PROGRAM : Write a function to find the maximum and minimum elements in an array.

```java
import java.util.Scanner;
public class minmax {
  public static int maxx(int[] a, int m) {
    for (int i = 0; i < a.length; i++) {
      if (a[i] > m) {
        m = a[i];
      }
    }
    return m;
  }
  public static int minn(int[] a, int m) {
    for (int i = 0; i < a.length; i++) {
      if (a[i] < m) {
        m = a[i];
      }
    }
    return m;
  }
  public static void main(String[] args) {
    Scanner obj = new Scanner(System.in);
    System.out.println("Enter the size of the array:");
    int size = obj.nextInt();
    int[] a = new int[size];
    for (int i = 0; i < a.length; i++) {
      System.out.println("Enter the elements:"+i);
      a[i] = obj.nextInt();
    }
    int m = Integer.MIN_VALUE;
    int lr = maxx(a, m);
    System.out.println("Max value in arr" + lr);
    m = Integer.MAX_VALUE;
    int sm = minn(a, m);
    System.out.println("Min value in arr " + sm);

  }
}
```

PROGRAM : Write a function to reverse an array in place.

Sourcecode:

```java
import java.util.Scanner;
public class reversearr {
   public void rr(int a[]){
      int i=0;
      int j=a.length-1;
      while (i<j){
         int temp=a[i];
         a[i]=a[j];
         a[j]=temp;

         i++;
         j--;
      }
      for(int k=0;k<a.length;k++){
         System.out.print(a[k]+" ");
      }

   }
   public static void main(String[] args) {
      Scanner obj = new Scanner(System.in);
      System.out.println("Enter the size of the array:");
      int size = obj.nextInt();
      int[] a = new int[size];
      for (int i = 0; i < a.length; i++) {
         System.out.println("Enter the elements:"+i);
         a[i] = obj.nextInt();
      }

      reversearr palat = new reversearr();
      palat.rr(a);
   }
}
```

PROGRAM : Find the Kth Smallest/Largest Element in an Array: Write a function to find the Kth smallest or largest element in an array.

PROGRAM :

```
import java.util.Arrays;

public class Main {
  public static int findKthSmallest(int[] arr, int k) {
    Arrays.sort(arr);
    return arr[k - 1];
  }

  public static int findKthLargest(int[] arr, int k) {
    Arrays.sort(arr);
    return arr[arr.length - k];
  }

  public static void main(String[] args) {
    int[] arr = {7, 10, 4, 3, 20, 15};
    int k = 3;

    System.out.println("Kth Smallest Element: " + findKthSmallest(arr, k));
    System.out.println("Kth Largest Element: " + findKthLargest(arr, k));
  }
}
```

PROGRAM : Sort an Array of 0s, 1s, and 2s: Given an array containing only 0s, 1s, and 2s, sort the array in linear time.

```
public class Main {
    public static void main(String[] args) {
        int[] nums = {0, 1, 2, 1, 0, 2, 1};
        sortColors(nums);

        System.out.println("Sorted Array:");
        for (int num : nums) {
            System.out.print(num + " ");
        }
    }

    public static void sortColors(int[] nums) {
        int low = 0, mid = 0, high = nums.length - 1;

        while (mid <= high) {
            if (nums[mid] == 0) {
                int temp = nums[low];
                nums[low] = nums[mid];
                nums[mid] = temp;
                low++;
                mid++;
            } else if (nums[mid] == 1) {
                mid++;
            } else {
                int temp = nums[mid];
                nums[mid] = nums[high];
                nums[high] = temp;
                high--;
            }
        }
    }
}
```

PROGRAM : Move All Zeroes to End of Array: Write a function to move all zeroes in an array to the end while maintaining the relative order of other elements.

```
public class Main {
    public static void main(String[] args) {
        int[] nums = {0, 1, 0, 3, 12};
        moveZeroes(nums);

        System.out.println("Array After Moving Zeroes:");
        for (int num : nums) {
            System.out.print(num + " ");
        }
    }

    public static void moveZeroes(int[] nums) {
        int index = 0;

        for (int num : nums) {
            if (num != 0) {
                nums[index++] = num;
            }
        }

        while (index < nums.length) {
            nums[index++] = 0;
        }
    }
}
```

PROGRAM : Reverse a Linked List: Write a function to reverse a singly linked list.

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
    }
}

public class Main {
    public static void main(String[] args) {
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);

        System.out.println("Original Linked List:");
        printList(head);

        head = reverseList(head);

        System.out.println("Reversed Linked List:");
        printList(head);
    }

    public static ListNode reverseList(ListNode head) {
        ListNode prev = null;

        while (head != null) {
```

```
            ListNode nextNode = head.next;
            head.next = prev;
            prev = head;
            head = nextNode;
        }

        return prev;
    }

    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }
}
```

PROGRAM : Detect a Cycle in a Linked List: Write a function to detect if a cycle exists in a linked list.

```
public class Main {
   public static void main(String[] args) {
      ListNode head = new ListNode(3);
      head.next = new ListNode(2);
      head.next.next = new ListNode(0);
      head.next.next.next = new ListNode(-4);
      head.next.next.next.next = head.next; // Creates a cycle

      if (hasCycle(head)) {
         System.out.println("Cycle detected in the linked list.");
      } else {
         System.out.println("No cycle detected in the linked list.");
      }
   }

   public static boolean hasCycle(ListNode head) {
      ListNode slow = head, fast = head;

      while (fast != null && fast.next != null) {
         slow = slow.next;
```

```
            fast = fast.next.next;

            if (slow == fast) {
                return true;
            }
        }

        return false;
    }
}
```

PROGRAM : Find the Middle of a Linked List: Write a function to find the middle element of a linked list.

