Java Coding Questions

Basic Coding Questions

1. Write a Java program to print "Hello, World!" to the console.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

1. Write a program to find the sum of two numbers entered by the user.

```
import java.util.Scanner;

public class SumOfTwoNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first number: ");
        int num1 = scanner.nextInt();
        System.out.print("Enter the second number: ");
        int num2 = scanner.nextInt();
        int sum = num1 + num2;
        System.out.println("Sum: " + sum);
        scanner.close();
    }
}
```


1. Write a Java program to check if a number is even or odd.

```
import java.util.Scanner;

public class EvenOrOdd {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        String result = (num % 2 == 0) ? "even" : "odd";
        System.out.println(num + " is " + result);
        scanner.close();
    }
}
```

1. Write a function to find the factorial of a given number using recursion.

```
public class Factorial {
    public static void main(String[] args) {
        int number = 5;
        int factorial = findFactorial(number);
        System.out.println("Factorial of " + number + " is: " + factorial);
    }

    public static int findFactorial(int n) {
        if (n == 0 || n == 1) {
            return 1;
        }
        return n * findFactorial(n - 1);
    }
}
```

1. Implement a function to check if a string is a palindrome.

```
public class Palindrome {
    public static void main(String[] args) {
        String str = "radar";
        boolean isPalindrome = checkPalindrome(str);
        System.out.println(str + " is a palindrome: " + isPalindrome);
    }

    public static boolean checkPalindrome(String str) {
        int left = 0;
        int right = str.length() - 1;

        while (left < right) {
            if (str.charAt(left) != str.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }
}
```


1. Write a program to print the Fibonacci series up to a given number.

```
import java.util.Scanner;

public class FibonacciSeries {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of terms: ");
        int n = scanner.nextInt();

        int first = 0;
        int second = 1;
```

```
        System.out.print("Fibonacci Series: " + first + " " + second + " ");

        for (int i = 2; i < n; i++) {
            int next = first + second;
            System.out.print(next + " ");
            first = second;
            second = next;
        }

        scanner.close();
    }
}
```

1. Write a Java function to reverse an array of integers in-place.

```
public class ReverseArray {
    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 4, 5 };
        reverseArray(arr);

        System.out.print("Reversed Array: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }

    public static void reverseArray(int[] arr) {
        int left = 0;
        int right = arr.length - 1;

        while (left < right) {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
    }
}
```


1. Implement a function to find the maximum element in an array.

```
public class MaxElementInArray {
    public static void main(String[] args) {
        int[] arr = { 10, 25, 7, 42, 32 };
        int max = findMaxElement(arr);
        System.out.println("Maximum element in the array: " + max);
    }

    public static int findMaxElement(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
```

```
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }
}
```

1. Write a program to remove duplicates from an array in Java.

```
import java.util.Arrays;

public class RemoveDuplicates {
    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 3, 4, 5, 5, 6 };
        int[] uniqueArray = removeDuplicates(arr);

        System.out.print("Array with duplicates removed: ");
        for (int num : uniqueArray) {
            System.out.print(num + " ");
        }
    }

    public static int[] removeDuplicates(int[] arr) {
        int[] uniqueArray = new int[arr.length];
        int j = 0;

        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] != arr[i + 1]) {
                uniqueArray[j++] = arr[i];
            }
        }
        uniqueArray[j++] = arr[arr.length - 1];

        return Arrays.copyOf(uniqueArray, j);
    }
}
```


1. Implement a function to check if a number is prime.

```
import java.util.Scanner;

public class PrimeNumber {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        boolean isPrime = checkPrime(num);
        System.out.println(num + " is a prime number: " + isPrime);
        scanner.close();
    }
```

```
    public static boolean checkPrime(int num) {
        if (num <= 1) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

1. Write a Java program to swap two numbers without using a temporary variable.

```
public class SwapWithoutTempVariable {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 20;

        System.out.println("Before swapping: num1 = " + num1 + ", num2 = " + num2);

        num1 = num1 + num2;
        num2 = num1 - num2;
        num1 = num1 - num2;

        System.out.println("After swapping: num1 = " + num1 + ", num2 = " + num2);
    }
}
```


1. Implement a function to count

the occurrences of a specific element in an array.

```
public class CountOccurrences {
    public static void main(String[] args) {
        int[] arr = { 2, 3, 4, 2, 2, 5, 6, 2 };
        int element = 2;
        int count = countOccurrences(arr, element);
        System.out.println("Occurrences of " + element + " in the array: " + count);
    }

    public static int countOccurrences(int[] arr, int element) {
        int count = 0;
        for (int num : arr) {
            if (num == element) {
                count++;
            }
        }
        return count;
    }
}
```

1. Write a Java function to calculate the area of a circle given its radius.

```
import java.util.Scanner;

public class CircleArea {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble();
        double area = calculateArea(radius);
        System.out.println("Area of the circle: " + area);
        scanner.close();
    }

    public static double calculateArea(double radius) {
        return Math.PI * radius * radius;
    }
}
```

1. Implement a function to find the second largest element in an array.

```
import java.util.Arrays;

public class SecondLargestElement {
    public static void main(String[] args) {
        int[] arr = { 10, 25, 7, 42, 32 };
        int secondLargest = findSecondLargest(arr);
        System.out.println("Second largest element in the array: " + secondLargest);
    }

    public static int findSecondLargest(int[] arr) {
        Arrays.sort(arr);
        return arr[arr.length - 2];
    }
}
```


1. Write a program to sort an array of integers in ascending order using the Bubble Sort algorithm.

```
import java.util.Arrays;

public class BubbleSort {
    public static void main(String[] args) {
        int[] arr = { 64, 34, 25, 12, 22, 11, 90 };
        bubbleSort(arr);
        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
```

```
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

1. Implement a function to reverse a string in Java.

```
public class ReverseString {
    public static void main(String[] args) {
        String str = "Hello, World!";
        String reversed = reverseString(str);
        System.out.println("Reversed string: " + reversed);
    }

    public static String reverseString(String str) {
        StringBuilder sb = new StringBuilder(str);
        return sb.reverse().toString();
    }
}
```


1. Write a Java program to find the GCD (Greatest Common Divisor) of two numbers.

```
import java.util.Scanner;

public class GCD {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first number: ");
        int num1 = scanner.nextInt();
        System.out.print("Enter the second number: ");
        int num2 = scanner.nextInt();
        int gcd = findGCD(num1, num2);
        System.out.println("GCD of " + num1 + " and " + num2 + " is: " + gcd);
        scanner.close();
    }

    public static int findGCD(int a, int b) {
        if (b == 0) {
            return a;
        }
        return findGCD(b, a % b);
```

```
    }
}
```

1. Implement a function to check if two strings are anagrams.

```
import java.util.Arrays;

public class Anagrams {
    public static void main(String[] args) {
        String str1 = "listen";
        String str2 = "silent";
        boolean areAnagrams = checkAnagrams(str1, str2);
        System.out.println(str1 + " and " + str2 + " are anagrams: " + areAnagrams);
    }

    public static boolean checkAnagrams(String str1, String str2) {
        if (str1.length() != str2.length()) {
            return false;
        }
        char[] chars1 = str1.toCharArray();
        char[] chars2 = str2.toCharArray();
        Arrays.sort(chars1);
        Arrays.sort(chars2);
        return Arrays.equals(chars1, chars2);
    }
}
```


1. Write a Java program to find the factorial of a number using an iterative approach.

```
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        int factorial = findFactorial(number);
        System.out.println("Factorial of " + number + " is: " + factorial);
        scanner.close();
    }

    public static int findFactorial(int n) {
        int factorial = 1;
        for (int i = 2; i <= n; i++) {
            factorial *= i;
        }
        return factorial;
    }
}
```

1. Implement a function to find the sum of digits of a given number.

```
import java.util.Scanner;

public class SumOfDigits {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        int sum = sumOfDigits(number);
        System.out.println("Sum of digits of " + number + " is: " + sum);
        scanner.close();
    }

    public static int sumOfDigits(int num) {
        int sum = 0;
        while (num != 0) {
            sum += num % 10;
            num /= 10;
        }
        return sum;
    }
}
```

1. Write a Java program to find the prime factors of a given number.


```
import java.util.Scanner;

public class PrimeFactors {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        System.out.print("Prime factors of " + number + ": ");
        findPrimeFactors(number);
        scanner.close();
    }

    public static void findPrimeFactors(int num) {
        for (int i = 2; i <= num; i++) {
            while (num % i == 0) {
                System.out.print(i + " ");
                num /= i;
            }
        }
    }
}
```

1. Implement a function to check if a string contains only digits.

```
public class OnlyDigits {
    public static void main(String[] args

) {
        String str = "12345";
        boolean containsOnlyDigits = checkOnlyDigits(str);
        System.out.println(str + " contains only digits: " + containsOnlyDigits);
    }

    public static boolean checkOnlyDigits(String str) {
        return str.matches("\\\\d+");
    }
}
```

1. Write a program to print the Pascal's triangle for a given number of rows.

```
import java.util.Scanner;

public class PascalsTriangle {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of rows: ");
        int numRows = scanner.nextInt();
        printPascalsTriangle(numRows);
        scanner.close();
    }

    public static void printPascalsTriangle(int numRows) {
        for (int i = 0; i < numRows; i++) {
            int num = 1;
            for (int j = 0; j <= i; j++) {
                System.out.print(num + " ");
                num = num * (i - j) / (j + 1);
            }
            System.out.println();
        }
    }
}
```


1. Implement a function to find the intersection of two arrays.

```
import java.util.HashSet;
import java.util.Set;

public class ArrayIntersection {
    public static void main(String[] args) {
        int[] arr1 = { 1, 2, 3, 4, 5 };
        int[] arr2 = { 3, 4, 5, 6, 7 };
        int[] intersection = findIntersection(arr1, arr2);

        System.out.print("Intersection of the arrays: ");
```

```
        for (int num : intersection) {
            System.out.print(num + " ");
        }
    }

    public static int[] findIntersection(int[] arr1, int[] arr2) {
        Set<Integer> set1 = new HashSet<>();
        Set<Integer> intersect = new HashSet<>();
        for (int num : arr1) {
            set1.add(num);
        }
        for (int num : arr2) {
            if (set1.contains(num)) {
                intersect.add(num);
            }
        }
        int[] result = new int[intersect.size()];
        int index = 0;
        for (int num : intersect) {
            result[index++] = num;
        }
        return result;
    }
}
```

1. Write a Java program to find the nth term of the Fibonacci series using recursion.


```
import java.util.Scanner;

public class FibonacciRecursion {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the value of n: ");
        int n = scanner.nextInt();
        int nthTerm = fibonacci(n);
        System.out.println("The " + n + "th term of the Fibonacci series is: " + nthTe
rm);
        scanner.close();
    }

    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Moderate Java Coding Questions

1. Write a Java function to reverse a string in place.

