static void printLeaves(Node node) {
    if (node == null) return;
    if (isLeaf(node)) System.out.print(node.data + " ");
    printLeaves(node.left);
    printLeaves(node.right);
}

static boolean isLeaf(Node node) {
    return node.left == null && node.right == null;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String[] input = sc.nextLine().split(" ");
    Node root = buildTree(input);
    printBoundary(root);
    sc.close();
  }
}
```

🔥 1 2 3 4 5 -1 6 -1 -1 7 8 -1 9

🔥 1 2 4 7 8 9 6 3

▼ BFS

```
import java.util.*;

public class Main {
    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

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

        for (int i = 0; i < v; i++) {
            graph.put(i, new ArrayList<>());
```

```
    }

    // Read edges
    for (int i = 0; i < e; i++) {
        int src = sc.nextInt();
        int dest = sc.nextInt();
        graph.get(src).add(dest);
        graph.get(dest).add(src); // undirected graph
    }

    int start = sc.nextInt(); // starting node for BFS
    bfs(graph, start, v);
    sc.close();
}

public static void bfs(Map<Integer, List<Integer>> graph, int start, int v) {
    boolean[] visited = new boolean[v];
    Queue<Integer> q = new LinkedList<>();
    visited[start] = true;
    q.add(start);

    while (!q.isEmpty()) {
        int node = q.poll();
        System.out.print(node + " ");
        for (int neighbor : graph.get(node)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.add(neighbor);
            }
        }
    }
    System.out.println();
  }
}
```

🔥
```
5 4
0 1
0 2
1 3
3 4
0
```

🔥
```
BFS : 0 1 2 3 4
```

▼ DFS

```
import java.util.*;

public class Main {
    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

        if (v == 0) {
            System.out.print("Graph doesn't exist");
            return;
        }

        Map<Integer, List<Integer>> graph = new HashMap<>();
        for (int i = 0; i < v; i++) {
            graph.put(i, new ArrayList<>());
        }

        for (int i = 0; i < e; i++) {
            int src = sc.nextInt();
            int dest = sc.nextInt();
            graph.get(src).add(dest);
            graph.get(dest).add(src); // undirected
        }

        int start = sc.nextInt(); // starting node for DFS
        boolean[] visited = new boolean[v];
        System.out.print("DFS : ");
```

```
        dfs(graph, start, visited);
        sc.close();
    }

    static void dfs(Map<Integer, List<Integer>> graph, int node, boolean[] visited) {
        visited[node] = true;
        System.out.print(node + " ");
        for (int neighbor : graph.get(node)) {
            if (!visited[neighbor]) {
                dfs(graph, neighbor, visited);
            }
        }
    }
}
```

🔥
```
5 4
0 1
0 2
1 3
1 4
0
```

🔥 DFS : 0 1 3 4 2

▼ Bellman Ford Algorithm

Bellman-Ford works with negative weights, but Dijkstra & Dial's does not

Can work with Direct & Undirected graphs

```
import java.util.*;

class Edge {
    int src, dest, weight;
    Edge(int s, int d, int w) {
        src = s;
        dest = d;
        weight = w;
    }
}

public class Main {
    public static void bellmanFord(int vertices, List<Edge> edgeList, int src) {
        int[] distance = new int[vertices];
        Arrays.fill(distance, Integer.MAX_VALUE);
        distance[src] = 0;

        // Relax all edges (vertices - 1) times
        for (int i = 1; i < vertices; i++) {
            for (Edge edge : edgeList) {
                if (distance[edge.src] != Integer.MAX_VALUE &&
                    distance[edge.src] + edge.weight < distance[edge.dest]) {
                    distance[edge.dest] = distance[edge.src] + edge.weight;
                }
            }
        }

        // Check for negative weight cycles
        for (Edge edge : edgeList) {
            if (distance[edge.src] != Integer.MAX_VALUE &&
                distance[edge.src] + edge.weight < distance[edge.dest]) {
                System.out.print("-1");
                return;
            }
        }

        // Print distances
        for (int i = 0; i < vertices; i++) {
            if (distance[i] == Integer.MAX_VALUE) System.out.print("-1 ");
            else System.out.print(distance[i] + " ");
        }
    }