```
public class Main {
  public static void main(String[] args) {
    ListNode head = new ListNode(1);
    head.next = new ListNode(2);
    head.next.next = new ListNode(3);
    head.next.next.next = new ListNode(4);
    head.next.next.next.next = new ListNode(5);

    ListNode middle = findMiddle(head);

    System.out.println("Middle Element of the Linked List: " + middle.val);
  }

  public static ListNode findMiddle(ListNode head) {
    ListNode slow = head, fast = head;

    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;

       }

       return slow;
    }
  }
```

PROGRAM : Merge Two Sorted Linked Lists: Write a function to merge two sorted linked lists into one sorted linked list.

```
public class Main {
    public static void main(String[] args) {
        ListNode l1 = new ListNode(1);
        l1.next = new ListNode(3);
        l1.next.next = new ListNode(5);

        ListNode l2 = new ListNode(2);
        l2.next = new ListNode(4);
        l2.next.next = new ListNode(6);

        ListNode merged = mergeTwoLists(l1, l2);

        System.out.println("Merged Linked List:");
        printList(merged);
    }

    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        while (l1 != null && l2 != null) {
```

```java
            if (l1.val <= l2.val) {
                current.next = l1;
                l1 = l1.next;
            } else {
                current.next = l2;
                l2 = l2.next;
            }
            current = current.next;
        }

        current.next = (l1 != null) ? l1 : l2;

        return dummy.next;
    }

    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }
}
```

PROGRAM : Remove Nth Node from End of List: Write a function to remove the Nth node from the start/end of a linked list.

```
public class Main {
   public static void main(String[] args) {
      ListNode head = new ListNode(1);
      head.next = new ListNode(2);
      head.next.next = new ListNode(3);
      head.next.next.next = new ListNode(4);
      head.next.next.next.next = new ListNode(5);

      int n = 2;
      head = removeNthFromEnd(head, n);

      System.out.println("Linked List After Removing " + n + "th Node From End:");
      printList(head);
   }

   public static ListNode removeNthFromEnd(ListNode head, int n) {
      ListNode dummy = new ListNode(0);
      dummy.next = head;
      ListNode first = dummy;
      ListNode second = dummy;

      for (int i = 0; i <= n; i++) {
         first = first.next;
      }

      while (first != null) {
         first = first.next;
         second = second.next;
      }
```

```
        second.next = second.next.next;

        return dummy.next;
    }

    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }
}
```

PROGRAM : Implement a Stack Using Arrays/Lists: Write a function to implement a stack using an array or list with basic operations: push, pop, peek, and isEmpty.

```
class Stack {
  private int[] stack;
  private int top;

  public Stack(int size) {
    stack = new int[size];
    top = -1;
  }

  public void push(int x) {
    if (top == stack.length - 1) {
      System.out.println("Stack Overflow");
      return;
    }
    stack[++top] = x;
  }

  public int pop() {
    if (top == -1) {
      System.out.println("Stack Underflow");
      return -1;
    }
    return stack[top--];
  }

  public int peek() {
    if (top == -1) {
      System.out.println("Stack is Empty");
      return -1;
    }
    return stack[top];
```

```java
    }

    public boolean isEmpty() {
        return top == -1;
    }
}

public class Main {
    public static void main(String[] args) {
        Stack stack = new Stack(5);

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Top Element: " + stack.peek());

        System.out.println("Popped: " + stack.pop());
        System.out.println("Popped: " + stack.pop());

        System.out.println("Is Stack Empty? " + stack.isEmpty());
    }
}
```

PROGRAM : Implement a Stack Using Linked List: Write a function to implement a stack using a linked list with basic operations: push, pop, peek, and isEmpty.

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
};
class Stack {
private:
Node* top;
public:
Stack() {
top = nullptr;
}
void push(int x) {
Node* newNode = new Node();
newNode->data = x;
newNode->next = top;
top = newNode;
}
int pop() {
if (isEmpty()) {
cout << "Stack Underflow!" << endl;
return -1;
}
int poppedValue = top->data;
Node* temp = top;
top = top->next;
delete temp;
```

```
return poppedValue;
}
int peek() {
if (isEmpty()) {
cout << "Stack is empty!" << endl;
return -1;
}
return top->data;
}
bool isEmpty() {
return top == nullptr;
}
};
```

PROGRAM : Check for Balanced Parentheses: Write a function to check if a string containing parentheses is balanced.

```
#include <iostream>
#include <stack>
using namespace std;
bool isBalanced(string expr) {
stack<char> s;
for (char ch : expr) {
if (ch == '(' || ch == '{' || ch == '[') {
s.push(ch);
} else if (ch == ')' || ch == '}' || ch == ']') {
if (s.empty() || (ch == ')' && s.top() != '(') || (ch == '}' && s.top() !=
'{') || (ch == ']' && s.top() != '[')) {
return false;
}
s.pop();
}
}
return s.empty();}
```

PROGRAM : Evaluate Postfix Expression: Write a function to evaluate a given postfix expression.

```
#include <iostream>
#include <stack>
#include <cctype>
using namespace std;
int evaluatePostfix(string expr) {
stack<int> s;
for (char ch : expr) {
if (isdigit(ch)) {
s.push(ch - '0');
} else {
int b = s.top(); s.pop();
int a = s.top(); s.pop();
switch (ch) {
case '+': s.push(a + b); break;
case '-': s.push(a - b); break;
case '*': s.push(a * b); break;
case '/': s.push(a / b); break;
}
}
}
return s.top();
}
```

PROGRAM : Next Greater Element: Write a function to find the next greater element for each element.

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
vector<int> nextGreaterElement(vector<int>& nums) {
vector<int> result(nums.size(), -1);
stack<int> s;
for (int i = 0; i < nums.size(); ++i) {
while (!s.empty() && nums[s.top()] < nums[i]) {
result[s.top()] = nums[i];
s.pop();
}
s.push(i);
}
return result;
}
};
```

PROGRAM : Implement a Queue Using Arrays/Lists: Write a function to implement a queue using an array or list with basic operations: enqueue, dequeue, front, and isEmpty.