```
public class ReverseString {
    public String reverseString(String s) {
        char[] chars = s.toCharArray();
        int left = 0;
        int right = s.length() - 1;

        while (left < right) {
            char temp = chars[left];
            chars[left] = chars[right];
            chars[right] = temp;
            left++;
            right--;
        }

        return new String(chars);
    }
}
```

1. Given an array of integers, find the longest increasing subsequence.

```
public class LongestIncreasingSubsequence {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        int maxLength = 0;

        for (int i = 0; i < n; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }

        return maxLength;
    }
}
```


1. Write a Java program to find the intersection of two arrays.

```
import java.util.*;

public class IntersectionOfArrays {
    public int[] intersect(int[] nums1, int[] nums2) {
        Arrays.sort(nums1);
        Arrays.sort(nums2);
        List<Integer> result = new ArrayList<>();

        int i = 0;
```

```
        int j = 0;

        while (i < nums1.length && j < nums2.length) {
            if (nums1[i] == nums2[j]) {
                result.add(nums1[i]);
                i++;
                j++;
            } else if (nums1[i] < nums2[j]) {
                i++;
            } else {
                j++;
            }
        }

        int[] intersection = new int[result.size()];
        for (int k = 0; k < result.size(); k++) {
            intersection[k] = result.get(k);
        }

        return intersection;
    }
}
```

1. Implement a stack that supports push, pop, and getMin operations in O(1) time.

```
import java.util.*;

public class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

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

    public void pop() {
        int val = stack.pop();
        if (val == minStack.peek()) {
            minStack.pop();
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
```

```
        return minStack.peek();
    }
}
```

1. Given a binary tree, write a Java function to find the diameter (longest path between any two nodes).

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class DiameterOfBinaryTree {
    private int diameter;

    public int diameterOfBinaryTree(TreeNode root) {
        diameter = 0;
        depth(root);
        return diameter;
    }

    private int depth(TreeNode node) {
        if (node == null) {
            return 0;
        }

        int leftDepth = depth(node.left);
        int rightDepth = depth(node.right);

        diameter = Math.max(diameter, leftDepth + rightDepth);
        return 1 + Math.max(leftDepth, rightDepth);
    }
}
```


1. Implement a queue using two stacks in Java with enqueue and dequeue operations.

```
import java.util.Stack;

public class QueueUsingStacks {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public QueueUsingStacks() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
```

```
    }

    public void enqueue(int val) {
        stack1.push(val);
    }

    public int dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }

    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    public boolean isEmpty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```


1. Write a Java program to find the kth smallest element in a binary search tree.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class K

thSmallestElementInBST {
    public int kthSmallest(TreeNode root, int k) {
        int[] result = new int[2];
        inorder(root, k, result);
        return result[1];
    }
```

```
    private void inorder(TreeNode node, int k, int[] result) {
        if (node == null) {
            return;
        }

        inorder(node.left, k, result);

        if (++result[0] == k) {
            result[1] = node.val;
            return;
        }

        inorder(node.right, k, result);
    }
}
```

1. Given a matrix of 0s and 1s, find the largest square submatrix with all 1s.

```
public class MaxSquareSubmatrix {
    public int maximalSquare(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int m = matrix.length;
        int n = matrix[0].length;
        int[][] dp = new int[m + 1][n + 1];
        int maxSide = 0;

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (matrix[i - 1][j - 1] == '1') {
                    dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i -
 1][j - 1]) + 1;
                    maxSide = Math.max(maxSide, dp[i][j]);
                }
            }
        }

        return maxSide * maxSide;
    }
}
```


1. Write a Java program to find all valid IP addresses in a given string.

```
import java.util.*;

public class ValidIPAddresses {
    public List<String> restoreIpAddresses(String s) {
        List<String> result = new ArrayList<>();
        if (s.length() < 4 || s.length() > 12) {
```

```
            return result;
        }
        backtrack(s, 0, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(String s, int start, List<String> temp, List<String> resul
t) {
        if (temp.size() == 4 && start == s.length()) {
            result.add(String.join(".", temp));
            return;
        }

        for (int i = 1; i <= 3; i++) {
            if (start + i > s.length()) {
                break;
            }
            String part = s.substring(start, start + i);
            if (isValidPart(part)) {
                temp.add(part);
                backtrack(s, start + i, temp, result);
                temp.remove(temp.size() - 1);
            }
        }
    }

    private boolean isValidPart(String part) {
        if (part.length() > 1 && part.startsWith("0")) {
            return false;
        }
        int num = Integer.parseInt(part);
        return num >= 0 && num <= 255;
    }
}
```


1. Implement a priority queue (min-heap) in Java with insert and extractMin operations.

```
import java.util.ArrayList;
import java.util.List;

public class MinHeap {
    private List<Integer> heap;

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

    public void insert(int val) {
        heap.add(val);
        siftUp(heap.size() - 1);
    }

    public int extractMin() {
```

```
        if (isEmpty()) {
            throw new RuntimeException("Heap is empty");
        }
        int min = heap.get(0);
        int lastIdx = heap.size() - 1;
        heap.set(0, heap.get(lastIdx));
        heap.remove(lastIdx);
        siftDown(0);
        return min;
    }

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

    private void siftUp(int index) {
        while (index > 0) {
            int parentIdx = (index - 1) / 2;
            if (heap.get(parentIdx) > heap.get(index)) {
                swap(parentIdx, index);
                index = parentIdx;
            } else {
                break;
            }
        }
    }

    private void siftDown(int index) {
        int size = heap.size();
        while (index < size) {
            int leftChildIdx = 2 * index + 1;
            int rightChildIdx = 2 * index + 2;
            int minIdx = index;

            if (leftChildIdx < size && heap.get(leftChildIdx) < heap.get(minIdx)) {
                minIdx = leftChildIdx;
            }
            if (rightChildIdx < size && heap.get(rightChildIdx) < heap.get(minIdx)) {
                minIdx = rightChildIdx;
            }

            if (minIdx != index) {
                swap(index, minIdx);
                index = minIdx;
            } else {
                break;
            }
        }
    }

    private void swap(int i, int j) {
        int temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }
}
```

1. Given an array of integers, find the maximum product subarray.

```
public class MaximumProductSubarray {
    public int maxProduct(int[] nums) {
        int n = nums.length;
        if (n == 0) {
            return 0;
        }

        int maxProduct = nums[0];
        int maxEndingHere = nums[0];
        int minEndingHere = nums[0];

        for (int i = 1; i < n; i++) {
            int temp = maxEndingHere;
            maxEndingHere = Math.max(nums[i], Math.max(nums[i] * maxEndingHere, nums
[i] * minEndingHere));
            minEndingHere = Math.min(nums[i], Math.min(nums[i] * temp, nums[i] * minEn
dingHere));
            maxProduct = Math.max(maxProduct, maxEndingHere);
        }

        return maxProduct;
    }
}
```


1. Implement a function to check if a binary tree is balanced in Java.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class BalancedBinaryTree {
    public boolean isBalanced(TreeNode root) {
        return height(root) != -1;
    }

    private int height(TreeNode node) {
        if (node == null) {
            return 0;
        }

        int leftHeight = height(node.left);
        if (leftHeight == -1) {
            return -1;
        }
```

```
        int rightHeight = height(node.right);
        if (rightHeight == -1) {
            return -1;
        }

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

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

1. Write a Java program to implement a least recently used (LRU) cache.

```
import java.util.*;

class LRUCache {
    private LinkedHashMap<Integer, Integer> cache;
    private int capacity;

    public LRUCache(int capacity) {
        this

.capacity = capacity;
        cache = new LinkedHashMap<>(capacity, 0.75f, true);
    }

    public int get(int key) {
        return cache.getOrDefault(key, -1);
    }

    public void put(int key, int value) {
        cache.put(key, value);
        if (cache.size() > capacity) {
            int oldestKey = cache.entrySet().iterator().next().getKey();
            cache.remove(oldestKey);
        }
    }
}
```


1. Given a list of intervals, merge overlapping intervals in Java.

```
import java.util.*;

public class MergeIntervals {
    public int[][] merge(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return new int[0][0];
        }

        Arrays.sort(intervals, Comparator.comparingInt(a -> a[0]));
```

```
        List<int[]> mergedIntervals = new ArrayList<>();
        int[] currentInterval = intervals[0];

        for (int i = 1; i < intervals.length; i++) {
            if (currentInterval[1] >= intervals[i][0]) {
                currentInterval[1] = Math.max(currentInterval[1], intervals[i][1]);
            } else {
                mergedIntervals.add(currentInterval);
                currentInterval = intervals[i];
            }
        }

        mergedIntervals.add(currentInterval);
        return mergedIntervals.toArray(new int[mergedIntervals.size()][]);
    }
}
```

1. Write a Java program to find the longest palindromic substring in a given string.

```
public class LongestPalindromicSubstring {
    public String longestPalindrome(String s) {
        int n = s.length();
        if (n == 0) {
            return "";
        }

        boolean[][] dp = new boolean[n][n];
        int start = 0;
        int maxLength = 1;

        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        for (int i = n - 1; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                if (s.charAt(i) == s.charAt(j)) {
                    if (j - i == 1 || dp[i + 1][j - 1]) {
                        dp[i][j] = true;
                        if (j - i + 1 > maxLength) {
                            maxLength = j - i + 1;
                            start = i;
                        }
                    }
                }
            }
        }

        return s.substring(start, start + maxLength);
    }
}
```

1. Implement a function to find the longest common prefix of an array of strings.

```
public class LongestCommonPrefix {
    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }

        StringBuilder prefix = new StringBuilder(strs[0]);

        for (int i = 1; i < strs.length; i++) {
            int j = 0;
            while (j < prefix.length() && j < strs[i].length() && prefix.charAt(j) ==
 strs[i].charAt(j)) {
                j++;
            }
            prefix.setLength(j);
        }

        return prefix.toString();
    }
}
```

1. Given a binary tree, write a Java function to find the path with the maximum sum.


```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class MaximumPathSum {
    private int maxSum;

    public int maxPathSum(TreeNode root) {
        maxSum = Integer.MIN_VALUE;
        findMaxPathSum(root);
        return maxSum;
    }

    private int findMaxPathSum(TreeNode node) {
        if (node == null) {
            return 0;
        }

        int leftSum = Math.max(findMaxPathSum(node.left), 0);
        int rightSum = Math.max(findMaxPathSum(node.right), 0);
```

```
        int pathSum = node.val + leftSum + rightSum;
        maxSum = Math.max(maxSum, pathSum);

        return node.val + Math.max(leftSum, rightSum);
    }
}
```

1. Implement a function to find the first non-repeated character in a string.

```
import java.util.*;

public class FirstNonRepeatedCharacter {
    public char findFirstNonRepeatedCharacter(String s) {
        Map<Character, Integer> charCount = new LinkedHashMap<>();

        for (char c : s.toCharArray()) {
            charCount.put(c, charCount.getOrDefault(c, 0) + 1);
        }

        for (Map.Entry<Character, Integer> entry : charCount.entrySet()) {
            if (entry.getValue() == 1) {
                return entry.getKey();
            }
        }

        return '\\0'; // If no non-repeated character found
    }
}
```


1. Write a Java program to implement a stack with getMin operation in O(1) time.

```
import java.util.*;

public class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

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

    public void pop() {
        int val = stack.pop();
        if (val == minStack.peek()) {
```

```
            minStack.pop();
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

1. Given a list of words, find the longest word that can be formed by other words in the list.