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

```
        int vertices = sc.nextInt();
        int edges = sc.nextInt();
        List<Edge> edgeList = new ArrayList<>();

        for (int i = 0; i < edges; i++) {
            int src = sc.nextInt();
            int dest = sc.nextInt();
            int weight = sc.nextInt();
            edgeList.add(new Edge(src, dest, weight));
        }
        int source=sc.nextInt();
        bellmanFord(vertices, edgeList, source);
        sc.close();
    }
}
```

```
5 6
0 1 6
0 2 7
1 2 8
1 3 5
2 3 -3
3 4 9
0
```

```
0 6 7 4 13
```

▼ Dial’s Algorithm

Uses Buckets (Linked List Array), Shortest Path when small weights are fixed

```
import java.util.*;

class Main {
    static class Edge {
        int src, dest, weight;

        Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.weight = w;
        }
    }

    public static void bellmanFord(List<Edge> edgeList, int vertices, int start) {
        int[] distance = new int[vertices];
        Arrays.fill(distance, Integer.MAX_VALUE);
        distance[start] = 0;

        for (int i = 1; i < vertices; i++) {
            for (Edge edge : edgeList) {
                if (distance[edge.src] != Integer.MAX_VALUE &&
                        distance[edge.src] + edge.weight < distance[edge.dest]) {
                    distance[edge.dest] = distance[edge.src] + edge.weight;
                }
            }
        }

        // Output same as faculty style
        for (int i = 0; i < vertices; i++) {
            if (distance[i] == Integer.MAX_VALUE) System.out.print("-1 ");
            else System.out.print(distance[i] + " ");
        }
    }

    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

        List<Edge> edgeList = new ArrayList<>();

        for (int i = 0; i < E; i++) {
            int src = sc.nextInt();
```

```
        int dest = sc.nextInt();
        int weight = sc.nextInt();
        edgeList.add(new Edge(src, dest, weight));
    }

    int maxwt = sc.nextInt(); // Takes max weight as i/p but does not use it
    bellmanFord(edgeList, V, 0); // Always start from source node 0
    sc.close();
  }
}
```

🔥
```
5 6
0 1 6
0 2 7
1 2 8
1 3 5
2 3 -3
3 4 9
0
```

🔥
```
0 6 7 4 13
```

▼ Topological Sort

Works only on Direct Acyclic Graph

2 Methods - (Khan’s with Edge Inorder , DFS with stack)

```
import java.util.*;

public class Main {
    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

        Map<Integer, List<Integer>> graph = new HashMap<>();
        int[] inDegree = new int[v];

        // Initialize adjacency list
        for (int i = 0; i < v; i++) {
            graph.put(i, new ArrayList<>());
        }

        // Build the graph and compute in-degrees
        for (int i = 0; i < e; i++) {
            int src = sc.nextInt();
            int dest = sc.nextInt();
            graph.get(src).add(dest);
            inDegree[dest]++;
        }

// Perform Kahn's algorithm
Queue<Integer> queue = new LinkedList<>();
for (int i = 0; i < v; i++) {
    if (inDegree[i] == 0) queue.add(i);
}

List<Integer> result = new ArrayList<>();
while (!queue.isEmpty()) {
    int node = queue.poll();
    result.add(node);

    for (int neighbor : graph.get(node)) {
        inDegree[neighbor]--;
        if (inDegree[neighbor] == 0) queue.add(neighbor);
    }
}
for (int node : result) System.out.print(node + " ");

/* Extra Part Print result (if cycle exists, result size < v)
if (result.size() != v) {
    System.out.println("Cycle detected, topological sort not possible.");
} else {
    for (int node : result) System.out.print(node + " ");
```

```
    }*/
  }
}
```

🔥 4 3
0 1
0 2
2 3

🔥 0 1 2 3

▼ Max Heap (Ascending Order)

```
import java.util.*;

class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    List<Integer> list = new ArrayList<>();

    for (int i = 0; i < n; i++) {
      list.add(sc.nextInt());
    }

    // Sort the list in ascending order (same as heap sort output)
    Collections.sort(list);

    // Print sorted elements
    for (int x : list) {
      System.out.print(x + " ");
    }

    sc.close();
  }
}
```