```
#include <iostream>
#include <vector>
using namespace std;
class Queue {
private:
vector<int> arr;
public:
void enqueue(int x) {
arr.push_back(x);
}
int dequeue() {
if (isEmpty()) {
cout << "Queue Underflow!" << endl;
return -1;
}
int frontElement = arr[0];
arr.erase(arr.begin());
return frontElement;
}
int front() {
if (isEmpty()) {
cout << "Queue is empty!" << endl;
return -1;
}
return arr[0];
}
bool isEmpty() {
return arr.empty();
}
};
```

PROGRAM : Implement a Queue Using Linked List: Write a function to implement a queue using a linked list with basic operations: enqueue, dequeue, front, and isEmpty.

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
};
class Queue {
private:
Node* frontNode;
Node* rearNode;
public:
Queue() {
frontNode = rearNode = nullptr;
}
void enqueue(int x) {
Node* newNode = new Node();
newNode->data = x;
newNode->next = nullptr;
if (rearNode) {
rearNode->next = newNode;
}
rearNode = newNode;
if (!frontNode) {
frontNode = rearNode;
}
}
int dequeue() {
if (isEmpty()) {
cout << "Queue Underflow!" << endl;
return -1;
}
int frontValue = frontNode->data;
Node* temp = frontNode;
frontNode = frontNode->next;
delete temp;
if (!frontNode) {
rearNode = nullptr;
}
return frontValue;
}
int front() {
if (isEmpty()) {
cout << "Queue is empty!" << endl;
return -1;
}
return frontNode->data;
```

```
}
bool isEmpty() {
return frontNode == nullptr;
}
};
```

PROGRAM : Implement a Circular Queue: Write a function to implement a circular queue with basic operations: enqueue, dequeue, front, rear, and isEmpty.

```
#include <iostream>
using namespace std;
class CircularQueue {
private:
int* arr;
int size, front, rear, count;
public:
CircularQueue(int n) {
size = n;
arr = new int[n];
front = rear = count = 0;
}
void enqueue(int x) {
if (count == size) {
cout << "Queue Overflow!" << endl;
return;
}
arr[rear] = x;
rear = (rear + 1) % size;
count++;
}
int dequeue() {
if (isEmpty()) {
cout << "Queue Underflow!" << endl;
return -1;
}
int frontValue = arr[front];
front = (front + 1) % size;
count--;
return frontValue;
}
int frontElement() {
if (isEmpty()) {
cout << "Queue is empty!" << endl;
return -1;
}
return arr[front];
}
bool isEmpty() {
return count == 0;
}
};
```

PROGRAM : Generate Binary Numbers from 1 to N: Write a function to generate binary numbers from1 to N using a queue.

```
#include <iostream>
#include <queue>
using namespace std;
void generateBinary(int n) {
queue<string> q;
q.push("1");
while (n--) {
string curr = q.front();
q.pop();
cout << curr << " ";
q.push(curr + "0");
q.push(curr + "1");
}
}
```

PROGRAM : Implement a Queue Using Stacks: Write a function to implement a queue using two stacks.

```
#include <iostream>
#include <stack>
using namespace std;
class QueueUsingStacks {
private:
stack<int> s1, s2;
public:
void enqueue(int x) {
s1.push(x);
}
int dequeue() {
if (isEmpty()) {
cout << "Queue Underflow!" << endl;
return -1;
}
if (s2.empty()) {
while (!s1.empty()) {
s2.push(s1.top());
s1.pop();
}
}
int frontValue = s2.top();
s2.pop();
return frontValue;
}
bool isEmpty() {
return s1.empty() && s2.empty();
}
};
```

PROGRAM: Implement a Binary Tree: Write a class to implement a basic binary tree with insert, delete.

```
class BinaryTree {
  class Node {
    int key;
    Node left, right;

    public Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  public BinaryTree() {
    root = null;
  }

  // Insert a node
  void insert(int key) {
    root = insertRec(root, key);
  }

  Node insertRec(Node root, int key) {
    if (root == null) {
      root = new Node(key);
      return root;
    }
    if (key < root.key)
      root.left = insertRec(root.left, key);
```

```
    else if (key > root.key)
        root.right = insertRec(root.right, key);
    return root;
}

// Delete a node
void delete(int key) {
    root = deleteRec(root, key);
}

Node deleteRec(Node root, int key) {
    if (root == null) return root;

    if (key < root.key)
        root.left = deleteRec(root.left, key);
    else if (key > root.key)
        root.right = deleteRec(root.right, key);
    else {
        // Node with only one child or no child
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // Node with two children: Get the inorder successor (smallest in the right subtree)
        root.key = minValue(root.right);
        root.right = deleteRec(root.right, root.key);
    }
    return root;
}

int minValue(Node root) {
```

```
    int minValue = root.key;
    while (root.left != null) {
        minValue = root.left.key;
        root = root.left;
    }
    return minValue;
}

// Inorder traversal
void inorder() {
    inorderRec(root);
}

void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Main method to test the tree
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();

    // Insert nodes
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);

    System.out.println("Inorder traversal:");
```

```
        tree.inorder();

        System.out.println("\n\nDelete 20");
        tree.delete(20);
        System.out.println("Inorder traversal:");
        tree.inorder();

        System.out.println("\n\nDelete 30");
        tree.delete(30);
        System.out.println("Inorder traversal:");
        tree.inorder();
    }
}
```

PROGRAM: Implement a Binary Tree: Write a class to implement a basic binary tree with insert, delete.

```
class InorderTraversal {
  class Node {
    int key;
    Node left, right;

    Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  void insert(int key) {
    root = insertRec(root, key);
  }

  Node insertRec(Node root, int key) {
    if (root == null) {
      root = new Node(key);
      return root;
    }
    if (key < root.key) root.left = insertRec(root.left, key);
    else if (key > root.key) root.right = insertRec(root.right, key);
    return root;
  }

  void inorder() {
    System.out.print("Inorder Traversal: ");
```

```
        inorderRec(root);
        System.out.println();
    }

    void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.key + " ");
            inorderRec(root.right);
        }
    }

    public static void main(String[] args) {
        InorderTraversal tree = new InorderTraversal();
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);
        tree.inorder();
    }
}
```