```
import java.util.*;

public class LongestWordInDictionary {
    public String longestWord(String[] words) {
        Arrays.sort(words);

        Set<String> wordSet = new HashSet<>();
        String longestWord = "";

        for (String word : words) {
            if (word.length() == 1 || wordSet.contains(word.substring(0, word.length()
- 1))) {
                wordSet.add(word);
                if (word.length() > longestWord.length()) {
                    longestWord = word;
                }
            }
        }

        return longestWord;
    }
}
```


1. Implement a function to find the longest substring with at most two distinct characters.

```
public class LongestSubstringWithTwoDistinct {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        int n = s.length();
        if (n == 0) {
            return 0;
        }

        int maxLength = 0;
        int left = 0;
```

```
        int right = 0;
        Map<Character, Integer> charCount = new HashMap<>();

        while (right < n) {
            char c = s.charAt(right);
            charCount.put(c, charCount.getOrDefault(c, 0) + 1);
            while (charCount.size() > 2) {
                char leftChar = s.charAt(left);
                charCount.put(leftChar, charCount.get(leftChar) - 1);
                if (charCount.get(leftChar) == 0)

 {
                    charCount.remove(leftChar);
                }
                left++;
            }
            maxLength = Math.max(maxLength, right - left + 1);
            right++;
        }

        return maxLength;
    }
}
```

1. Given a 2D grid of characters, find all valid words from a dictionary using DFS or Trie.


```
import java.util.*;

public class WordSearchII {
    private static final int[] dx = { -1, 1, 0, 0 };
    private static final int[] dy = { 0, 0, -1, 1 };

    public List<String> findWords(char[][] board, String[] words) {
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }

        Set<String> result = new HashSet<>();
        int m = board.length;
        int n = board[0].length;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dfs(board, i, j, trie.getRoot(), result);
            }
        }

        return new ArrayList<>(result);
    }

    private void dfs(char[][] board, int x, int y, TrieNode node, Set<String> result)
 {
```

```
        char c = board[x][y];
        if (c == '#' || node.children[c - 'a'] == null) {
            return;
        }

        node = node.children[c - 'a'];
        if (node.word != null) {
            result.add(node.word);
            node.word = null; // Avoid duplicates
        }

        board[x][y] = '#'; // Mark as visited

        for (int i = 0; i < 4; i++) {
            int newX = x + dx[i];
            int newY = y + dy[i];

            if (newX >= 0 && newX < board.length && newY >= 0 && newY < board[0].lengt
h) {
                dfs(board, newX, newY, node, result);
            }
        }

        board[x][y] = c; // Restore the cell
    }
}

class TrieNode {
    TrieNode[] children;
    String word;

    TrieNode() {
        children = new TrieNode[26];
        word = null;
    }
}

class Trie {
    private TrieNode root;

    Trie() {
        root = new TrieNode();
    }

    void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.word = word;
    }

    TrieNode getRoot() {
        return root;
```

```
    }
}
```

1. Write a Java program to find the number of islands in a 2D grid of '1's and '0's.

```
public class NumberOfIslands {
    public int numIslands(char[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int count = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    dfs(grid, i, j);
                    count++;
                }
            }
        }

        return count;
    }

    private void dfs(char[][] grid, int x, int y) {
        int m = grid.length;
        int n = grid[0].length;

        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] != '1') {
            return;
        }

        grid[x][y] = '0'; // Mark as visited

        dfs(grid, x - 1, y);
        dfs(grid, x + 1, y);
        dfs(grid, x, y - 1);
        dfs(grid, x, y + 1);
    }
}
```


1. Implement an iterator for a binary search tree (BST) in Java.

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
    }
```

```
}

public class BSTIterator {
    private Stack<TreeNode> stack;

    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        pushAllLeft(root);
    }

    public int next() {
        TreeNode node = stack.pop();
        pushAllLeft(node.right);
        return node.val;
    }

    public boolean hasNext() {
        return !stack.isEmpty();
    }

    private void pushAllLeft(TreeNode node) {
        while (node != null) {
            stack.push(node);
            node = node.left;
        }
    }
}
```


1. Given a binary tree, write a Java function to find the sum of all left leaves.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class SumOfLeftLeaves {
    public int sumOfLeftLeaves(TreeNode root) {
        return sumOfLeftLeavesHelper(root, false);
    }

    private int sumOfLeftLeavesHelper(TreeNode node, boolean isLeft) {
        if (node == null) {
            return 0;
        }

        if (node.left == null && node.right == null && isLeft) {
            return node.val;
        }

        int leftSum = sumOfLeftLeavesHelper(node.left, true);
```

```
        int rightSum = sumOfLeftLeavesHelper(node.right, false);

        return leftSum + rightSum;
    }
}
```

These moderate-level coding questions cover a range of topics and challenges that can help you prepare for Java interviews. Be sure to understand the solutions and practice implementing them to enhance your problem-solving skills. Good luck with your interviews!

1. Given a binary tree, write a Java function to check if it is a binary search tree (BST).

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class ValidateBST {
    public boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean isValidBSTHelper(TreeNode node, long min, long max) {
        if (node == null) {
            return true;
        }
        if (node.val <= min || node.val >= max) {
            return false;
        }
        return isValidBSTHelper(node.left, min, node.val) && isValidBSTHelper(node.rig
ht, node.val, max);
    }
}
```


1. Implement a stack using linked lists in Java with push, pop, and peek operations.

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

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

```
}

public class LinkedStack {
    private ListNode top;

    public void push(int val) {
        ListNode newNode = new ListNode(val);
        newNode.next = top;
        top = newNode;
    }

    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        int popped = top.val;
        top = top.next;
        return popped;
    }

    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return top.val;
    }

    public boolean isEmpty() {
        return top == null;
    }
}
```


1. Write a Java program to find the kth smallest element in a binary search tree.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class KthSmallestInBST {
    private int k;
    private int result;

    public int kthSmallest(TreeNode root, int k) {
        this.k = k;
        inorderTraversal(root);
        return result;
    }

    private void inorderTraversal(TreeNode node) {
```

```
        if (node == null) {
            return;
        }
        inorderTraversal(node.left);
        k--;
        if (k == 0) {
            result = node.val;
            return;
        }
        inorderTraversal(node.right);
    }
}
```

1. Given an array of integers, find all pairs that sum up to a specific target.

```
import java.util.*;

public class TwoSum {
    public List<List<Integer>> twoSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        Map<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                result.add(Arrays.asList(nums[i], complement));
            }
            map.put(nums[i], i);
        }

        return result;
    }
}
```


1. Implement a queue using two stacks in Java with enqueue and dequeue operations.

```
import java.util.Stack;

public class QueueUsingStacks {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public QueueUsingStacks() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public void enqueue(int val) {
        stack1.push(val);
    }
```

```
    public int dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }

    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    public boolean isEmpty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```


1. Write a Java function to reverse a linked list.

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

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

public class ReverseLinkedList {
    public ListNode reverse(ListNode head) {
        ListNode prev = null;
        ListNode current = head;
        ListNode next;

        while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }

        return prev;
```

```
    }
}
```

1. Given a 2D grid of 1s and 0s, find the size of the largest island (connected 1s).

```
public class LargestIsland {
    private static final int[] dx = { -1, 1, 0, 0 };
    private static final int[] dy = { 0, 0, -1, 1 };

    public int largestIsland(int[][] grid) {
        int maxArea = 0;
        int rows = grid.length;
        int cols = grid[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == 1) {
                    maxArea = Math.max(maxArea, dfs(grid, i, j));
                }
            }
        }

        return maxArea;
    }

    private int dfs(int[][] grid, int x, int y) {
        if (x < 0 || x >= grid.length || y < 0 || y >= grid[0].length || grid[x][y] ==
0) {
            return 0;
        }

        grid[x][y] = 0;
        int area = 1;

        for (int i = 0; i < 4; i++) {
            int newX = x + dx[i];
            int newY = y + dy[i];
            area += dfs(grid, newX, newY);
        }

        return area;
    }
}
```


1. Implement a Trie (prefix tree) in Java to support insert, search, and startsWith operations.

```
class TrieNode {
    boolean isEnd;
    TrieNode[] children;

    TrieNode() {
```

```
        isEnd = false;
        children = new TrieNode[26];
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.isEnd = true;
    }

    public boolean search(String word) {
        TrieNode node = findNode(word);
        return node != null && node

.isEnd;
    }

    public boolean startsWith(String prefix) {
        return findNode(prefix) != null;
    }

    private TrieNode findNode(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return null;
            }
            node = node.children[index];
        }
        return node;
    }
}
```


1. Write a Java program to find the longest common subsequence (LCS) of two strings.

```
public class LongestCommonSubsequence {
    public String longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
```

```
        int n = text2.length();
        int[][] dp = new int[m + 1][n + 1];

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        StringBuilder lcs = new StringBuilder();
        int i = m, j = n;

        while (i > 0 && j > 0) {
            if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                lcs.insert(0, text1.charAt(i - 1));
                i--;
                j--;
            } else if (dp[i - 1][j] > dp[i][j - 1]) {
                i--;
            } else {
                j--;
            }
        }

        return lcs.toString();
    }
}
```


1. Given a matrix of 0s and 1s, find the size of the largest square submatrix with all 1s.

