Loop Detection:

```
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();
  }
  int n1=sc.nextInt();
  for(int i=0;i<n;i++){
    System.out.print(arr[i]+"->");
  }

  if(n1==-1){

    System.out.println("NULL");
    System.out.println("Loop not detected");
  }
  else{
    System.out.println(arr[n1]);
    System.out.println("Loop detected at node "+ (n1+1));
  }

        }
}
```

Celebrity problem

```
import java.io.*;
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][n];
                for(int i=0;i<n;i++){
                  for(int j=0;j<n;j++){
                    arr[i][j]=sc.nextInt();
                  }
                }
                for(int i=0;i<n;i++){
                  int count=0;
                  int flag=1;
                  for(int j=0;j<n;j++){
                    if(arr[i][j]==1){
                      flag=0;
                    }
                  }
                  for(int j=0;j<n;j++){
                    if(arr[j][i]==1){
                      count++;
                    }
```

```
                    }
                    if(flag==1 && count==n-1){
                        System.out.print("Celebrity is : "+(i+1));
                        return;
                    }
                }
                System.out.print("No Celebrity found");
        }
}
```

Stock Span:

```
import java.io.*;
import java.util.*;
public class Node{

   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();
                }
                for(int i=0;i<n;i++){
                    int count=1;
                    int j=i-1;
                    while(j>=0){
                        if(arr[j]<arr[i]){
                            count++;
                        }
                        j--;
                    }
                    System.out.print(count+" ");
                }
        }
}
```

Sliding window

```
import java.io.*;
import java.util.*;
public class Node{
   public static int findmax(int[] arr,int i,int j){
      int max=-99999;
      for(int k=i;k<j;k++){
         if(arr[k]>max){
            max=arr[k];
         }
      }
      return max;
   }
   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 k=sc.nextInt();
            for(int i=0;i<=n-k;i++){
                System.out.print(findmax(arr,i,i+k)+" ");
            }
        }
}
```

Stack permutation:

```
import java.util.*;

public class Main {
    public static boolean isStackPermutation(int[] original, int[] target) {
        Stack<Integer> stack = new Stack<>();
        int i = 0; // Pointer for target array

        for (int num : original) {
            stack.push(num); // Push current element

            // Pop from stack while it matches the target array
            while (!stack.isEmpty() && stack.peek() == target[i]) {
                stack.pop();
                i++;
            }
        }

        // If stack is empty, it's a valid permutation
        return stack.isEmpty();
    }

    public static void main(String[] args) {

        int[] original = {1, 2, 3};
        int[] target = {2, 1, 3}; // Change this to test different cases

        boolean result = isStackPermutation(original, target);
        System.out.println("Is it a stack permutation? " + result);
    }
}
```

Minimum Stack :

```
import java.util.*;

public class Main {
    static Stack<Integer> stack = new Stack<>();
    static Stack<Integer> mstack = new Stack<>();

    public static void push(int val) {
        stack.push(val);
        if (mstack.isEmpty() || val <= mstack.peek()) {
            mstack.push(val);
        }
    }

    public static void pop() {
```

```
    if (!stack.isEmpty()) {
      if (stack.peek().equals(mstack.peek())) {
        mstack.pop();
      }
      stack.pop();
    }
  }

  public static int getTop() {
    return stack.peek();
  }

  public static int getMin() {
    return mstack.peek();
  }

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

    push(5);
    push(2);
    push(7);
    push(1);
    pop();

    System.out.println(getMin());
  }
}
```

Tower of Haonoi:

```
import java.util.*;
class Main{
  public static void main(String[] args) {
    Scanner s1=new Scanner(System.in);
    System.out.print("Enter the number of disks-");
    int n=s1.nextInt();
    towerofHanoi(n,'A','B','C');
  }
public static void towerofHanoi(int n,char fromrod,char helperrod,char torod){
  if(n==1)
{
  System.out.println("Move disk from " + fromrod + "->" + torod);
  return;
}
towerofHanoi(n-1,fromrod,torod,helperrod);
System.out.println("Move disk from " + fromrod + "->" + torod);towerofHanoi(n-
1,helperrod,fromrod,torod);
}}
```