PROGRAM: Preorder Traversal: Write a function to perform preorder traversal of a binary tree.

```
class PreorderTraversal {
    class Node {
        int key;
        Node left, right;

        Node(int item) {
            key = item;
            left = right = null;
        }
    }

    Node root;

    void insert(int key) {
        root = insertRec(root, key);
    }

    Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }
        if (key < root.key) root.left = insertRec(root.left, key);
        else if (key > root.key) root.right = insertRec(root.right, key);
        return root;
    }

    void preorder() {
        System.out.print("Preorder Traversal: ");
```

```java
        preorderRec(root);
        System.out.println();
    }

    void preorderRec(Node root) {
        if (root != null) {
            System.out.print(root.key + " ");
            preorderRec(root.left);
            preorderRec(root.right);
        }
    }

    public static void main(String[] args) {
        PreorderTraversal tree = new PreorderTraversal();
        tree.insert(50); tree.insert(30); tree.insert(20);
        tree.insert(40); tree.insert(70); tree.insert(60); tree.insert(80);
        tree.preorder();
    }
}
```

PROGRAM: Postorder Traversal: Write a function to perform postorder traversal of a binary tree.

```
class PostorderTraversal {
  class Node {
    int key;
    Node left, right;

    Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  void insert(int key) {
    root = insertRec(root, key);
  }

  Node insertRec(Node root, int key) {
    if (root == null) {
      root = new Node(key);
      return root;
    }
    if (key < root.key) root.left = insertRec(root.left, key);
    else if (key > root.key) root.right = insertRec(root.right, key);
    return root;
  }

  void postorder() {
    System.out.print("Postorder Traversal: ");
```

```
        postorderRec(root);
        System.out.println();
    }

    void postorderRec(Node root) {
        if (root != null) {
            postorderRec(root.left);
            postorderRec(root.right);
            System.out.print(root.key + " ");
        }
    }

    public static void main(String[] args) {
        PostorderTraversal tree = new PostorderTraversal();
        tree.insert(50); tree.insert(30); tree.insert(20);
        tree.insert(40); tree.insert(70); tree.insert(60); tree.insert(80);
        tree.postorder();
    }
}
```

PROGRAM: Level Order Traversal: Write a function to perform level order traversal of a binary tree.

```
import java.util.LinkedList;
import java.util.Queue;

class LevelOrderTraversal {
  class Node {
    int key;
    Node left, right;

    Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  void insert(int key) {
    root = insertRec(root, key);
  }

  Node insertRec(Node root, int key) {
    if (root == null) {
      root = new Node(key);
      return root;
    }
    if (key < root.key) root.left = insertRec(root.left, key);
    else if (key > root.key) root.right = insertRec(root.right, key);
    return root;
```

```
    }

    void levelOrder() {
        System.out.print("Level Order Traversal: ");
        if (root == null) return;

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

        while (!queue.isEmpty()) {
            Node tempNode = queue.poll();
            System.out.print(tempNode.key + " ");
            if (tempNode.left != null) queue.add(tempNode.left);
            if (tempNode.right != null) queue.add(tempNode.right);
        }
        System.out.println();
    }

    public static void main(String[] args) {
        LevelOrderTraversal tree = new LevelOrderTraversal();
        tree.insert(50); tree.insert(30); tree.insert(20);
        tree.insert(40); tree.insert(70); tree.insert(60); tree.insert(80);
        tree.levelOrder();
    }
}
```

PROGRAM: Height of a Binary Tree: Write a function to find the height of a binary tree.

```
class BinaryTreeHeight {
  class Node {
    int key;
    Node left, right;

    public Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  // Function to insert nodes in the binary tree
  void insert(int key) {
    root = insertRec(root, key);
  }

  Node insertRec(Node root, int key) {
    if (root == null) {
      root = new Node(key);
      return root;
    }
    if (key < root.key)
      root.left = insertRec(root.left, key);
    else if (key > root.key)
      root.right = insertRec(root.right, key);
    return root;
```

```
    }

    // Function to calculate the height of the binary tree
    int height(Node node) {
        if (node == null) {
            return 0; // Base case: height of an empty tree is 0
        }

        // Recursively find the height of left and right subtrees
        int leftHeight = height(node.left);
        int rightHeight = height(node.right);

        // Height of the tree is the maximum of the two subtree heights + 1
        return Math.max(leftHeight, rightHeight) + 1;
    }

    public static void main(String[] args) {
        BinaryTreeHeight tree = new BinaryTreeHeight();

        // Insert nodes into the binary tree
        tree.insert(50);
        tree.insert(30);
        tree.insert(20);
        tree.insert(40);

        // Calculate and print the height of the binary tree
        int treeHeight = tree.height(tree.root);
        System.out.println("Height of the Binary Tree: " + treeHeight);
    }
}
```

PROGRAM: Diameter of a Binary Tree: Write a function to find the diameter of a binary tree.

```
class DiameterBinaryTree {
  class Node {
    int key;
    Node left, right;

    public Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  int diameter(Node root) {
    int[] diameter = new int[1]; // To store the result
    height(root, diameter);
    return diameter[0];
  }

  int height(Node node, int[] diameter) {
    if (node == null) {
      return 0;
    }
    int leftHeight = height(node.left, diameter);
    int rightHeight = height(node.right, diameter);
    diameter[0] = Math.max(diameter[0], leftHeight + rightHeight);
    return Math.max(leftHeight, rightHeight) + 1;
  }
```

```java
    public static void main(String[] args) {
        DiameterBinaryTree tree = new DiameterBinaryTree();
        tree.root = tree.new Node(1);
        tree.root.left = tree.new Node(2);
        tree.root.right = tree.new Node(3);
        tree.root.left.left = tree.new Node(4);
        tree.root.left.right = tree.new Node(5);

        System.out.println("Diameter of the tree: " + tree.diameter(tree.root));
    }
}
```