```
public class MaxSquareSubmatrix {
    public int maximalSquare(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int m = matrix.length;
        int n = matrix[0].length;
        int[][] dp = new int[m + 1][n + 1];
        int maxSide = 0;

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (matrix[i - 1][j - 1] == '1') {
                    dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i -
 1][j - 1]) + 1;
                    maxSide = Math.max(maxSide, dp[i][j]);
                }
            }
```

```
        }

        return maxSide * maxSide;
    }
}
```

1. Implement a hash table (dictionary) in Java with put, get, and remove operations.

```
class Entry<K, V> {
    K key;
    V value;
    Entry<K, V> next;

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

public class HashTable<K, V> {
    private static final int INITIAL_CAPACITY = 16;
    private Entry<K, V>[] table;
    private int size;

    public HashTable() {
        table = new Entry[INITIAL_CAPACITY];
        size = 0;
    }

    public void put(K key, V value) {
        int index = getIndex(key);
        Entry<K, V> entry = new Entry<>(key, value);

        if (table[index] == null) {
            table[index] = entry;
            size++;
        } else {
            Entry<K, V> current = table[index];
            while (current.next != null) {
                if (current.key.equals(key)) {
                    current.value = value;
                    return;
                }
                current = current.next;
            }
            current.next = entry;
            size++;
        }

        if (size >= table.length * 0.75) {
            resize();
        }
    }
```

```
public V get(K key) {
    int index = getIndex(key);
    Entry<K, V> current = table[index];
    while (current != null) {
        if (current.key.equals(key)) {
            return current.value;
        }
        current = current.next;
    }
    return null;
}

public void remove(K key) {
    int index = getIndex(key);
    Entry<K, V> prev = null;
    Entry<K, V> current = table[index];

    while (current != null) {
        if (current.key.equals(key)) {
            if (prev == null) {
                table[index] = current.next;
            } else {
                prev.next = current.next;
            }
            size--;
            return;
        }
        prev = current;
        current = current.next;
    }
}

public boolean containsKey(K key) {
    int index = getIndex(key);
    Entry<K, V> current = table[index];
    while (current != null) {
        if (current.key.equals(key)) {
            return true;
        }
        current = current.next;
    }
    return false;
}

public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

private int getIndex(K key) {
    int hashCode = key.hashCode();
    return hashCode % table.length;
}
```

```
    private void resize() {
        int newCapacity = table.length * 2;
        Entry<K, V>[] newTable = new Entry[newCapacity];

        for (Entry<K, V> entry : table) {
            while (entry != null) {
                int newIndex = entry.key.hashCode() % newCapacity;
                Entry<K, V> newEntry = new Entry<>(entry.key, entry.value);
                newEntry.next = newTable[newIndex];
                newTable[newIndex] = newEntry;
                entry = entry.next;
            }
        }

        table = newTable;
    }
}
```

1. Write a Java function to find the first non-repeated character in a string.

```
import java.util.HashMap;
import java.util.Map;

public class FirstNonRepeatedCharacter {
    public char findFirstNonRepeatedCharacter(String str) {
        Map<Character, Integer> charCountMap = new HashMap<>();

        for (char c : str.toCharArray()) {
            charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);
        }

        for (char c : str.toCharArray()) {
            if (charCountMap.get(c) == 1) {
                return c;
            }
        }

        return '\\0'; // Return null character if no non-repeated character found
    }
}
```


1. Given a binary tree, write a Java function to find the maximum path sum between any two nodes.

```
class TreeNode {
    int val

;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
```

```
        this.val = val;
    }
}

public class MaxPathSum {
    private int maxSum = Integer.MIN_VALUE;

    public int maxPathSum(TreeNode root) {
        findMaxPathSum(root);
        return maxSum;
    }

    private int findMaxPathSum(TreeNode node) {
        if (node == null) {
            return 0;
        }

        int leftSum = Math.max(findMaxPathSum(node.left), 0);
        int rightSum = Math.max(findMaxPathSum(node.right), 0);

        int pathSum = node.val + leftSum + rightSum;
        maxSum = Math.max(maxSum, pathSum);

        return node.val + Math.max(leftSum, rightSum);
    }
}
```

1. Implement a priority queue (min-heap) in Java with insert and extractMin operations.


```
import java.util.ArrayList;
import java.util.List;

public class MinHeap {
    private List<Integer> heap;

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

    public void insert(int val) {
        heap.add(val);
        siftUp(heap.size() - 1);
    }

    public int extractMin() {
        if (isEmpty()) {
            throw new RuntimeException("Heap is empty");
        }
        int min = heap.get(0);
        int lastIdx = heap.size() - 1;
        heap.set(0, heap.get(lastIdx));
        heap.remove(lastIdx);
        siftDown(0);
```

```
        return min;
    }

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

    private void siftUp(int index) {
        while (index > 0) {
            int parentIdx = (index - 1) / 2;
            if (heap.get(parentIdx) > heap.get(index)) {
                swap(parentIdx, index);
                index = parentIdx;
            } else {
                break;
            }
        }
    }

    private void siftDown(int index) {
        int size = heap.size();
        while (index < size) {
            int leftChildIdx = 2 * index + 1;
            int rightChildIdx = 2 * index + 2;
            int minIdx = index;

            if (leftChildIdx < size && heap.get(leftChildIdx) < heap.get(minIdx)) {
                minIdx = leftChildIdx;
            }
            if (rightChildIdx < size && heap.get(rightChildIdx) < heap.get(minIdx)) {
                minIdx = rightChildIdx;
            }

            if (minIdx != index) {
                swap(index, minIdx);
                index = minIdx;
            } else {
                break;
            }
        }
    }

    private void swap(int i, int j) {
        int temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }
}
```


1. Write a Java program to merge k sorted arrays into a single sorted array.

```
import java.util.PriorityQueue;

public class MergeKSortedArrays {
    public int[] mergeKSortedArrays(int[][] arrays) {
```

```
        if (arrays == null || arrays.length == 0) {
            return new int[0];
        }

        PriorityQueue<ArrayElement> pq = new PriorityQueue<>((a, b) -> a.val - b.val);
        int totalSize = 0;

        for (int i = 0; i < arrays.length; i++) {
            if (arrays[i].length > 0) {
                pq.offer(new ArrayElement(arrays[i][0], i, 0));
                totalSize += arrays[i].length;
            }
        }

        int[] mergedArray = new int[totalSize];
        int index = 0;

        while (!pq.isEmpty()) {
            ArrayElement element = pq.poll();
            mergedArray[index++] = element.val;

            int arrayIndex = element.arrayIndex;
            int nextIndex = element.nextIndex + 1;
            if (nextIndex < arrays[arrayIndex].length) {
                pq.offer(new ArrayElement(arrays[arrayIndex][nextIndex], arrayIndex, n
extIndex));
            }
        }

        return mergedArray;
    }

    class ArrayElement {
        int val;
        int arrayIndex;
        int nextIndex;

        ArrayElement(int val, int arrayIndex, int nextIndex) {
            this.val = val;
            this.arrayIndex = arrayIndex;
            this.nextIndex = nextIndex;
        }
    }
}
```


1. Given an array of integers, find the length of the longest increasing subarray.

```
public class LongestIncreasingSubarray {
    public int findLengthOfLIS(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        dp[0] = 1;
        int maxLength = 1;

        for (int i = 1; i < n; i++) {
```

```
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }

        return maxLength;
    }
}
```

1. Implement a graph in Java with BFS (Breadth-First Search) and DFS (Depth-First Search) traversal algorithms.

```
import java.util.*;

class Graph {
    private int V;
    private List<Integer>[] adjList;

    public Graph(int V) {
        this.V = V;
        adjList = new ArrayList[V];
        for (int i = 0; i < V; i++) {
            adjList[i] = new ArrayList<>();
        }
    }

    public void addEdge(int u, int v) {
        adjList[u].add(v);
        adjList[v].add(u); // For undirected graph
    }

    public void bfsTraversal(int start) {
        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);
        visited[start] = true;

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

            for (int neighbor : adjList[node]) {
                if (!visited[neighbor]) {
                    queue.add(neighbor);
                    visited[neighbor] = true;
                }
            }
        }
    }
```

```
    public void dfsTraversal(int start) {
        boolean[] visited = new boolean[V];
        dfsHelper(start, visited);
    }

    private void dfsHelper(int node, boolean[] visited) {
        visited[node] = true;
        System.out.print(node + " ");

        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                dfsHelper(neighbor, visited);
            }
        }
    }
}
```

1. Write a Java program to find the median of two sorted arrays of different sizes.

```
public class MedianOfTwoSortedArrays {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length;
        int n = nums2.length;

        if (m > n) {
            return find

MedianSortedArrays(nums2, nums1);
        }

        int left = 0;
        int right = m;
        int total = (m + n + 1) / 2;

        while (left < right) {
            int partition1 = left + (right - left) / 2;
            int partition2 = total - partition1;

            int maxLeft1 = (partition1 == 0) ? Integer.MIN_VALUE : nums1[partition1 -
 1];
            int minRight1 = (partition1 == m) ? Integer.MAX_VALUE : nums1[partition1];

            int maxLeft2 = (partition2 == 0) ? Integer.MIN_VALUE : nums2[partition2 -
 1];
            int minRight2 = (partition2 == n) ? Integer.MAX_VALUE : nums2[partition2];

            if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
                if ((m + n) % 2 == 0) {
                    return (Math.max(maxLeft1, maxLeft2) + Math.min(minRight1, minRigh
t2)) / 2.0;
                } else {
                    return Math.max(maxLeft1, maxLeft2);
                }
            } else if (maxLeft1 > minRight2) {
                right = partition1 - 1;
```

```
            } else {
                left = partition1 + 1;
            }
        }

        throw new IllegalArgumentException("Input arrays are not sorted.");
    }
}
```

1. Given a 2D grid of characters, find all valid words from a dictionary using DFS or Trie.