1.Winner tree

Sort the Array and take the min element

```
import java.util.*;
public class Main
{
	public static void main(String[] args) {
		int arr[]={4,2,1,4,1,5};
		Arrays.sort(arr);
		System.out.println(arr[0]);
	}
}
```

2.K- Array element

Sort the Array and get Max element

```
import java.util.*;
public class Main
{
	public static void main(String[] args) {
		int arr[]={4,2,1,4,1,5};
		Arrays.sort(arr);
		System.out.println(arr.length-1);
	}
}
```

3. Binomial Heap

Cobine the given number of array and find min element

```
import java.util.*;

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

      // Input size of first array
      System.out.print("Enter size of first array: ");
      int n1 = sc.nextInt();
      int[] arr1 = new int[n1];
      System.out.println("Enter elements of first array:");
      for (int i = 0; i < n1; i++) {
         arr1[i] = sc.nextInt();
      }

      // Input size of second array
      System.out.print("Enter size of second array: ");
      int n2 = sc.nextInt();
      int[] arr2 = new int[n2];
      System.out.println("Enter elements of second array:");
      for (int i = 0; i < n2; i++) {
         arr2[i] = sc.nextInt();
      }

      // Merge the two arrays
      int[] mergedArray = new int[n1 + n2];
      System.arraycopy(arr1, 0, mergedArray, 0, n1);
```

```
        System.arraycopy(arr2, 0, mergedArray, n1, n2);

        // Find the minimum element
        int minElement = mergedArray[0];
        for (int i = 1; i < mergedArray.length; i++) {
            if (mergedArray[i] < minElement) {
                minElement = mergedArray[i];
            }
        }

        // Output the merged array and minimum element
        System.out.println("Merged Array: " + Arrays.toString(mergedArray));
        System.out.println("Minimum Element: " + minElement);

        sc.close();
    }
}
```

4.Heap Sort

Use sort and find the max or min element

5. Recover the BST

Sort the array

BFS

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int V = sc.nextInt();
      List<Integer>[] adj = new ArrayList[V];
      for (int i = 0; i < V; i++) adj[i] = new ArrayList<>();

      while (true) {
         int v = sc.nextInt(), e = sc.nextInt();
         if (v == -1 && e == -1) break;
         adj[v].add(e);
         adj[e].add(v);
      }

      int s = sc.nextInt();
      boolean[] visited = new boolean[V];
      Queue<Integer> q = new LinkedList<>();
      q.add(s);
      visited[s] = true;

      while (!q.isEmpty()) {
         int node = q.poll();
         System.out.print(node + " ");
         for (int n : adj[node])
            if (!visited[n]) {
               visited[n] = true;
```

```
                q.add(n);
            }
        }
    }
}
```

DFS

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int V = sc.nextInt();
        List<Integer>[] adj = new ArrayList[V];
        for (int i = 0; i < V; i++) adj[i] = new ArrayList<>();

        while (true) {
            int v = sc.nextInt(), e = sc.nextInt();
            if (v == -1 && e == -1) break;
            adj[v].add(e);
            adj[e].add(v);
        }

        int start = sc.nextInt();
        boolean[] visited = new boolean[V];

        dfs(start, adj, visited);
    }

    static void dfs(int s, List<Integer>[] adj, boolean[] visited) {
        visited[s] = true;
        System.out.print(s + " ");
```

```
        for (int n : adj[s])
            if (!visited[n]) dfs(n, adj, visited);
    }
}
```

Bellman Ford

```
import java.util.*;

class Main {
    static class Edge {
        int src, dest, weight;
        Edge(int u, int v, int w) { src = u; dest = v; weight = w; }
    }

    static void bellmanFord(int V, int E, List<Edge> edges) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[0] = 0;

        for (int i = 1; i < V; i++)
            for (Edge e : edges)
                if (dist[e.src] != Integer.MAX_VALUE && dist[e.src] + e.weight < dist[e.dest])
                    dist[e.dest] = dist[e.src] + e.weight;

        for (Edge e : edges)
            if (dist[e.src] != Integer.MAX_VALUE && dist[e.src] + e.weight < dist[e.dest]) {
                System.out.println("-1");
                return;
            }

        for (int d : dist) System.out.print((d == Integer.MAX_VALUE ? -1 : d) + " ");
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int V = sc.nextInt(), E = sc.nextInt();
        List<Edge> edges = new ArrayList<>();

        for (int i = 0; i < E; i++)
            edges.add(new Edge(sc.nextInt(), sc.nextInt(), sc.nextInt()));

        bellmanFord(V, E, edges);
        sc.close();
    }
}
```