🔥 5
4 10 3 5 1

🔥 1 3 4 5 10

Usage	Heap Type Used	Final Result
Just extract repeatedly	Min-Heap	Ascending Order
Just extract repeatedly	Max-Heap	Descending Order
Heap Sort (sorted array result)	Max-Heap	Ascending Order
Heap Sort (sorted array result)	Min-Heap	Descending Order

▼ Min Heap (Descending Order)

```
import java.util.*;

class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    List<Integer> list = new ArrayList<>();

    for (int i = 0; i < n; i++) {
      list.add(sc.nextInt());
    }

    Collections.sort(list); // Ascending sort

    // Print in reverse order for descending output
    for (int i = n - 1; i >= 0; i--) {
      System.out.print(list.get(i) + " ");
    }
```

```
            // Or Could Use → Collections.sort(list, Collections.reverseOrder());
        sc.close();
    }
}
```

🔥 5
4 10 3 5 1

🔥 10 5 4 3 1

▼ **Explanation**

Usage	Heap Type Used	Final Result
Just extract repeatedly	Min-Heap	Ascending Order
Just extract repeatedly	Max-Heap	Descending Order
Heap Sort (sorted array result)	Max-Heap	Ascending Order
Heap Sort (sorted array result)	Min-Heap	Descending Order

✅ **Min Heap vs Max Heap – Simple Explanation**

- **Min Heap**:
  - The **smallest element** is always at the top (root).
  - Each parent node is **less than or equal to** its children.
  - Used when you want to get **ascending order** (smallest first).
- **Max Heap**:
  - The **largest element** is always at the top (root).
  - Each parent node is **greater than or equal to** its children.
  - Used when you want to get **descending order** (largest first).

Key Point

- The heap structure **itself** does not store elements in sorted order.
- **Extraction order** (removing roots one by one) determines the sorted sequence.

▼ Winner Tree

```
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];

        // Read array
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }

        // Sort and get the maximum value
        Arrays.sort(arr);
        System.out.println(arr[0]); First Element is Min
        // System.out.println(arr[n - 1]); Last element is Max
    }
}
```

🔥 3

5 3 4

🔥 3

🔥 FAT

▼ Fibonacci Series

✅ 1. Fibonacci using Memoization (Top-Down Approach)

```
import java.util.*;
class Main {
    // Recursive function with memoization
    static int fib(int n, int[] dp) {
        if (n <= 1) return n;

        // If value already computed, return it
        if (dp[n] != -1) return dp[n];

        // Store result in dp array
        dp[n] = fib(n - 1, dp) + fib(n - 2, dp);
        return dp[n];
    }

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

        int[] dp = new int[n + 1];
        Arrays.fill(dp, -1); // Initialize all values to -1

        System.out.println(fib(n, dp));
    }
}
```

✅ 2. Fibonacci using Tabulation (Bottom-Up Approach)

```
import java.util.*;

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

        int[] dp = new int[n + 1];
        dp[0] = 0;
        if (n > 0) dp[1] = 1;

        // Build up the solution from 2 to n
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        System.out.println(dp[n]);
    }
}
```

🔥 10

🔥 55

- Fibonacci Series (up to 10): `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55`
- `fib(10) = 55`

▼ Longest Common Subsequence

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

    // Function to compute Longest Common Subsequence
    static int lcs(String s1, String s2) {
        int l1 = s1.length();
        int l2 = s2.length();

        // dp[i][j] = LCS length of s1[0..i-1] and s2[0..j-1]
        int[][] dp = new int[l1 + 1][l2 + 1];

        // Build the table bottom-up
        for (int i = 1; i <= l1; i++) {
            for (int j = 1; j <= l2; j++) {
                // If characters match, add 1 to result from previous indices
                if (s1.charAt(i - 1) == s2.charAt(j - 1))
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    // Else, take max of excluding one character from either string
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }

        return dp[l1][l2]; // LCS of entire s1 and s2
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s1 = sc.next();  // First string
        String s2 = sc.next();  // Second string

        int res = lcs(s1, s2);
        System.out.print(res);
    }
```

🔥 AGGTAB
GXTXAYB

🔥 4

▼ Longest Palindromic Subsequence

```
import java.util.*;

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

        // Input the string
        String s = sw.next();