```
import java.util.*;

public class WordSearchII {
    private static final int[] dx = { -1, 1, 0, 0 };
    private static final int[] dy = { 0, 0, -1, 1 };

    public List<String> findWords(char[][] board, String[] words) {
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }

        Set<String> result = new HashSet<>();
        int m = board.length;
        int n = board[0].length;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dfs(board, i, j, trie.getRoot(), result);
            }
        }

        return new ArrayList<>(result);
    }

    private void dfs(char[][] board, int x, int y, TrieNode node, Set<String> result)
 {
        char c = board[x][y];
        if (c == '#' || node.children[c - 'a'] == null) {
            return;
        }

        node = node.children[c - 'a'];
        if (node.word != null) {
            result.add(node.word);
            node.word = null; // Avoid duplicates
        }

        board[x][y] = '#'; // Mark as visited

        for (int i = 0; i < 4; i++) {
            int newX = x + dx[i];
```

```
            int newY = y + dy[i];

            if (newX >= 0 && newX < board.length && newY >= 0 && newY < board[0].lengt
h) {
                dfs(board, newX, newY, node, result);
            }
        }

        board[x][y] = c; // Restore the cell
    }
}

class TrieNode {
    TrieNode[] children;
    String word;

    TrieNode() {
        children = new TrieNode[26];
        word = null;
    }
}

class Trie {
    private TrieNode root;

    Trie() {
        root = new TrieNode();
    }

    void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.word = word;
    }

    TrieNode getRoot() {
        return root;
    }
}
```


1. Implement a circular linked list in Java with insert, delete, and traverse operations.

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

    ListNode(int val) {
```

```
        this.val = val;
        this.next = null;
    }
}

public class CircularLinkedList {
    private ListNode head;

    public CircularLinkedList() {
        head = null;
    }

    public void insert(int val) {
        ListNode newNode = new ListNode(val);
        if (head == null) {
            head = newNode;
            head.next = head;
        } else {
            ListNode tail = head;
            while (tail.next != head) {
                tail = tail.next;
            }
            tail.next = newNode;
            newNode.next = head;
        }
    }

    public void delete(int val) {
        if (head == null) {
            return;
        }
        ListNode curr = head;
        ListNode prev = null;
        while (curr.next != head) {
            if (curr.val == val) {
                if (prev != null) {
                    prev.next = curr.next;
                } else {
                    ListNode tail = head;
                    while (tail.next != head) {
                        tail = tail.next;
                    }
                    head = head.next;
                    tail.next = head;
                }
                return;
            }
            prev = curr;
            curr = curr.next;
        }
        if (curr.val == val) {
            if (prev != null) {
                prev.next = head;
            } else {
                head = null;
            }
        }
    }
```

```
    public void traverse() {
        if (head == null) {
            return;
        }
        ListNode curr = head;
        do {
            System.out.print(curr.val + " ");
            curr = curr.next;
        } while (curr != head);
    }
}
```

1. Write a Java program to find the maximum sum subarray using the Kadane's algorithm.

```
public class MaximumSubarraySum {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];

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

        return maxSum;
    }
}
```


1. Given a list of intervals, merge overlapping intervals in Java.

```
import java.util.*;

public class MergeIntervals {
    public int[][] merge(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return new int[0][0];
        }

        Arrays.sort(intervals, Comparator.comparingInt(a -> a[0]));

        List<int[]> mergedIntervals = new ArrayList<>();
        int[] currentInterval = intervals[0];

        for (int i = 1; i < intervals.length; i++) {
            if (currentInterval[1] >= intervals[i][0]) {
                currentInterval[1] = Math.max(currentInterval[1], intervals[i][1]);
            } else {
                mergedIntervals.add(currentInterval);
                currentInterval = intervals[i];
            }
```

```
        }

        mergedIntervals.add(currentInterval);
        return

 mergedIntervals.toArray(new int[mergedIntervals.size()][]);
    }
}
```

1. Implement a binary search in Java to find the index of a given element in a sorted array.

```
public class BinarySearch {
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return -1;
    }
}
```


1. Write a Java function to serialize and deserialize a binary tree (convert to/from a string representation).

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class SerializeDeserializeBinaryTree {
    private static final String NULL_NODE = "null";
    private static final String DELIMITER = ",";

    // Serialize a binary tree to a string
```

```
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append(NULL_NODE).append(DELIMITER);
        } else {
            sb.append(node.val).append(DELIMITER);
            buildString(node.left, sb);
            buildString(node.right, sb);
        }
    }

    // Deserialize a string to a binary tree
    public TreeNode deserialize(String data) {
        Queue<String> queue = new LinkedList<>(Arrays.asList(data.split(DELIMITER)));
        return buildTree(queue);
    }

    private TreeNode buildTree(Queue<String> queue) {
        String val = queue.poll();
        if (val.equals(NULL_NODE)) {
            return null;
        } else {
            TreeNode node = new TreeNode(Integer.parseInt(val));
            node.left = buildTree(queue);
            node.right = buildTree(queue);
            return node;
        }
    }
}
```


1. Given a directed graph, check if it contains a cycle using DFS or BFS in Java.

```
import java.util.*;

public class CycleInDirectedGraph {
    public boolean hasCycle(int numCourses, int[][] prerequisites) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }

        for (int[] prerequisite : prerequisites) {
            graph.get(prerequisite[0]).add(prerequisite[1]);
        }

        int[] visited = new int[numCourses];

        for (int i = 0; i < numCourses; i++) {
            if (hasCycleDFS(i, graph, visited)) {
                return true;
```

```
            }
        }

        return false;
    }

    private boolean hasCycleDFS(int course, List<List<Integer>> graph, int[] visited)
 {
        if (visited[course] == 1) {
            return true;
        }

        if (visited[course] == -1) {
            return false;
        }

        visited[course] = 1;

        for (int prerequisite : graph.get(course)) {
            if (hasCycleDFS(prerequisite, graph, visited)) {
                return true;
            }
        }

        visited[course] = -1;
        return false;
    }
}
```


Hard Level Coding Questions

1. Implement a function to find the longest increasing subsequence in a given array.

```
public class LongestIncreasingSubsequence {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        int maxLength = 0;

        for (int i = 0; i < n; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }

        return maxLength;
    }
}
```

1. Given a 2D grid of characters and a word, check if the word exists in the grid.

```
public class WordSearch {
    public boolean exist(char[][] board, String word) {
        int m = board.length;
        int n = board[0].length;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (dfs(board, i, j, word, 0)) {
                    return true;
                }
            }
        }

        return false;
    }

    private boolean dfs(char[][] board, int x, int y, String word, int index) {
        if (index == word.length()) {
            return true;
        }

        if (x < 0 || x >= board.length || y < 0 || y >= board[0].length || board[x][y]
!= word.charAt(index)) {
            return false;
        }

        char temp = board[x][y];
        board[x][y] = '#'; // Mark as visited

        boolean found = dfs(board, x - 1, y, word, index + 1) ||
                        dfs(board, x + 1, y, word, index + 1) ||
                        dfs(board, x, y - 1, word, index + 1) ||
                        dfs(board, x, y + 1, word, index + 1);

        board[x][y] = temp; // Restore the cell
        return found;
    }
}
```


1. Write a Java program to find the kth largest element in an unsorted array.

```
import java.util.*;

public class KthLargestElement {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
```

```
                minHeap.poll();
            }
        }

        return minHeap.peek();
    }
}
```

1. Given an array of integers, find the maximum subarray sum (Kadane's algorithm).

```
public class MaximumSubarraySum {
    public int maxSubArray(int[] nums) {
        int maxSum = Integer.MIN_VALUE;
        int currentSum = 0;

        for (int num : nums) {
            currentSum = Math.max(num, currentSum + num);
            maxSum = Math.max(maxSum, currentSum);
        }

        return maxSum;
    }
}
```


1. Implement a function to find the minimum window substring in a given string.

```
import java.util.*;

public class MinimumWindowSubstring {
    public String minWindow(String s, String t) {
        if (s.length() == 0 || t.length() == 0) {
            return "";
        }

        Map<Character, Integer> targetCount = new HashMap<>();
        for (char c : t.toCharArray()) {
            targetCount.put(c, targetCount.getOrDefault(c, 0) + 1);
        }

        int left = 0;
        int minLen = Integer.MAX_VALUE;
        int minStart = 0;
        int count = t.length();

        for (int right = 0; right < s.length(); right++) {
            char rightChar = s.charAt(right);
            if (targetCount.containsKey(rightChar)) {
                targetCount.put(rightChar, targetCount.get(rightChar) - 1);
                if (targetCount.get(rightChar) >= 0) {
                    count--;
                }
```

```
            }

            while (count == 0) {
                if (right - left + 1 < minLen) {
                    minLen = right - left + 1;
                    minStart = left;
                }

                char leftChar = s.charAt(left);
                if (targetCount.containsKey(leftChar)) {
                    targetCount.put(leftChar, targetCount.get(leftChar) + 1);
                    if (targetCount.get(leftChar) > 0) {
                        count++;
                    }
                }

                left++;
            }
        }

        return minLen == Integer.MAX_VALUE ? "" : s.substring(minStart, minStart + min
Len);
    }
}
```

1. Write a Java program to implement a regular expression matcher.


```
public class RegularExpressionMatcher {
    public boolean isMatch(String s, String p) {
        if (p.isEmpty()) {
            return s.isEmpty();
        }

        boolean firstMatch = !s.isEmpty() && (s.charAt(0) == p.charAt(0) || p.charAt
(0) == '.');

        if (p.length() >= 2 && p.charAt(1) == '*') {
            return (isMatch(s, p.substring(2)) || (firstMatch && isMatch(s.substring
(1), p)));
        } else {
            return firstMatch && isMatch(s.substring(1), p.substring(1));
        }
    }
}
```

1. Implement a function to find the median of two sorted arrays in Java.

```
public class MedianOfTwoSortedArrays {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length;
        int n = nums2.length;
        if (m > n) {
```

```
            return findMedianSortedArrays(nums2, nums1);
        }

        int imin = 0;
        int imax = m;
        int halfLen = (m + n + 1) / 2;

        while (imin <= imax) {
            int i = (imin + imax) / 2;
            int j = halfLen - i;

            if (i < m && nums2[j - 1] > nums1[i]) {
                imin = i + 1;
            } else if (i > 0 && nums1[i - 1] > nums2[j]) {
                imax = i - 1;
            } else {
                int maxOfLeft;
                if (i == 0) {
                    maxOfLeft = nums2[j - 1];
                } else if (j == 0) {
                    maxOfLeft = nums1[i - 1];
                } else {
                    maxOfLeft = Math.max(nums1[i - 1], nums2[j - 1]);
                }

                if ((m + n) % 2 == 1) {
                    return maxOfLeft;
                }

                int minOfRight;
                if (i == m) {


 minOfRight = nums2[j];
                } else if (j == n) {
                    minOfRight = nums1[i];
                } else {
                    minOfRight = Math.min(nums1[i], nums2[j]);
                }

                return (maxOfLeft + minOfRight) / 2.0;
            }
        }

        return 0.0;
    }
}
```


1. Given a 2D grid of 0s and 1s, find the largest rectangle containing only 1s.