PROGRAM: Check if a Binary Tree is Balanced: Write a function to check if a binary tree is height.

```
class BalancedBinaryTree {
  class Node {
    int key;
    Node left, right;

    public Node(int item) {
      key = item;
      left = right = null;
    }
  }

  Node root;

  boolean isBalanced(Node root) {
    return checkHeight(root) != -1;
  }

  int checkHeight(Node node) {
    if (node == null) return 0;

    int leftHeight = checkHeight(node.left);
    if (leftHeight == -1) return -1;

    int rightHeight = checkHeight(node.right);
    if (rightHeight == -1) return -1;

    if (Math.abs(leftHeight - rightHeight) > 1) return -1;

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

```java
    }

    public static void main(String[] args) {
        BalancedBinaryTree tree = new BalancedBinaryTree();
        tree.root = tree.new Node(1);
        tree.root.left = tree.new Node(2);
        tree.root.right = tree.new Node(3);
        tree.root.left.left = tree.new Node(4);
        tree.root.left.right = tree.new Node(5);

        System.out.println("Is the tree balanced? " + tree.isBalanced(tree.root));
    }
}
```

PROGRAM: Lowest Common Ancestor: Write a function to find the lowest common ancestor of two nodes in a binary tree.

```
class LCABinaryTree {
    class Node {
        int key;
        Node left, right;

        public Node(int item) {
            key = item;
            left = right = null;
        }
    }

    Node root;

    Node findLCA(Node root, int n1, int n2) {
        if (root == null) return null;

        if (root.key == n1 || root.key == n2) return root;

        Node leftLCA = findLCA(root.left, n1, n2);
        Node rightLCA = findLCA(root.right, n1, n2);

        if (leftLCA != null && rightLCA != null) return root;

        return (leftLCA != null) ? leftLCA : rightLCA;
    }

    public static void main(String[] args) {
        LCABinaryTree tree = new LCABinaryTree();
```

```
        tree.root = tree.new Node(3);
        tree.root.left = tree.new Node(5);
        tree.root.right = tree.new Node(1);
        tree.root.left.left = tree.new Node(6);
        tree.root.left.right = tree.new Node(2);

        System.out.println("LCA of 5 and 1: " + tree.findLCA(tree.root, 5, 1).key);
        System.out.println("LCA of 6 and 2: " + tree.findLCA(tree.root, 6, 2).key);
    }
}
```

PROGRAM: Implement Graph Using Adjacency List: Write a class to implement a basic graph using an adjacency list with methods to add vertices and edges.

```
import java.util.*;


class Graph {

    private Map<Integer, List<Integer>> adjList;


    public Graph() {

        adjList = new HashMap<>();

    }


    // Add a vertex

    void addVertex(int v) {

        adjList.putIfAbsent(v, new ArrayList<>());

    }


    // Add an edge

    void addEdge(int src, int dest) {

        adjList.putIfAbsent(src, new ArrayList<>());

        adjList.putIfAbsent(dest, new ArrayList<>());

        adjList.get(src).add(dest);

        adjList.get(dest).add(src); // For undirected graph

    }


    // Print the graph

    void printGraph() {

        for (var entry : adjList.entrySet()) {

            System.out.print(entry.getKey() + " -> ");

            for (int neighbor : entry.getValue()) {
```

```
            System.out.print(neighbor + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    Graph graph = new Graph();

    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);

    graph.printGraph();
}
}
```

PROGRAM: Breadth-First Search (BFS): Write a function to perform BFS on a graph from a given start vertex.

```
import java.util.*;

class GraphBFS {
  private Map<Integer, List<Integer>> adjList = new HashMap<>();

  void addEdge(int src, int dest) {
    adjList.putIfAbsent(src, new ArrayList<>());
    adjList.putIfAbsent(dest, new ArrayList<>());
    adjList.get(src).add(dest);
    adjList.get(dest).add(src); // For undirected graph
  }

  void bfs(int startVertex) {
    Set<Integer> visited = new HashSet<>();
    Queue<Integer> queue = new LinkedList<>();

    queue.add(startVertex);
    visited.add(startVertex);

    while (!queue.isEmpty()) {
      int vertex = queue.poll();
      System.out.print(vertex + " ");

      for (int neighbor : adjList.get(vertex)) {
        if (!visited.contains(neighbor)) {
          queue.add(neighbor);
          visited.add(neighbor);
        }
```

```java
            }
        }
    }

    public static void main(String[] args) {
        GraphBFS graph = new GraphBFS();
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(3, 5);

        System.out.print("BFS starting from vertex 1: ");
        graph.bfs(1);
    }
}
```

PROGRAM: Depth-First Search (DFS): Write a function to perform DFS on a graph from a given start vertex.

```
import java.util.*;

class GraphDFS {
   private Map<Integer, List<Integer>> adjList = new HashMap<>();

   void addEdge(int src, int dest) {
      adjList.putIfAbsent(src, new ArrayList<>());
      adjList.putIfAbsent(dest, new ArrayList<>());
      adjList.get(src).add(dest);
      adjList.get(dest).add(src); // For undirected graph
   }

   void dfs(int startVertex) {
      Set<Integer> visited = new HashSet<>();
      dfsHelper(startVertex, visited);
   }

   private void dfsHelper(int vertex, Set<Integer> visited) {
      visited.add(vertex);
      System.out.print(vertex + " ");

      for (int neighbor : adjList.get(vertex)) {
         if (!visited.contains(neighbor)) {
            dfsHelper(neighbor, visited);
         }
      }
   }
```

```
    public static void main(String[] args) {
        GraphDFS graph = new GraphDFS();
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(3, 5);

        System.out.print("DFS starting from vertex 1: ");
        graph.dfs(1);
    }
}
```

PROGRAM: Detect Cycle in an Undirected Graph: Write a function to detect if there is a cycle in an undirected graph.