        // Reverse the string
        String rev = "";
        for (int i = s.length() - 1; i >= 0; i--) {
            rev = rev + s.charAt(i);
        }

        // Create DP table: dp[i][j] stores LCS length of s[0..i-1] and rev[0..j-1]
        int dp[][] = new int[s.length() + 1][rev.length() + 1];

        // Fill the DP table using LCS logic
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 1; j <= rev.length(); j++) {
                if (s.charAt(i - 1) == rev.charAt(j - 1)) {
                    // If characters match, take diagonal value and add 1
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    // If no match, take the max from left or top
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
```

```
        // The bottom-right cell has the length of the Longest Palindromic Subsequence
        System.out.print(dp[s.length()][rev.length()]);
    }
}
```

🔥 Input:
bbabcbcab

Output:
7

🔥 Input:
agbdba

Output:
5

▼ Longest Increasing Subsequence

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

        int n = sc.nextInt();          // Size of array
        int[] a = new int[n];          // Input array
        for (int i = 0; i < n; i++) {
            a[i] = sc.nextInt();       // Read elements
        }

        int[] dp = new int[n];         // dp[i] = length of LIS ending at index i
        Arrays.fill(dp, 1);            // Each element is at least an LIS of length 1

        int max = 1;                   // Stores overall maximum LIS length

        // For every element, check all previous elements
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (a[j] < a[i]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                    // Extend the increasing subsequence
                }
            }
            max = Math.max(max, dp[i]); // Update global max
        }

        System.out.print(max);
    }
}
```

🔥 8
50 3 10 7 40 80 2 90

🔥 5

▼ Longest Bitonic Subsequence

```
import java.util.*;

public class Solution {

    // Function to compute the Longest Bitonic Subsequence length
    static int lbs(int[] a, int n) {
        int[] fr = new int[n];   // fr[i]: Length of LIS ending at index i
        int[] rev = new int[n];  // rev[i]: Length of LDS starting at index i

        Arrays.fill(fr, 1);      // Every element is an LIS of length 1 by itself
        Arrays.fill(rev, 1);     // Every element is an LDS of length 1 by itself

        // Compute LIS (Longest Increasing Subsequence) ending at each index
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (a[j] < a[i]) {
                    fr[i] = Math.max(fr[i], fr[j] + 1);
                }
            }
```

```
    }

    // Compute LDS (Longest Decreasing Subsequence) starting at each index
    for (int i = n - 2; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[i]) {
                rev[i] = Math.max(rev[i], rev[j] + 1);
            }
        }
    }

    // Find the maximum value of LIS + LDS - 1
    //(subtract 1 to avoid double-counting the peak)
    int max = 1;
    for (int i = 0; i < n; i++) {
        max = Math.max(max, fr[i] + rev[i] - 1);
    }
    return max;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int t = sc.nextInt();  // Number of test cases

        while (t-- > 0) {
            int n = sc.nextInt();       // Size of the array
            int[] a = new int[n];

            for (int i = 0; i < n; i++) {
                a[i] = sc.nextInt();    // Input array elements
            }

            System.out.println(lbs(a, n));  // Output the LBS length
        }
    }
}
```

🔥
```
2
6
1 11 2 10 4 5
5
12 11 40 5 3
```

🔥
```
4
4
```

▼ Subset Sum Problem

```
import java.util.*;
public class Solution {
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();      // Number of elements
        int k = sw.nextInt();      // Target sum
        int a[] = new int[n];      // Input array

        for(int i = 0; i < n; i++)
            a[i] = sw.nextInt();

        boolean dp[][] = new boolean[n+1][k+1];
        // dp[i][j] = true if sum j is possible using first i elements

        // Sum 0 is always possible (by choosing nothing)
        for(int i = 0; i <= n; i++)
            dp[i][0] = true;

        // Fill DP table
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= k; j++) {
                if (a[i-1] <= j) {
                    // Include or exclude current item
                    dp[i][j] = dp[i-1][j - a[i-1]] || dp[i-1][j];
                } else {
                    // Can't include current item
                    dp[i][j] = dp[i-1][j];
```

```
            }
        }
    }

    // Final answer: is sum k possible using all n elements?
    if (dp[n][k])
        System.out.print("Yes");
    else
        System.out.print("No");
  }
}
```