```
public class MaximalRectangle {
    public int maximalRectangle(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
```

```
        int m = matrix.length;
        int n = matrix[0].length;
        int[] heights = new int[n];
        int maxArea = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == '1') {
                    heights[j]++;
                } else {
                    heights[j] = 0;
                }
            }
            maxArea = Math.max(maxArea, largestRectangleArea(heights));
        }

        return maxArea;
    }

    private int largestRectangleArea(int[] heights) {
        int n = heights.length;
        int maxArea = 0;
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i <= n; i++) {
            int h = (i == n) ? 0 : heights[i];
            while (!stack.isEmpty() && h < heights[stack.peek()]) {
                int height = heights[stack.pop()];
                int width = stack.isEmpty() ? i : i - stack.peek() - 1;
                maxArea = Math.max(maxArea, height * width);
            }
            stack.push(i);
        }

        return maxArea;
    }
}
```


1. Given a binary tree, write a Java function to find the largest BST subtree.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

class BSTInfo {
    int min;
    int max;
    int size;
```

```
    boolean isBST;

    BSTInfo(int min, int max, int size, boolean isBST) {
        this.min = min;
        this.max = max;
        this.size = size;
        this.isBST = isBST;
    }
}

public class LargestBSTSubtree {
    public int largestBSTSubtree(TreeNode root) {
        return largestBSTSubtreeHelper(root).size;
    }

    private BSTInfo largestBSTSubtreeHelper(TreeNode node) {
        if (node == null) {
            return new BSTInfo(Integer.MAX_VALUE, Integer.MIN_VALUE, 0, true);
        }

        BSTInfo leftInfo = largestBSTSubtreeHelper(node.left);
        BSTInfo rightInfo = largestBSTSubtreeHelper(node.right);

        if (leftInfo.isBST && rightInfo.isBST && node.val > leftInfo.max && node.val <
rightInfo.min) {
            int size = leftInfo.size + rightInfo.size + 1;
            int min = Math.min(node.val, leftInfo.min);
            int max = Math.max(node.val, rightInfo.max);
            return new BSTInfo(min, max, size, true);
        } else {
            return new BSTInfo(0, 0, Math.max(leftInfo.size, rightInfo.size), false);
        }
    }
}
```


1. Implement a function to serialize and deserialize a binary tree in Java.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class SerializeDeserializeBinaryTree {
    private static final String NULL_NODE = "null";
    private static final String DELIMITER = ",";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
```

```
        return sb.toString();
    }

    private void serializeHelper(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append(NULL_NODE).append(DELIMITER);
            return;
        }

        sb.append(node.val).append(DELIMITER);
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Queue<String> nodes = new LinkedList<>(Arrays.asList(data.split(DELIMITER)));
        return deserializeHelper(nodes);
    }

    private TreeNode deserializeHelper(Queue<String> nodes) {
        String val = nodes.poll();
        if (val.equals(NULL_NODE)) {
            return null;
        }

        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.left = deserializeHelper(nodes);
        node.right = deserializeHelper(nodes);
        return node;
    }
}
```


1. Write a Java program to implement a binary search tree (BST) with insert, delete, and search operations.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class BinarySearchTree {
    private TreeNode root;

    public void insert(int val) {
        root = insertHelper(root, val);
    }

    private TreeNode insertHelper(TreeNode node, int val) {
```

```
        if (node == null) {
            return new TreeNode(val);
        }

        if (val < node.val) {
            node.left = insertHelper(node.left, val);
        } else if (val > node.val) {
            node.right = insertHelper(node.right, val);
        }

        return node;
    }

    public boolean search(int val) {
        return searchHelper(root, val);
    }

    private boolean searchHelper(TreeNode node, int val) {
        if (node == null) {
            return false;
        }

        if (val == node.val) {
            return true;
        } else if (val < node.val) {
            return searchHelper(node.left, val);
        } else {
            return searchHelper(node.right, val);
        }
    }

    public void delete(int val) {
        root = deleteHelper(root, val);
    }

    private TreeNode deleteHelper(TreeNode node, int val) {
        if (node == null) {
            return null;
        }

        if (val < node.val) {
            node.left = deleteHelper(node.left, val);
        } else if (val > node.val) {
            node.right = deleteHelper(node.right, val);
        } else {
            if (node.left == null) {
                return node.right;
            } else if (node.right == null) {
                return node.left;
            }

            node.val = findMinValue(node.right);
            node.right = deleteHelper(node.right, node.val);
        }

        return node;
    }
```

```
    private int findMinValue(TreeNode node) {
        while (node.left != null) {
            node = node.left;
        }
        return node.val;
    }
}
```

1. Implement a function to find the longest palindromic substring in a given string (Manacher's algorithm).

```
public class LongestPalindromicSubstring {


 public String longestPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }

        char[] t = preProcess(s);
        int n = t.length;
        int[] p = new int[n];
        int center = 0;
        int right = 0;

        for (int i = 1; i < n - 1; i++) {
            int mirror = 2 * center - i;

            if (right > i) {
                p[i] = Math.min(right - i, p[mirror]);
            }

            while (t[i + p[i] + 1] == t[i - p[i] - 1]) {
                p[i]++;
            }

            if (i + p[i] > right) {
                center = i;
                right = i + p[i];
            }
        }

        int maxLen = 0;
        int centerIndex = 0;

        for (int i = 1; i < n - 1; i++) {
            if (p[i] > maxLen) {
                maxLen = p[i];
                centerIndex = i;
            }
        }

        int start = (centerIndex - maxLen) / 2;
        return s.substring(start, start + maxLen);
```

```
    }

    private char[] preProcess(String s) {
        int n = s.length();
        char[] t = new char[2 * n + 3];
        t[0] = '^';
        for (int i = 0; i < n; i++) {
            t[2 * i + 1] = '#';
            t[2 * i + 2] = s.charAt(i);
        }
        t[2 * n + 1] = '#';
        t[2 * n + 2] = '$';
        return t;
    }
}
```

1. Given a matrix of m x n elements (m rows, n columns), write a Java function to find the longest increasing path.

```
public class LongestIncreasingPathInMatrix {
    private static final int[] dx = { -1, 1, 0, 0 };
    private static final int[] dy = { 0, 0, -1, 1 };

    public int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int m = matrix.length;
        int n = matrix[0].length;
        int[][] cache = new int[m][n];
        int maxPathLength = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                maxPathLength = Math.max(maxPathLength, dfs(matrix, i, j, cache));
            }
        }

        return maxPathLength;
    }

    private int dfs(int[][] matrix, int x, int y, int[][] cache) {
        if (cache[x][y] != 0) {
            return cache[x][y];
        }

        int m = matrix.length;
        int n = matrix[0].length;
        int maxPath = 1;

        for (int i = 0; i < 4; i++) {
            int newX = x + dx[i];
            int newY = y + dy[i];
```

```
            if (newX >= 0 && newX < m && newY >= 0 && newY < n && matrix[newX][newY] >
matrix[x][y]) {
                maxPath = Math.max(maxPath, 1 + dfs(matrix, newX, newY, cache));
            }
        }

        cache[x][y] = maxPath;
        return maxPath;
    }
}
```

1. Implement an LRU (Least Recently Used) cache using a doubly-linked list and a hashmap.

```
import java.util.*;

class Node {
    int key;
    int value;
    Node prev;
    Node next;

    Node(int key, int value) {
        this.key = key;
        this.value = value;
    }
}

public class LRUCache {
    private final int capacity;
    private final Map<Integer, Node> cache;
    private Node head;
    private Node tail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new HashMap<>();
        this.head = new Node(0, 0);
        this.tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;
    }

    private void addToFront(Node node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
```

```
    }

    private void moveToHead(Node node) {
        removeNode(node);
        addToFront(node);
    }

    public int get(int key) {
        if (cache.containsKey(key)) {
            Node node = cache.get(key);
            moveToHead(node);
            return node.value;
        }
        return -1;
    }

    public void put(int key, int value) {
        if (cache.containsKey(key)) {
            Node node = cache.get(key);
            node.value = value;
            moveToHead(node);
        } else {
            if (cache.size() >= capacity) {
                Node evictNode = tail.prev;
                removeNode(evictNode);
                cache.remove(evictNode.key);
            }
            Node newNode = new Node(key, value);
            addToFront(newNode);
            cache.put(key, newNode);
        }
    }
}
```


1. Given a list of non-negative integers, write a Java function to find the maximum amount of water that can be trapped.

```
public class TrappingRainWater {
    public int trap(int[] height) {
        int n = height.length;
        int[] leftMax = new int[n];
        int[] rightMax = new int[n];
        int totalWater = 0;

        int maxLeft = 0;
        for (int i = 0; i < n; i++) {
            leftMax[i] = maxLeft;
            maxLeft = Math.max(maxLeft, height[i]);
        }

        int maxRight = 0;
        for (int i = n - 1; i >= 0; i--) {
            rightMax[i] = maxRight;
            maxRight = Math.max(maxRight, height[i]);
```

```
            int minHeight = Math.min(leftMax[i], rightMax[i]);
            if (minHeight > height[i]) {
                totalWater += minHeight - height[i];
            }
        }

        return totalWater;
    }
}
```

1. Write a Java program to implement a trie (prefix tree) with insert, search, and startsWith operations.

```
class TrieNode {
    boolean isWord;
    TrieNode[] children;

    TrieNode() {
        isWord = false;
        children = new TrieNode[26];
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {


 int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.isWord = true;
    }

    public boolean search(String word) {
        TrieNode node = findNode(word);
        return node != null && node.isWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = findNode(prefix);
        return node != null;
    }
```

```
    private TrieNode findNode(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return null;
            }
            node = node.children[index];
        }
        return node;
    }
}
```

1. Implement a function to find the longest common prefix in an array of strings.

```
public class LongestCommonPrefix {
    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }

        String prefix = strs[0];

        for (int i = 1; i < strs.length; i++) {
            while (!strs[i].startsWith(prefix)) {
                prefix = prefix.substring(0, prefix.length() - 1);
            }
        }

        return prefix;
    }
}
```


1. Given a non-empty binary tree, write a Java function to find the maximum average value of a subtree.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

class SubtreeInfo {
    int sum;
    int count;