```
import java.util.*;

class GraphCycleDetection {
   private Map<Integer, List<Integer>> adjList = new HashMap<>();

   void addEdge(int src, int dest) {
      adjList.putIfAbsent(src, new ArrayList<>());
      adjList.putIfAbsent(dest, new ArrayList<>());
      adjList.get(src).add(dest);
      adjList.get(dest).add(src);
   }

   boolean hasCycle() {
      Set<Integer> visited = new HashSet<>();
      for (int vertex : adjList.keySet()) {
         if (!visited.contains(vertex)) {
            if (detectCycle(vertex, -1, visited)) {
               return true;
            }
         }
      }
      return false;
   }

   private boolean detectCycle(int current, int parent, Set<Integer> visited) {
      visited.add(current);

      for (int neighbor : adjList.get(current)) {
```

```
            if (!visited.contains(neighbor)) {
                if (detectCycle(neighbor, current, visited)) {
                    return true;
                }
            } else if (neighbor != parent) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        GraphCycleDetection graph = new GraphCycleDetection();
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 3);

        System.out.println("Does the graph have a cycle? " + graph.hasCycle());
    }
}
```

PROGRAM: Connected Components in an Undirected Graph: Write a function to find the number of connected components in an undirected graph.

```
import java.util.*;

class GraphConnectedComponents {
  private Map<Integer, List<Integer>> adjList = new HashMap<>();

  void addEdge(int src, int dest) {
    adjList.putIfAbsent(src, new ArrayList<>());
    adjList.putIfAbsent(dest, new ArrayList<>());
    adjList.get(src).add(dest);
    adjList.get(dest).add(src);
  }

  int countConnectedComponents() {
    Set<Integer> visited = new HashSet<>();
    int count = 0;

    for (int vertex : adjList.keySet()) {
      if (!visited.contains(vertex)) {
        dfs(vertex, visited);
        count++;
      }
    }
    return count;
  }

  private void dfs(int vertex, Set<Integer> visited) {
    visited.add(vertex);
```

```
        for (int neighbor : adjList.get(vertex)) {
            if (!visited.contains(neighbor)) {
                dfs(neighbor, visited);
            }
        }
    }

    public static void main(String[] args) {
        GraphConnectedComponents graph = new GraphConnectedComponents();
        graph.addEdge(1, 2);
        graph.addEdge(3, 4);
        graph.addEdge(5, 6);

        System.out.println("Number of connected components: " +
graph.countConnectedComponents());
    }
}
```

PROGRAM: Find MST Using Kruskal's Algorithm: Write a function to find the Minimum Spanning Tree of a graph using Kruskal's algorithm.

```
import java.util.*;

class KruskalMST {
  class Edge implements Comparable<Edge> {
    int src, dest, weight;

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

    public int compareTo(Edge other) {
      return this.weight - other.weight;
    }
  }

  class Subset {
    int parent, rank;
  }

  int vertices;
  List<Edge> edges = new ArrayList<>();

  KruskalMST(int vertices) {
    this.vertices = vertices;
  }
```

```
void addEdge(int src, int dest, int weight) {
    edges.add(new Edge(src, dest, weight));
}

int find(Subset[] subsets, int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

void union(Subset[] subsets, int x, int y) {
    int xRoot = find(subsets, x);
    int yRoot = find(subsets, y);

    if (subsets[xRoot].rank < subsets[yRoot].rank) {
        subsets[xRoot].parent = yRoot;
    } else if (subsets[xRoot].rank > subsets[yRoot].rank) {
        subsets[yRoot].parent = xRoot;
    } else {
        subsets[yRoot].parent = xRoot;
        subsets[xRoot].rank++;
    }
}

void kruskalMST() {
    List<Edge> result = new ArrayList<>();
    Collections.sort(edges);

    Subset[] subsets = new Subset[vertices];
    for (int i = 0; i < vertices; i++) {
```

```java
            subsets[i] = new Subset();
            subsets[i].parent = i;
            subsets[i].rank = 0;
        }

        for (Edge edge : edges) {
            int x = find(subsets, edge.src);
            int y = find(subsets, edge.dest);

            if (x != y) {
                result.add(edge);
                union(subsets, x, y);
            }
        }

        System.out.println("Edges in MST:");
        for (Edge edge : result) {
            System.out.println(edge.src + " - " + edge.dest + ": " + edge.weight);
        }
    }

    public static void main(String[] args) {
        KruskalMST graph = new KruskalMST(4);

        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.kruskalMST();
    }
}
```

PROGRAM: Find MST Using Prim's Algorithm: Write a function to find the Minimum Spanning Tree of a graph using Prim's algorithm.

```
import java.util.*;

class PrimMST {
   int vertices;

   PrimMST(int vertices) {
      this.vertices = vertices;
   }

   void primMST(int[][] graph) {
      int[] parent = new int[vertices];
      int[] key = new int[vertices];
      boolean[] mstSet = new boolean[vertices];

      Arrays.fill(key, Integer.MAX_VALUE);
      key[0] = 0;
      parent[0] = -1;

      for (int count = 0; count < vertices - 1; count++) {
         int u = minKey(key, mstSet);
         mstSet[u] = true;

         for (int v = 0; v < vertices; v++) {
            if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {
               parent[v] = u;
               key[v] = graph[u][v];
            }
         }   }
```

```java
    printMST(parent, graph);
  }

  int minKey(int[] key, boolean[] mstSet) {
    int min = Integer.MAX_VALUE, minIndex = -1;

    for (int v = 0; v < vertices; v++) {
      if (!mstSet[v] && key[v] < min) {
        min = key[v];
        minIndex = v;
      }
    }
    return minIndex;
  }

  void printMST(int[] parent, int[][] graph) {
    System.out.println("Edge \tWeight");
    for (int i = 1; i < vertices; i++) {
      System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);
    }
  }

  public static void main(String[] args) {
    PrimMST graph = new PrimMST(5);
    int[][] matrix = {
      {0, 2, 0, 6, 0},
      {2, 0, 3, 8, 5},
      {0, 3, 0, 0, 7},
    };
graph.primMST(matrix);
  }  }
```

PROGRAM: Fibonacci Sequence: Write a function to compute the nth Fibonacci number using dynamic programming.