Topological algo:

```
import java.io.*;
import java.util.*;
public class Main
{
        public static void main(String[] args) {
                Scanner sc=new Scanner(System.in);
                int nodes=sc.nextInt();
                int edges=sc.nextInt();
                int[] arr1=new int[edges];
                int[] arr2=new int[edges];
                int[] dist=new int[nodes];
                for(int i=0;i<edges;i++){
                   arr1[i]=sc.nextInt();
                   arr2[i]=sc.nextInt();
                }
                for(int i=0;i<nodes;i++){
```

```
                for(int j=0;j<nodes;j++){
                    if(arr2[j]==i){
                        dist[i]++;
                    }
                }
            }
            int n=nodes;
            while(n>0){
                for(int i=0;i<nodes;i++){
                    if(dist[i]==0){
                        System.out.print(i+" ");
                        dist[i]=99999;
                        for(int j=0;j<edges;j++){
                            if(arr1[j]==i){
                                dist[arr2[j]]--;
                            }
                        }
                        n--;
                    }
                }
            }
        }
}
```

Boundaty traversal

```
import java.util.*;

class Node {
   int data;
   Node left, right;
   Node(int d) { data = d; }
}
```

```
public class Main {
    static int idx = 0;

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

        int[] arr = new int[input.length];
        for (int i = 0; i < input.length; i++) arr[i] = Integer.parseInt(input[i]);

        Node root = buildTree(arr);

        List<Integer> boundary = boundaryTraversal(root);
        for (int val : boundary) System.out.print(val + " ");
    }

    static Node buildTree(int[] arr) {
        if (arr.length == 0) return null;
        Node[] nodes = new Node[arr.length];
        for (int i = 0; i < arr.length; i++) nodes[i] = new Node(arr[i]);
        for (int i = 0; i < arr.length; i++) {
            if (2*i + 1 < arr.length) nodes[i].left = nodes[2*i + 1];
            if (2*i + 2 < arr.length) nodes[i].right = nodes[2*i + 2];
        }
        return nodes[0];
    }

    static List<Integer> boundaryTraversal(Node root) {
        List<Integer> res = new ArrayList<>();
```

```
    if (root == null) return res;
    if (!isLeaf(root)) res.add(root.data);

    Node node = root.left;
    while (node != null) {
        if (!isLeaf(node)) res.add(node.data);
        node = (node.left != null) ? node.left : node.right;
    }

    addLeaves(root, res);

    Stack<Integer> stack = new Stack<>();
    node = root.right;
    while (node != null) {
        if (!isLeaf(node)) stack.push(node.data);
        node = (node.right != null) ? node.right : node.left;
    }
    while (!stack.isEmpty()) res.add(stack.pop());

    return res;
}

static void addLeaves(Node node, List<Integer> res) {
    if (node == null) return;
    if (isLeaf(node)) res.add(node.data);
    addLeaves(node.left, res);
    addLeaves(node.right, res);
}

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

```
    }
}
```

Verticle order

```
import java.util.Scanner;

class Node {
    int data;
    Node left, right;
    Node(int d) { data = d; }
}

public class Main {
    static final int OFFSET = 500; // To handle negative horizontal distances
    static int[][] vertical = new int[1000][1000]; // vertical[hd][list of nodes]
    static int[] count = new int[1000]; // count[hd]
    static int minHd = OFFSET, maxHd = OFFSET;

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

        int n = input.length;
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = Integer.parseInt(input[i]);