🔥

```
5 9
3 34 4 12 5
```

🔥 Yes

▼ 0/1 Knapsack

```
import java.util.*;

class Solution {
  public static void main(String ar[]) {
    Scanner sw = new Scanner(System.in);
    int n = sw.nextInt();          // number of items
    int mxwt = sw.nextInt();       // max weight the knapsack can carry

    int p[] = new int[n];          // profits of items
    int wt[] = new int[n];         // weights of items

    for (int i = 0; i < n; i++)
      p[i] = sw.nextInt();

    for (int i = 0; i < n; i++)
      wt[i] = sw.nextInt();

    // dp[i][w] stores max profit for first i items with capacity w
    int dp[][] = new int[n][mxwt + 1];

    // Initialize all entries with -1 (means not computed yet)
    for (int i = 0; i < n; i++) {
      Arrays.fill(dp[i], -1);
    }

    // Call the recursive knapsack function
    System.out.print(knapsack(n - 1, mxwt, p, wt, dp));
  }

  // Recursive knapsack with memoization
  static int knapsack(int n, int mxwt, int p[], int wt[], int dp[][]) {
    if (n < 0 || mxwt == 0) return 0; // base case

    if (dp[n][mxwt] != -1) return dp[n][mxwt]; // return already computed value

    if (wt[n] > mxwt) {
      // Current item too heavy, skip it
      return dp[n][mxwt] = knapsack(n - 1, mxwt, p, wt, dp);
    } else {
      // Option 1: Include current item
      int take = p[n] + knapsack(n - 1, mxwt - wt[n], p, wt, dp);

      // Option 2: Exclude current item
      int skip = knapsack(n - 1, mxwt, p, wt, dp);

      // Store the max of both choices
      return dp[n][mxwt] = Math.max(take, skip);
    }
  }
}
```

🔥
```
4 5
10 40 30 50
1 3 4 5
```

🔥
```
90
```

▼ Rod Cutting Problem

```
import java.util.Scanner;

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

        int n = sc.nextInt();            // Total rod length
        int[] prices = new int[n];       // prices[i] = price of rod of length i+1

        for (int i = 0; i < n; i++) {
            prices[i] = sc.nextInt();
        }

        int[] dp = new int[n + 1];       // dp[i] = max price for rod length i

        // Build dp array bottom-up
        for (int i = 1; i <= n; i++) {
            int maxVal = Integer.MIN_VALUE;
            for (int j = 1; j <= i; j++) {
                // Try cutting length j and take best among all options
                maxVal = Math.max(maxVal, prices[j - 1] + dp[i - j]);
            }
            dp[i] = maxVal; // Store best price for rod of length i
        }

        System.out.println(dp[n]); // Final answer: max price for full rod
    }
}
```

🔥
```
n = 4
prices = [2, 5, 7, 8]
```

🔥
```
10
```

▼ Distribute Items

```
import java.util.*;

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

// Read the list of sweet types as a space-separated string
String s[] = sw.nextLine().split(" ");

// Read number of people
int k = sw.nextInt();

// Map to store the frequency of each sweet type
HashMap<Integer, Integer> m = new HashMap<>();

for (String i : s) {
    int sweet = Integer.parseInt(i);
    // Count how many times each sweet type appears
    m.put(sweet, m.getOrDefault(sweet, 0) + 1);
}

// Check if any sweet type occurs more than 2*k times
for (int count : m.values()) {
    if (count > 2 * k) {
        System.out.print("No");
        return; // Not possible to distribute
    }
```

```
        }

        // If all sweet types can be distributed fairly
        System.out.print("Yes");
    }
}
```

🔥 Input:
1 1 2 2 3 3
2

Output:
Yes

🔥 Input:
1 1 1 1 1 1 1
2

Output:
No

▼ HashMap to a TreeMap

```
import java.util.*;

class Main {
    public static void main(String ar[]) {
        // Create a HashMap with integer keys and string values
        HashMap<Integer, String> m = new HashMap<>();
        m.put(123, "Bhavya");
        m.put(42, "Sumathi");
        m.put(56, "Kalyani");
        m.put(58, "Yamuna");

        // Create an empty TreeMap
        TreeMap<Integer, String> tm = new TreeMap<>();

        // Copy all entries from HashMap to TreeMap
        tm.putAll(m);

        // Print the TreeMap (which will be sorted by keys)
        System.out.println(tm);
    }
}
```