```
class FibonacciDP {
    int fibonacci(int n) {
        if (n <= 1) return n;

        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;

        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }

    public static void main(String[] args) {
        FibonacciDP fib = new FibonacciDP();
        int n = 10;
        System.out.println("Fibonacci number at position " + n + ": " +
fib.fibonacci(n));
    }
}
```

PROGRAM: Climbing Stairs: Write a function to determine how many distinct ways there are to climb a staircase with n steps if you can climb either 1 or 2 steps at a time.

```
class ClimbingStairs {
    int countWays(int n) {
        if (n <= 1) return 1;

        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }

    public static void main(String[] args) {
        ClimbingStairs stairs = new ClimbingStairs();
        int n = 5;  // Example: 5 steps
        System.out.println("Number of ways to climb " + n + " steps: " +
stairs.countWays(n));
    }
}
```

PROGRAM: Min Cost Climbing Stairs: Write a function to determine the minimum cost to reach the top of a staircase given a list of costs associated with each step.

```
class MinCostClimbingStairs {
    int minCost(int[] cost) {
        int n = cost.length;
        if (n == 0) return 0;
        if (n == 1) return cost[0];

        int[] dp = new int[n];
        dp[0] = cost[0];
        dp[1] = cost[1];

        for (int i = 2; i < n; i++) {
            dp[i] = cost[i] + Math.min(dp[i - 1], dp[i - 2]);
        }

        return Math.min(dp[n - 1], dp[n - 2]);
    }

    public static void main(String[] args) {
        MinCostClimbingStairs stairs = new MinCostClimbingStairs();
        int[] cost = {10, 15, 20};  // Example: Costs at each step
        System.out.println("Minimum cost to reach the top: " +
stairs.minCost(cost));
    }
}
```

PROGRAM: House Robber: Write a function to determine the maximum amount of money you can rob from a row of houses without robbing two adjacent houses.

```
class HouseRobber {
    int rob(int[] nums) {
        if (nums.length == 0) return 0;
        if (nums.length == 1) return nums[0];

        int prev2 = 0, prev1 = 0;
        for (int num : nums) {
            int temp = prev1;
            prev1 = Math.max(prev1, prev2 + num);
            prev2 = temp;
        }

        return prev1;
    }

    public static void main(String[] args) {
        HouseRobber robber = new HouseRobber();
        int[] nums = {2, 7, 9, 3, 1};  // Example: Amounts of money in houses
        System.out.println("Maximum money that can be robbed: " +
robber.rob(nums));
    }
}
```

PROGRAM: Maximum Subarray Sum (Kadane's Algorithm): Write a function to find the contiguous subarray with the maximum sum.

```
class MaximumSubarraySum {
    int maxSubArray(int[] nums) {
        int maxSoFar = nums[0];
        int maxEndingHere = nums[0];

        for (int i = 1; i < nums.length; i++) {
            maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
            maxSoFar = Math.max(maxSoFar, maxEndingHere);
        }

        return maxSoFar;
    }

    public static void main(String[] args) {
        MaximumSubarraySum maxSum = new MaximumSubarraySum();
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};  // Example array
        System.out.println("Maximum subarray sum: " +
maxSum.maxSubArray(nums));
    }
}
```

PROGRAM: Activity Selection: Given a set of activities with start and end times, select the maximum number of activities that do not overlap.

```
import java.util.*;

class ActivitySelection {
    static class Activity {
        int start, end;

        Activity(int start, int end) {
            this.start = start;
            this.end = end;
        }
    }

    void selectActivities(Activity[] activities) {
        Arrays.sort(activities, Comparator.comparingInt(a -> a.end));  // Sort by
end time

        List<Activity> selected = new ArrayList<>();
        selected.add(activities[0]);

        int lastEndTime = activities[0].end;

        for (int i = 1; i < activities.length; i++) {
            if (activities[i].start >= lastEndTime) {
                selected.add(activities[i]);
```

```
                lastEndTime = activities[i].end;
            }
        }

        System.out.println("Selected activities:");
        for (Activity activity : selected) {
            System.out.println("Start: " + activity.start + ", End: " + activity.end);
        }
    }

    public static void main(String[] args) {
        ActivitySelection selection = new ActivitySelection();
        Activity[] activities = {
            new Activity(1, 3),
            new Activity(2, 5),
            new Activity(4, 7),
            new Activity(6, 8),
            new Activity(5, 9)
        };

        selection.selectActivities(activities);
    }
}
```

PROGRAM: Fractional Knapsack Problem: Given weights and values of items and the maximum capacity of a knapsack, determine the maximum value that can be obtained by including fractions of items.

```
import java.util.*;

class FractionalKnapsack {
    static class Item {
        int weight, value;

        Item(int weight, int value) {
            this.weight = weight;
            this.value = value;
        }
    }

    double knapSack(int W, Item[] items) {
        Arrays.sort(items, (a, b) -> Double.compare(b.value * 1.0 / b.weight,
a.value * 1.0 / a.weight));

        double totalValue = 0;
        for (Item item : items) {
            if (W == 0) break;

            if (item.weight <= W) {
                totalValue += item.value;
                W -= item.weight;
```

```java
            } else {
                totalValue += item.value * ((double) W / item.weight);
                break;
            }
        }

        return totalValue;
    }

    public static void main(String[] args) {
        FractionalKnapsack knapsack = new FractionalKnapsack();
        Item[] items = {
            new Item(60, 100),
            new Item(100, 120),
            new Item(120, 150)
        };

        int W = 50;  // Maximum weight capacity of the knapsack
        System.out.println("Maximum value we can obtain: " +
knapsack.knapSack(W, items));
    }
}
```

PROGRAM: Huffman Coding: Given a set of characters and their frequencies, construct the Huffman Tree to encode the characters.