    SubtreeInfo(int sum, int count) {
        this.sum = sum;
```

```
        this.count = count;
    }
}

public class MaximumAverageSubtree {
    private double maxAverage;

    public double maximumAverageSubtree(TreeNode root) {
        maxAverage = 0;
        findMaxAverage(root);
        return maxAverage;
    }

    private SubtreeInfo findMaxAverage(TreeNode node) {
        if (node == null) {
            return new SubtreeInfo(0, 0);
        }

        SubtreeInfo leftInfo = findMaxAverage(node.left);
        SubtreeInfo rightInfo = findMaxAverage(node.right);

        int sum = leftInfo.sum + rightInfo.sum + node.val;
        int count = leftInfo.count + rightInfo.count + 1;

        double average = (double) sum / count;
        maxAverage = Math.max(maxAverage, average);

        return new SubtreeInfo(sum, count);
    }
}
```


1. Write a Java program to find the maximum distance between two nodes in a binary tree.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class MaximumDistanceBetweenNodes {
    private int maxDistance;

    public int maxDistance(TreeNode root) {
        maxDistance = 0;
        findMaxDistance(root);
        return maxDistance;
    }

    private int[] findMaxDistance(TreeNode node) {
```

```
        if (node == null) {
            return new int[] { -1, -1 };
        }

        int[] left = findMaxDistance(node.left);
        int[] right = findMaxDistance(node.right);

        int heightLeft = left[0] + 1;
        int heightRight = right[0] + 1;
        int distanceLeft = left[1];
        int distanceRight = right[1];

        maxDistance = Math.max(maxDistance, distanceLeft);
        maxDistance = Math.max(maxDistance, distanceRight);
        maxDistance = Math.max(maxDistance, heightLeft + heightRight);

        return new int[] { Math.max(heightLeft, heightRight), Math.max(heightLeft + he
ightRight, Math.max(distanceLeft, distanceRight)) };
    }
}
```

1. Implement a function to find the number of possible paths in a 2D grid from the top-left corner to the bottom-right corner.

```
public class UniquePathsInGrid {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];

        for (int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }

        for (int i = 0; i < n; i++) {
            dp[0][i] = 1;
        }

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

        return dp[m - 1][n - 1];
    }
}
```


1. Write a Java program to find the longest palindromic substring with a dynamic programming approach.

```
public class LongestPalindromicSubstringDP {
    public String longestPalindrome(String s) {
```

```
        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        String longestPalindrome = "";

        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                dp[i][j] = (s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j -
 1]));

                if (dp[i][j] && (longestPalindrome.isEmpty() || j - i + 1 > longestPal
indrome.length())) {
                    longestPalindrome = s.substring(i, j + 1);
                }
            }
        }

        return longestPalindrome;
    }
}
```

1. Given a string s, write a Java program to find the length of the longest substring without repeating characters.

```
import java.util.*;

public class LongestSubstringWithoutRepeatingCharacters {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        Map<Character, Integer> charIndexMap = new HashMap<>();
        int maxLength = 0;
        int left = 0;

        for (int right = 0; right < n; right++) {
            char rightChar = s.charAt(right);
            if (charIndexMap.containsKey(rightChar)) {
                left = Math.max(left, charIndexMap.get(rightChar) + 1);
            }
            charIndexMap.put(rightChar, right);
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }
}
```


1. Given a string containing only digits, write a Java program to restore it by returning all possible valid IP address combinations.

```
import java.util.*;

public class RestoreIPAddresses {
```

```
    public List<String> restoreIpAddresses(String s) {
        List<String> result = new ArrayList<>();
        int n = s.length();

        for (int i = 1; i <= 3 && i <= n - 3; i++) {
            for (int j = i + 1; j <= i + 3 && j <= n - 2; j++) {
                for (int k = j + 1; k <= j +

3 && k <= n - 1; k++) {
                    String part1 = s.substring(0, i);
                    String part2 = s.substring(i, j);
                    String part3 = s.substring(j, k);
                    String part4 = s.substring(k);

                    if (isValid(part1) && isValid(part2) && isValid(part3) && isValid
(part4)) {
                        result.add(part1 + "." + part2 + "." + part3 + "." + part4);
                    }
                }
            }
        }

        return result;
    }

    private boolean isValid(String part) {
        if (part.length() > 1 && part.startsWith("0")) {
            return false;
        }
        int num = Integer.parseInt(part);
        return num >= 0 && num <= 255;
    }
}
```


1. Write a Java program to find the longest increasing subsequence with a binary search approach.

```
import java.util.*;

public class LongestIncreasingSubsequenceBinarySearch {
    public int lengthOfLIS(int[] nums) {
        List<Integer> lis = new ArrayList<>();

        for (int num : nums) {
            int index = Collections.binarySearch(lis, num);

            if (index < 0) {
                index = -(index + 1);
            }

            if (index == lis.size()) {
                lis.add(num);
            } else {
                lis.set(index, num);
```

```
            }
        }

        return lis.size();
    }
}
```

1. Given a list of words, write a Java program to find all word squares.

```
import java.util.*;

public class WordSquares {
    public List<List<String>> wordSquares(String[] words) {
        List<List<String>> result = new ArrayList<>();
        Map<String, List<String>> prefixMap = new HashMap<>();

        for (String word : words) {
            for (int i = 1; i <= word.length(); i++) {
                String prefix = word.substring(0, i);
                prefixMap.putIfAbsent(prefix, new ArrayList<>());
                prefixMap.get(prefix).add(word);
            }
        }

        for (String word : words) {
            List<String> wordSquare = new ArrayList<>();
            wordSquare.add(word);
            findWordSquares(prefixMap, wordSquare, result);
        }

        return result;
    }

    private void findWordSquares(Map<String, List<String>> prefixMap, List<String> wor
dSquare, List<List<String>> result) {
        int size = wordSquare.get(0).length();
        int index = wordSquare.size();

        if (index == size) {
            result.add(new ArrayList<>(wordSquare));
            return;
        }

        StringBuilder prefixBuilder = new StringBuilder();

        for (String word : wordSquare) {
            prefixBuilder.append(word.charAt(index));
        }

        String prefix = prefixBuilder.toString();

        if (!prefixMap.containsKey(prefix)) {
            return;
        }
```

```
        for (String nextWord : prefixMap.get(prefix)) {
            wordSquare.add(nextWord);
            findWordSquares(prefixMap, wordSquare, result);
            wordSquare.remove(wordSquare.size() - 1);
        }
    }
}
```

1. Given an array of integers, find the kth smallest element in it.

```
import java.util.*;

public class KthSmallestElement {
    public int kthSmallest(int[] nums, int k) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder
());

        for (int num : nums) {
            maxHeap.offer(num);
            if (maxHeap.size() > k) {
                maxHeap.poll();
            }
        }

        return maxHeap.peek();
    }
}
```


1. Implement a function to calculate the n-th Fibonacci number efficiently using memoization.

```
import java.util.*;

public class FibonacciMemoization {
    public int fib(int n) {
        Map<Integer, Integer> memo = new HashMap<>();
        return fibHelper(n, memo);
    }

    private int fibHelper(int n, Map<Integer, Integer> memo) {
        if (memo.containsKey(n)) {
            return memo.get(n);
        }

        if (n <= 1) {
            return n;
        }

        int fibN = fibHelper(n - 1, memo) + fibHelper(n - 2, memo);
        memo.put(n, fibN);
        return fibN;
```

```
    }
}
```

1. Given a list of meeting intervals, merge overlapping intervals.

```
import java.util.*;

class Interval {
    int start;
    int end;

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

public class MergeIntervals {
    public List<Interval> merge(List<Interval> intervals) {
        if (intervals == null || intervals.size() <= 1) {
            return intervals;
        }

        Collections.sort(intervals, (a, b) -> a.start - b.start);

        List<Interval> mergedIntervals = new ArrayList<>();
        Interval currentInterval = intervals.get(0);

        for (int i = 1; i < intervals.size(); i++) {
            Interval nextInterval = intervals.get(i);
            if (currentInterval.end >= nextInterval.start) {
                currentInterval.end = Math.max(currentInterval.end, nextInterval.end);
            } else {
                mergedIntervals.add(currentInterval);
                currentInterval = nextInterval;
            }
        }

        mergedIntervals.add(currentInterval);
        return mergedIntervals;
    }
}
```


1. Write a Java program to perform matrix multiplication efficiently.

```
public class MatrixMultiplication {
    public int[][] multiply(int[][] A, int[][] B) {
        int m = A.length;
        int n = A[0].length;
        int p = B[0].length;

        int[][] result = new int[m][p];
```

```
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < p; j++) {
                for (int k = 0; k < n; k++) {
                    result[i][j] += A[i][k] * B[k][j];
                }
            }
        }

        return result;
    }
}
```

1. Implement a function to find the longest common subsequence of two strings.

```
public class LongestCommonSubsequence {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();
        int[][] dp = new int[m + 1][n + 1];

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[m][n];
    }
}
```


1. Write a Java program to implement a binary indexed tree (Fenwick tree).

```
public class FenwickTree {
    private int[] BIT;

    public FenwickTree(int n) {
        BIT = new int[n + 1];
    }

    public void update(int index, int value) {
        while (index < BIT.length) {
            BIT[index] += value;
            index += index & -index;
        }
    }

    public int query(int index) {
```

```
        int sum = 0;
        while (index > 0) {
            sum += BIT[index];
            index -= index & -index;
        }
        return sum;
    }

    public int rangeSum(int left, int right) {
        return query(right) - query(left - 1);
    }
}
```

1. Given a sorted array of integers, write a Java program to find

the closest elements to a target value.

```
import java.util.*;

public class ClosestElements {
    public List<Integer> findClosestElements(int[] arr, int k, int target) {
        int left = 0;
        int right = arr.length - k;

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (target - arr[mid] > arr[mid + k] - target) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        List<Integer> closestElements = new ArrayList<>();
        for (int i = left; i < left + k; i++) {
            closestElements.add(arr[i]);
        }

        return closestElements;
    }
}
```


1. Implement a function to find the minimum number of coins needed to make a given amount using dynamic programming.

```
import java.util.*;

public class CoinChange {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
```

```
        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }

        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

1. Given a list of unique numbers, write a Java program to find all permutations of the numbers.