        Node root = buildTree(arr);

        verticalOrder(root, OFFSET);
```

```
        for (int hd = minHd; hd <= maxHd; hd++) {
            for (int i = 0; i < count[hd]; i++) {
                System.out.print(vertical[hd][i] + " ");
            }
        }
    }

    static Node buildTree(int[] arr) {
        if (arr.length == 0) return null;
        Node[] nodes = new Node[arr.length];
        for (int i = 0; i < arr.length; i++) nodes[i] = new Node(arr[i]);
        for (int i = 0; i < arr.length; i++) {
            if (2*i + 1 < arr.length) nodes[i].left = nodes[2*i + 1];
            if (2*i + 2 < arr.length) nodes[i].right = nodes[2*i + 2];
        }
        return nodes[0];
    }

    static void verticalOrder(Node node, int hd) {
        if (node == null) return;
        vertical[hd][count[hd]++] = node.data;
        if (hd < minHd) minHd = hd;
        if (hd > maxHd) maxHd = hd;
        verticalOrder(node.left, hd - 1);
        verticalOrder(node.right, hd + 1);
    }
}
```

Top:

```
import java.util.Scanner;
```

```
class Node {
    int data;
    Node left, right;
    Node(int d) { data = d; }
}

public class Main {
    static final int OFFSET = 500;
    static int[] top = new int[1000]; // top[hd] = first seen node
    static boolean[] seen = new boolean[1000];
    static int minHd = OFFSET, maxHd = OFFSET;

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

        int n = input.length;
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = Integer.parseInt(input[i]);

        Node root = buildTree(arr);

        topView(root, OFFSET, 0);

        for (int hd = minHd; hd <= maxHd; hd++) {
            System.out.print(top[hd] + " ");
        }
    }

    static Node buildTree(int[] arr) {
```

```
        if (arr.length == 0) return null;
        Node[] nodes = new Node[arr.length];
        for (int i = 0; i < arr.length; i++) nodes[i] = new Node(arr[i]);
        for (int i = 0; i < arr.length; i++) {
            if (2*i + 1 < arr.length) nodes[i].left = nodes[2*i + 1];
            if (2*i + 2 < arr.length) nodes[i].right = nodes[2*i + 2];
        }
        return nodes[0];
    }

    static void topView(Node node, int hd, int level) {
        if (node == null) return;
        if (!seen[hd]) {
            top[hd] = node.data;
            seen[hd] = true;
        }
        if (hd < minHd) minHd = hd;
        if (hd > maxHd) maxHd = hd;
        topView(node.left, hd - 1, level + 1);
        topView(node.right, hd + 1, level + 1);
    }
}
```

Bottom:

```
import java.util.Scanner;

class Node {
    int data;
    Node left, right;
    Node(int d) { data = d; }
}
```

```
public class Main {
    static final int OFFSET = 500;
    static int[] bottom = new int[1000]; // bottom[hd] = last seen node
    static int minHd = OFFSET, maxHd = OFFSET;

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

        int n = input.length;
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = Integer.parseInt(input[i]);

        Node root = buildTree(arr);

        bottomView(root, OFFSET);

        for (int hd = minHd; hd <= maxHd; hd++) {
            System.out.print(bottom[hd] + " ");
        }
    }

    static Node buildTree(int[] arr) {
        if (arr.length == 0) return null;
        Node[] nodes = new Node[arr.length];
        for (int i = 0; i < arr.length; i++) nodes[i] = new Node(arr[i]);
        for (int i = 0; i < arr.length; i++) {
            if (2*i + 1 < arr.length) nodes[i].left = nodes[2*i + 1];
            if (2*i + 2 < arr.length) nodes[i].right = nodes[2*i + 2];
```

```
        }
        return nodes[0];
    }

    static void bottomView(Node node, int hd) {
        if (node == null) return;
        bottom[hd] = node.data; // Overwrite every time
        if (hd < minHd) minHd = hd;
        if (hd > maxHd) maxHd = hd;
        bottomView(node.left, hd - 1);
        bottomView(node.right, hd + 1);
    }
}
```

Boundary/ Top/Bottom/Left/Right

```
import java.util.*;

class Node {
    int data;
    Node left, right;
    Node(int data) { this.data = data; }
}

class BinaryTree {
    Node root;

    void printBoundary(Node node) {
        if (node == null) return;
        System.out.print(node.data + " ");
        printBoundaryLeft(node.left);
        printLeaves(node);
```

```
    printBoundaryRight(node.right);
    System.out.println();
}

void printBoundaryLeft(Node node) {
    while (node != null) {
        if (node.left != null || node.right != null) System.out.print(node.data + " ");
        node = (node.left != null) ? node.left : node.right;
    }
}