```
import java.util.*;

class HuffmanCoding {
    static class Node {
        char ch;
        int freq;
        Node left, right;

        Node(char ch, int freq) {
            this.ch = ch;
            this.freq = freq;
            this.left = this.right = null;
        }
    }

    static class MinHeap {
        List<Node> heap;

        MinHeap() {
            heap = new ArrayList<>();
        }

        void insert(Node node) {
            heap.add(node);
            int i = heap.size() - 1;
            while (i > 0 && heap.get(i).freq < heap.get((i - 1) / 2).freq) {
                Collections.swap(heap, i, (i - 1) / 2);
                i = (i - 1) / 2;
```

```
        }
    }

    Node extractMin() {
        Node min = heap.get(0);
        heap.set(0, heap.get(heap.size() - 1));
        heap.remove(heap.size() - 1);
        minHeapify(0);
        return min;
    }

    void minHeapify(int i) {
        int left = 2 * i + 1, right = 2 * i + 2;
        int smallest = i;

        if (left < heap.size() && heap.get(left).freq < heap.get(smallest).freq) {
            smallest = left;
        }

        if (right < heap.size() && heap.get(right).freq < heap.get(smallest).freq) {
            smallest = right;
        }

        if (smallest != i) {
            Collections.swap(heap, i, smallest);
            minHeapify(smallest);
        }
    }
}

void buildHuffmanTree(Map<Character, Integer> freqMap) {
```

```
        MinHeap minHeap = new MinHeap();
        for (Map.Entry<Character, Integer> entry : freqMap.entrySet()) {
            minHeap.insert(new Node(entry.getKey(), entry.getValue()));
        }

        while (minHeap.heap.size() > 1) {
            Node left = minHeap.extractMin();
            Node right = minHeap.extractMin();
            Node merged = new Node('$', left.freq + right.freq);
            merged.left = left;
            merged.right = right;
            minHeap.insert(merged);
        }

        Node root = minHeap.extractMin();
        printCodes(root, "");
    }

    void printCodes(Node root, String code) {
        if (root == null) return;
        if (root.ch != '$') {
            System.out.println(root.ch + ": " + code);
        }
        printCodes(root.left, code + "0");
        printCodes(root.right, code + "1");
    }

    public static void main(String[] args) {
        HuffmanCoding huffman = new HuffmanCoding();
        Map<Character, Integer> freqMap = new HashMap<>();
        freqMap.put('a', 5);
```

```
        freqMap.put('b', 9);
        freqMap.put('c', 12);
        freqMap.put('d', 13);
        freqMap.put('e', 16);
        freqMap.put('f', 45);

        huffman.buildHuffmanTree(freqMap);
    }
}
```

PROGRAM: Job Sequencing Problem: Given a set of jobs, each with a deadline and profit, maximize the total profit by scheduling the jobs to be done before their deadlines.

```
import java.util.*;

class JobSequencing {
   static class Job {
      int id, deadline, profit;

      Job(int id, int deadline, int profit) {
         this.id = id;
         this.deadline = deadline;
         this.profit = profit;
      }
   }

   void jobScheduling(Job[] jobs, int n) {
      Arrays.sort(jobs, (a, b) -> b.profit - a.profit);  // Sort jobs by profit in
descending order

      boolean[] slots = new boolean[n];
      Arrays.fill(slots, false);
      int maxProfit = 0;
      List<Integer> jobSequence = new ArrayList<>();

      for (Job job : jobs) {
```

```
            for (int j = job.deadline - 1; j >= 0; j--) {
                if (!slots[j]) {
                    slots[j] = true;
                    maxProfit += job.profit;
                    jobSequence.add(job.id);
                    break;
                }
            }
        }

        System.out.println("Job sequence: " + jobSequence);
        System.out.println("Total profit: " + maxProfit);
    }

    public static void main(String[] args) {
        JobSequencing scheduling = new JobSequencing();
        Job[] jobs = {
            new Job(1, 4, 20),
            new Job(2, 1, 10),
            new Job(3, 1, 40),
            new Job(4, 1, 30)
        };
        int n = 4;
        scheduling.jobScheduling(jobs, n);
    }
}
```

PROGRAM: Minimum Number of Coins: Given different denominations of coins and an amount, find the minimum number of coins needed to make up that amount.

```
import java.util.*;
class MinimumCoins {
   int minCoins(int[] coins, int amount) {
      int[] dp = new int[amount + 1];
      Arrays.fill(dp, Integer.MAX_VALUE);
      dp[0] = 0;

      for (int i = 1; i <= amount; i++) {
         for (int coin : coins) {
            if (i - coin >= 0 && dp[i - coin] != Integer.MAX_VALUE) {
               dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
         }
      }
      return dp[amount] == Integer.MAX_VALUE ? -1 : dp[amount];
   }
   public static void main(String[] args) {
      MinimumCoins coins = new MinimumCoins();
      int[] coinDenominations = {1, 2, 5};  // Coin denominations
      int amount = 11;
 System.out.println("Minimum coins required:
"coins.minCoins(coinDenominations, amount));
   }
}
```

PROGRAM: N-Queens Problem: Place N queens on an N×N chessboard so that no two queens threaten each other.

```
class NQueens {
    static final int N = 4;

    boolean isSafe(int[][] board, int row, int col) {
        for (int i = 0; i < col; i++) {
            if (board[row][i] == 1) return false;
        }

        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) return false;
        }

        for (int i = row, j = col; j >= 0 && i < N; i++, j--) {
            if (board[i][j] == 1) return false;
        }

        return true;
    }

    boolean solveNQueens(int[][] board, int col) {
        if (col >= N) return true;

        for (int i = 0; i < N; i++) {
```

```
            if (isSafe(board, i, col)) {
                board[i][col] = 1;
                if (solveNQueens(board, col + 1)) return true;
                board[i][col] = 0;
            }
        }

        return false;
    }

    void printSolution(int[][] board) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(board[i][j] + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        NQueens queens = new NQueens();
        int[][] board = new int[N][N];

        if (queens.solveNQueens(board, 0)) {
            queens.printSolution(board);
        } else {
```

```
            System.out.println("Solution does not exist");
        }
    }
}
```