```
import java.util.*;

public class Permutations {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        boolean[] used = new boolean[nums.length];
        List<Integer> current = new ArrayList<>();
        permuteHelper(nums, used, current, result);
        return result;
    }

    private void permuteHelper(int[] nums, boolean[] used, List<Integer> current, List
<List<Integer>> result) {
        if (current.size() == nums.length) {
            result.add(new ArrayList<>(current));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (!used[i]) {
                used[i] = true;
                current.add(nums[i]);
                permuteHelper(nums, used, current, result);
                current.remove(current.size() - 1);
                used[i] = false;
            }
        }
    }
}
```


1. Write a Java program to implement a stack with the ability to get the minimum element in constant time.

```
import java.util.*;

public class MinStack {
    private Stack<Integer> stack;
```

```
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

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

    public void pop() {
        if (!stack.isEmpty()) {
            int val = stack.pop();
            if (val == minStack.peek()) {
                minStack.pop();
            }
        }
    }

    public int top() {
        return stack.isEmpty() ? -1 : stack.peek();
    }

    public int getMin() {
        return minStack.isEmpty() ? -1 : minStack.peek();
    }
}
```


1. Implement a function to find the maximum sum subarray of a given array using Kadane's algorithm.

```
public class MaximumSubarray {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];

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

        return maxSum;
    }
}
```

1. Given an n x n matrix representing an image, write a Java program to rotate the image by 90 degrees (clockwise).

```
public class RotateImage {
    public void rotate(int[][] matrix) {
        int n = matrix.length;

        // Transpose the matrix
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }

        // Reverse each row
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n / 2; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[i][n - 1 - j];
                matrix[i][n - 1 - j] = temp;
            }
        }
    }
}
```

1. Write a Java program to implement a trie (prefix tree) with insert, search, and delete operations.


```
class TrieNode {
    boolean isWord;
    TrieNode[] children;

    TrieNode() {
        isWord = false;
        children = new TrieNode[26];
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
```

```
        node.isWord = true;
    }

    public boolean search(String word) {
        TrieNode node = findNode(word);
        return node != null && node.isWord;
    }

    public boolean startsWith(String prefix) {
        TrieNode node = findNode(prefix);
        return node != null;
    }

    private TrieNode findNode(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return null;
            }
            node = node.children[index];
        }
        return node;
    }

    public void delete(String word) {
        deleteHelper(root, word, 0);
    }

    private boolean deleteHelper(TrieNode node, String word, int depth) {
        if (node == null) {
            return false;
        }

        if (depth == word.length()) {
            if (!node.isWord) {
                return false;
            }
            node.isWord = false;
            return allChildrenNull(node);
        }

        int index = word.charAt(depth) - 'a';
        if (deleteHelper(node.children[index], word, depth + 1)) {
            node.children[index] = null;
            return !node.isWord && allChildrenNull(node);
        }

        return false;
    }

    private boolean allChildrenNull(TrieNode node) {
        for (TrieNode child : node.children) {
            if (child != null) {
                return false;
            }
        }
        return true;
```

```
    }
}
```

1. Given a list of non-overlapping intervals, insert a new interval and merge if necessary.

```
import java.util.*;

public class InsertInterval {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> mergedIntervals = new ArrayList<>();
        int i = 0;

        while (i < intervals.length && intervals[i][1] < newInterval[0]) {
            mergedIntervals.add(intervals[i]);
            i++;
        }

        while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
            newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
            i++;
        }

        mergedIntervals.add(newInterval);

        while (i < intervals.length) {
            mergedIntervals.add(intervals[i]);
            i++;
        }

        return mergedIntervals.toArray(new int[mergedIntervals.size()][]);
    }
}
```


1. Write a Java program to find the maximum profit by buying and selling stocks (multiple transactions allowed).

```
public class BestTimeToBuyAndSellStockII {
    public int maxProfit(int[] prices) {
        int maxProfit = 0;

        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1]) {
                maxProfit += prices[i] - prices[i - 1];
            }
        }

        return maxProfit;
```

```
    }
}
```

1. Implement a function to serialize and deserialize a binary tree.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class SerializeDeserializeBinaryTree {
    // Serialize a binary tree to a string
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    private void serializeHelper(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("null").append(",");
            return;
        }

        sb.append(node.val).append(",");
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
    }

    // Deserialize a string to a binary tree
    public TreeNode deserialize(String data) {
        String[] nodes = data.split(",");
        Queue<String> queue = new LinkedList<>(Arrays.asList(nodes));
        return deserializeHelper(queue);
    }

    private TreeNode deserializeHelper(Queue<String> queue) {
        String val = queue.poll();
        if (val.equals("null")) {
            return null;
        }

        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.left = deserializeHelper(queue);
        node.right = deserializeHelper(queue);
        return node;
    }
}
```

1. Given an array of integers, write a Java program to find the largest sum of a contiguous subarray.

```
public class MaximumSubarraySum {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];

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

        return maxSum;
    }
}
```

1. Write a Java program to implement a priority queue using a min-heap.

```
import java.util.*;

public class MinHeapPriorityQueue {
    private int[] heap;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    public MinHeapPriorityQueue() {
        heap = new int[DEFAULT_CAPACITY];
        size = 0;
    }

    public void offer(int val) {
        if (size == heap.length - 1) {
            resize();
        }

        size++;
        int index = size;
        heap[index] = val;

        while (index > 1 && heap[index] < heap[index / 2]) {
            swap(index, index / 2);
            index = index / 2;
        }
    }

    public int poll() {
        if (size == 0) {
            throw new NoSuchElementException("Priority queue is empty.");
        }

        int min = heap[1];
        heap[1] = heap[size];
```

```
        size--;

        int index = 1;
        while (true) {
            int smallest = index;
            int leftChild = index * 2;
            int rightChild = index * 2 + 1;

            if (leftChild <= size && heap[leftChild] < heap[smallest]) {
                smallest = leftChild;
            }

            if (rightChild <= size && heap[rightChild] < heap[smallest]) {
                smallest = rightChild;
            }

            if (smallest == index) {
                break;
            }

            swap(index, smallest);
            index = smallest;
        }

        return min;
    }

    public int peek() {
        if (size == 0) {
            throw new NoSuchElementException("Priority queue is empty.");
        }

        return heap[1];
    }

    public boolean isEmpty() {
        return size == 0;
    }

    private void resize() {
        int[] newHeap = new int[heap.length * 2];
        System.arraycopy(heap, 0, newHeap, 0, heap.length);
        heap = newHeap;
    }

    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }
}
```


1. Implement a function to find the longest common prefix among an array of strings.

```
public class LongestCommonPrefix {
    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }

        String prefix = strs[0];

        for (int i = 1; i < strs.length; i++) {
            while (!strs[i].startsWith(prefix)) {
                prefix = prefix.substring(0, prefix.length() - 1);
            }
        }

        return prefix;
    }
}
```

1. Given a list of intervals, find the minimum number of intervals to be removed to make the rest of the intervals non-overlapping.

```
import java.util.*;

class Interval {
    int start;
    int end;

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

public class NonOverlappingIntervals {
    public int eraseOverlapIntervals(Interval[] intervals) {
        if (intervals == null || intervals.length <= 1) {
            return 0;
        }

        Arrays.sort(intervals, (a, b) -> a.end - b.end);

        int count = 0;
        int end = intervals[0].end;

        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i].start < end) {
                count++;
            } else {
                end = intervals[i].end;
            }
        }

        return count;
```

```
    }
}
```

1. Write a Java program to find the longest palindromic substring in a given string.

```
public class LongestPal

indromicSubstring {
    public String longestPalindrome(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        String longestPalindrome = "";

        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                dp[i][j] = (s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j -
 1]));

                if (dp[i][j] && (longestPalindrome.isEmpty() || j - i + 1 > longestPal
indrome.length())) {
                    longestPalindrome = s.substring(i, j + 1);
                }
            }
        }

        return longestPalindrome;
    }
}
```


1. Given a list of words, write a Java program to find all valid word squares.

```
import java.util.*;

public class WordSquares {
    public List<List<String>> wordSquares(String[] words) {
        List<List<String>> result = new ArrayList<>();
        Map<String, List<String>> prefixMap = new HashMap<>();

        for (String word : words) {
            for (int i = 1; i <= word.length(); i++) {
                String prefix = word.substring(0, i);
                prefixMap.putIfAbsent(prefix, new ArrayList<>());
                prefixMap.get(prefix).add(word);
            }
        }

        for (String word : words) {
            List<String> wordSquare = new ArrayList<>();
            wordSquare.add(word);
            findWordSquares(prefixMap, wordSquare, result);
        }
```

```
        return result;
    }

    private void findWordSquares(Map<String, List<String>> prefixMap, List<String> wor
dSquare, List<List<String>> result) {
        int size = wordSquare.get(0).length();
        int index = wordSquare.size();

        if (index == size) {
            result.add(new ArrayList<>(wordSquare));
            return;
        }

        StringBuilder prefixBuilder = new StringBuilder();

        for (String word : wordSquare) {
            prefixBuilder.append(word.charAt(index));
        }

        String prefix = prefixBuilder.toString();

        if (!prefixMap.containsKey(prefix)) {
            return;
        }

        for (String nextWord : prefixMap.get(prefix)) {
            wordSquare.add(nextWord);
            findWordSquares(prefixMap, wordSquare, result);
            wordSquare.remove(wordSquare.size() - 1);
        }
    }
}
```


1. Implement a function to find the kth largest element in an unsorted array.

```
import java.util.*;

public class KthLargestElement {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }
        return minHeap.peek();
    }
}
```

1. Write a Java program to find the longest increasing subsequence in an array.

```
public class LongestIncreasingSubsequence {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int maxLength = 0;

        for (int i = 0; i < nums.length; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }

        return maxLength;
    }
}
```

1. Given a list of strings, write a Java program to find the longest word made of other words in the list.

```
import java.util.*;

public class LongestWordMadeOfOtherWords {
    public String longestWord(String[] words) {
        Set<String> wordSet = new HashSet<>(Arrays.asList(words));
        Arrays.sort(words, (a, b) -> a.length() != b.length() ? b.length() - a.length
() : a.compareTo(b));

        for (String word : words) {
            wordSet.remove(word);
            if (isMadeOfOtherWords(word, wordSet)) {
                return word;
            }
            wordSet.add(word);
        }

        return "";
    }

    private boolean isMadeOfOtherWords(String word, Set<String> wordSet) {
        if (word.isEmpty()) {
            return true;
        }

        for (int i = 1; i <= word.length(); i++) {
            if (wordSet.contains(word.substring(0, i)) && isMadeOfOtherWords(word.subs
tring(i), wordSet)) {
                return true;
            }
        }
```

```
            return false;
        }
    }
```