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

void printBoundaryRight(Node node) {
    Stack<Integer> stack = new Stack<>();
    while (node != null) {
        if (node.left != null || node.right != null) stack.push(node.data);
        node = (node.right != null) ? node.right : node.left;
    }
    while (!stack.isEmpty()) System.out.print(stack.pop() + " ");
}

void printTopView(Node node) {
    if (node == null) return;
    Map<Integer, Integer> map = new TreeMap<>();
    Queue<Pair> queue = new LinkedList<>();
```

```
    queue.add(new Pair(node, 0));
    while (!queue.isEmpty()) {
        Pair p = queue.poll();
        if (!map.containsKey(p.hd)) map.put(p.hd, p.node.data);
        if (p.node.left != null) queue.add(new Pair(p.node.left, p.hd - 1));
        if (p.node.right != null) queue.add(new Pair(p.node.right, p.hd + 1));
    }
    for (int val : map.values()) System.out.print(val + " ");
    System.out.println();
}

void printBottomView(Node node) {
    if (node == null) return;
    Map<Integer, Integer> map = new TreeMap<>();
    Queue<Pair> queue = new LinkedList<>();
    queue.add(new Pair(node, 0));
    while (!queue.isEmpty()) {
        Pair p = queue.poll();
        map.put(p.hd, p.node.data);
        if (p.node.left != null) queue.add(new Pair(p.node.left, p.hd - 1));
        if (p.node.right != null) queue.add(new Pair(p.node.right, p.hd + 1));
    }
    for (int val : map.values()) System.out.print(val + " ");
    System.out.println();
}

void printLeftView(Node node) {
    if (node == null) return;
    Queue<Node> queue = new LinkedList<>();
    queue.add(node);
    while (!queue.isEmpty()) {
```

```
        int size = queue.size();
        System.out.print(queue.peek().data + " ");
        for (int i = 0; i < size; i++) {
            Node temp = queue.poll();
            if (temp.left != null) queue.add(temp.left);
            if (temp.right != null) queue.add(temp.right);
        }
    }
    System.out.println();
}

void printRightView(Node node) {
    if (node == null) return;
    Queue<Node> queue = new LinkedList<>();
    queue.add(node);
    while (!queue.isEmpty()) {
        int size = queue.size();
        Node last = null;
        for (int i = 0; i < size; i++) {
            Node temp = queue.poll();
            last = temp;
            if (temp.left != null) queue.add(temp.left);
            if (temp.right != null) queue.add(temp.right);
        }
        System.out.print(last.data + " ");
    }
    System.out.println();
}

Node buildTree(String input) {
    String[] values = input.split(" ");
```

```
        Queue<Node> queue = new LinkedList<>();
        Node root = new Node(Integer.parseInt(values[0]));
        queue.add(root);
        for (int i = 1; i < values.length; i += 2) {
            Node current = queue.poll();
            current.left = new Node(Integer.parseInt(values[i]));
            queue.add(current.left);
            if (i + 1 < values.length) {
                current.right = new Node(Integer.parseInt(values[i + 1]));
                queue.add(current.right);
            }
        }
        return root;
    }

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

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

        BinaryTree tree = new BinaryTree();
        tree.root = tree.buildTree(input);
```

```
        System.out.println("Boundary Traversal:");
        tree.printBoundary(tree.root);

        System.out.println("Top View:");
        tree.printTopView(tree.root);

        System.out.println("Bottom View:");
        tree.printBottomView(tree.root);

        System.out.println("Left View:");
        tree.printLeftView(tree.root);

        System.out.println("Right View:");
        tree.printRightView(tree.root);
    }
}
```

1. HashMap to TreeMap

```
import java.util.*;


class Main {


   public static <K, V> Map<K, V> convertToTreeMap(Map<K, V> hashMap) {
      Map<K, V> treeMap = new TreeMap<>();
      treeMap.putAll(hashMap);
      return treeMap;
   }


   public static void main(String args[]) {
      Scanner s = new Scanner(System.in);
      Map<String, String> hashMap = new HashMap<>();


      int n = s.nextInt();
      for (int i = 0; i < n; i++) {
         hashMap.put(s.next(), s.next());
      }