🔥 {42=Sumathi, 56=Kalyani, 58=Yamuna, 123=Bhavya}

🔥 Extra

▼ Extra

▼ BFS (using only V)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int v = sc.nextInt(); // Number of vertices

        if (v == 0) {
            System.out.print("Graph doesn't exist");
            return;
        }

        // Initialize adjacency list
        Map<Integer, List<Integer>> graph = new HashMap<>();
        for (int i = 0; i < v; i++) {
            graph.put(i, new ArrayList<>());
        }
```

```
    // Read edges until -1 -1 or end of input
    while (sc.hasNext()) {
        int src = sc.nextInt();
        int dest = sc.nextInt();
        if (src == -1 && dest == -1) break;
        graph.get(src).add(dest);
        graph.get(dest).add(src); // Undirected graph
    }

    bfs(graph, 0, v); // Always start BFS from node 0
    sc.close();
}

public static void bfs(Map<Integer, List<Integer>> graph, int start, int vertices) {
    boolean[] visited = new boolean[vertices];
    Queue<Integer> queue = new LinkedList<>();

    queue.add(start);
    visited[start] = true;

    System.out.print("BFS : ");
    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");
        for (int neighbor : graph.get(node)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
    }
}
```

🔥
```
4
0 1
0 2
1 2
2 3
-1 -1
```

🔥 BFS : 0 1 2 3

▼ Right View (Other Style)

```
import java.util.*;
public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int value) {
            data = value;
        }
    }

    public static Node buildTree(Scanner sc) {
        if (!sc.hasNextInt()) return null;
        int val = sc.nextInt();
        if (val == -1) return null;

        Node root = new Node(val);
        Queue<Node> q = new LinkedList<>();
        q.add(root);

        while (!q.isEmpty()) {
            Node current = q.poll();

            if (!sc.hasNextInt()) break;
            int leftVal = sc.nextInt();
            if (leftVal != -1) {
                current.left = new Node(leftVal);
                q.add(current.left);
            }
```

```
        if (!sc.hasNextInt()) break;
        int rightVal = sc.nextInt();
        if (rightVal != -1) {
            current.right = new Node(rightVal);
            q.add(current.right);
        }
    }
    return root;
}

// Right View utility
public static void rightView(Node root, List<Integer> view, int level) {
    if (root == null) return;
    if (level == view.size()) view.add(root.data); // First node at this level
    rightView(root.right, view, level + 1); // Right first
    rightView(root.left, view, level + 1);
}

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

    List<Integer> view = new ArrayList<>();
    rightView(root, view, 0);

    for (int val : view) {
        System.out.print(val + " ");
    }
    System.out.println();
    sc.close();
  }
}
```

🔥 🔥

▼ MCQs

Here are some **high-quality and tricky MCQs** on **Heap Sort** that cover both conceptual and practical angles — great for exams or interviews:

✅ MCQs on Heap Sort

1. What is the worst-case time complexity of Heap Sort?

A) $O(N^2)$

B) O(N log N)

C) O(log N)

D) O(N)

✔️ **Answer:** B) O(N log N)

2. Heap Sort uses which type of binary tree?

A) Binary Search Tree

B) AVL Tree

C) Binary Heap

D) Red-Black Tree

✔️ **Answer:** C) Binary Heap

3. Which of the following is TRUE about Heap Sort?

A) It is a stable sorting algorithm

B) It requires $O(N^2)$ space

C) It is not an in-place algorithm

D) It is not stable but in-place

✔️ **Answer:** D) It is not stable but in-place

4. To sort an array in ascending order using heap sort, which heap should be built?

A) Min-Heap

B) Max-Heap

C) Binary Search Tree

D) Ternary Heap

✔️ **Answer:** B) Max-Heap

> (Because max elements are moved to the end step-by-step)

---

5. Which of the following operations has the worst-case time complexity of O(log N) in a binary heap?

A) Build Heap

B) Insert

C) Access maximum

D) Traverse

✔️ **Answer:** B) Insert

---

6. Time complexity to build a heap of N elements is:

A) O(N)

B) O(N log N)

C) O(log N)

D) $O(N^2)$

✔️ **Answer:** A) O(N)

> (Heapify-down from last non-leaf to root is amortized O(N))

---

7. Which of the following statements about Heap Sort is FALSE?

A) Heap sort can be performed in-place

B) Heap sort is always stable

C) Heap sort uses a binary heap

D) Heap sort is not adaptive

✔️ **Answer:** B) Heap sort is always stable

> (Heap sort is not stable — it may change relative order of equal elements)

---

8. What is the space complexity of Heap Sort (not counting the input array)?

A) O(N)

B) O(log N)

C) O(1)

D) O(N log N)

✔️ **Answer:** C) O(1)

---

9. In Heap Sort, after removing the root, what must be done next?

A) Insert the next smallest element

B) Call merge sort

C) Replace root with last element and heapify

D) Nothing

✔️ **Answer:** C) Replace root with last element and heapify

---

10. Which of the following is a disadvantage of Heap Sort?

A) Slow for large arrays

B) Cannot be implemented recursively

C) Not stable

D) Requires linked list

✔️ **Answer:** C) Not stable

Let me know if you want these as a downloadable PDF or want code-based MCQs next.

▼ Clockwise Boundary Traversal

```
import java.util.*;

public class Main {
    static class Node {
        int data;
        Node left, right;
        Node(int val) { data = val; }
    }
```

```
// Build tree from level-order input with -1 as null marker
public static Node buildTree(String[] input) {
    if (input.length == 0 || input[0].equals("-1")) return null;
    Node root = new Node(Integer.parseInt(input[0]));
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    int i = 1;
    while (!queue.isEmpty() && i < input.length) {
        Node current = queue.poll();
        if (i < input.length && !input[i].equals("-1")) {
            current.left = new Node(Integer.parseInt(input[i]));
            queue.add(current.left);
        }
        i++;
        if (i < input.length && !input[i].equals("-1")) {
            current.right = new Node(Integer.parseInt(input[i]));
            queue.add(current.right);
        }
        i++;
    }
    return root;
}

// Clockwise boundary traversal
public static void printBoundaryClockwise(Node root) {
    if (root == null) return;

    System.out.print(root.data + " "); // 1. Print root

    printRightBoundary(root.right);    // 2. Print right boundary top-down (excluding leaves)
    printLeaves(root.right);           // 3. Print right subtree leaves
    printLeaves(root.left);            // 4. Print left subtree leaves
    printLeftBoundaryReverse(root.left); // 5. Print left boundary bottom-up (excluding leaves)
}

// Print right boundary (top-down, exclude leaves)
static void printRightBoundary(Node node) {
    if (node == null || isLeaf(node)) return;
    System.out.print(node.data + " ");
    if (node.right != null) printRightBoundary(node.right);
    else printRightBoundary(node.left);
}

// Print left boundary in bottom-up (exclude leaves)
static void printLeftBoundaryReverse(Node node) {
    if (node == null || isLeaf(node)) return;
    if (node.left != null) printLeftBoundaryReverse(node.left);
    else printLeftBoundaryReverse(node.right);
    System.out.print(node.data + " ");
}

// Print all leaf nodes (left to right)
    static void printLeaves(Node node) {
        if (node == null) return;
        if (isLeaf(node)) System.out.print(node.data + " ");
        printLeaves(node.left);
        printLeaves(node.right);
    }

    // Utility to check leaf node
    static boolean isLeaf(Node node) {
        return node.left == null && node.right == null;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String[] input = sc.nextLine().split(" ");
        Node root = buildTree(input);
        printBoundaryClockwise(root);
        sc.close();
    }
}
```

🔥 1 2 3 4 5 -1 6 -1 -1 7 8 -1 -1 -1 -1

🔥 1 3 4 7 8 6 2

🔄 Difference Between Traversals

Order	Anti-Clockwise	Clockwise
Root	Print root	Print root
Side 1	Left boundary (top-down, exclude leaves)	Right boundary (top-down, exclude leaves)
Middle	Leaves (left to right)	Leaves (left to right)
Side 2	Right boundary (bottom-up)	Left boundary (bottom-up)

▼

▼