      System.out.println("HashMap: " + hashMap);


      Map<String, String> treeMap = convertToTreeMap(hashMap);
      System.out.println("TreeMap: " + treeMap);
   }
}
```

2. Fibonacci problem:

```
import java.util.*;


class Main {
   public static void main(String[] args) {
      Scanner s = new Scanner(System.in);
      int n = s.nextInt(), a = 0, b = 1;
      for (int i = 0; i < n; i++) {
```

```
            System.out.print(a + " ");
            int c = a + b;
            a = b;
            b = c;
        }
    }
}
```

3. LCS

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String a = s.next(), b = s.next();
        int m = a.length(), n = b.length();

        // Create a 2D DP table to store LCS lengths
        int[][] dp = new int[m + 1][n + 1];

        // Fill the DP table using the logic explained above
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // If characters match, extend the LCS from the diagonal
                if (a.charAt(i - 1) == b.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    // If characters don't match, take the maximum of the left or top cell
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        // The bottom-right cell contains the length of the LCS
        System.out.println(dp[m][n]);
```

```
    }
}
```

4. Longest palindrominc subseuqnce

```
import java.util.Scanner;

public class Main {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        String s = sc.next();
        System.out.println(longestPalindromicSubsequence(s));
    }

    public static int longestPalindromicSubsequence(String s) {
        int n = s.length();
        int[][] dp = new int[n][n];

        for (int i = 0; i < n; i++)
            dp[i][i] = 1;  // Each single character is a palindrome of length 1

        for (int len = 2; len <= n; len++) {
            for (int start = 0; start <= n - len; start++) {
                int end = start + len - 1;
                if (s.charAt(start) == s.charAt(end))
                    dp[start][end] = 2 + dp[start + 1][end - 1];
                else
                    dp[start][end] = Math.max(dp[start + 1][end], dp[start][end - 1]);
            }
        }

        return dp[0][n - 1];  // Longest palindromic subsequence in the entire string
    }
}
```

5. 0/1 knapsack

```
public class Main {
   public static void main(String[] args) {
      int[] weights = {1, 2, 3};  // Item weights
      int[] values = {60, 100, 120};  // Item values
      int capacity = 5;  // Knapsack capacity

      System.out.println(knapsack(weights, values, capacity));
   }

   public static int knapsack(int[] weights, int[] values, int W) {
      int n = weights.length;
      int[] dp = new int[W + 1];

      for (int i = 0; i < n; i++) {
         for (int w = W; w >= weights[i]; w--) {
            dp[w] = Math.max(dp[w], dp[w - weights[i]] + values[i]);
         }
      }

      return dp[W];
   }
}
```

6. Count coin:

```
import java.util.Scanner;
class Main{
  public static void main(String args[]){
    Scanner s = new Scanner(System.in);
    int n=s.nextInt();
    int deno[] = new int[n];
    for(int i=0; i<n; i++)
       deno[i]=s.nextInt();
    int amount = s.nextInt();
    int count=0;
    for(int i = deno.length - 1; i >= 0; i--) {
      while(amount >= deno[i]) {
         count+=1;
       amount -= deno[i];
      }
    }
    if(amount==0)
```

```
      System.out.print(count);
    else
      System.out.print(-1);
  }
}
```

7.Shortest subsequence

Len of a + len of b – len of subsequence

8. Rod cutting

```
public class Main {

    // Function to calculate the maximum revenue for a rod of length n
    public static int rodCutting(int[] prices, int n) {
        int[] dp = new int[n + 1];

        // Fill the dp array with the maximum revenue for each rod length
        for (int i = 1; i <= n; i++) {
            int maxRevenue = prices[i - 1]; // Case when no cut is made, use price of full rod
            for (int j = 1; j < i; j++) {
                maxRevenue = Math.max(maxRevenue, dp[j] + dp[i - j]);
            }
            dp[i] = maxRevenue;
        }

        // The maximum revenue for a rod of length n is stored in dp[n]
        return dp[n];
    }

    public static void main(String[] args) {
        int[] prices = {1, 5, 8, 9}; // Prices for lengths 1 to 8
        int n = 4; // Length of the rod
        System.out.println("Maximum Revenue: " + rodCutting(prices, n));
    }
}
```