Answer 1)

```
import java.util.Arrays;

public class ArrayPairSum {
    public static int arrayPairSum(int[] nums) {
        Arrays.sort(nums); // Sort the array in ascending order
        int sum = 0;
        for (int i = 0; i < nums.length; i += 2) {
            sum += nums[i]; // Sum the minimum value of each pair
        }
        return sum;
    }

    public static void main(String[] args) {
        int[] nums = {1, 4, 3, 2};
        int maxSum = arrayPairSum(nums);
        System.out.println("Maximized sum: " + maxSum);
    }
}
```

Answer 2)

```
import java.util.HashSet;

public class MaxCandies {
    public static int maxCandies(int[] candyType) {
        int maxEat = candyType.length / 2; // Maximum candies Alice can eat
        HashSet<Integer> uniqueCandies = new HashSet<>();

        for (int candy : candyType) {
            uniqueCandies.add(candy); // Add each candy type to the set
        }

        return Math.min(uniqueCandies.size(), maxEat); // Return the minimum of unique candy
types and maxEat
    }

    public static void main(String[] args) {
        int[] candyType = {1, 1, 2, 2, 3, 3};
        int maxNum = maxCandies(candyType);
        System.out.println("Maximum number of different candy types Alice can eat: " + maxNum);
    }
}
```

Answer 3)

```
import java.util.HashMap;

public class LongestHarmoniousSubsequence {
    public static int findLHS(int[] nums) {
        HashMap<Integer, Integer> frequencyMap = new HashMap<>();
        int longestSubsequenceLength = 0;

        // Count the frequency of each number in the array
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Iterate over the numbers in the array
        for (int num : nums) {
            // Check if there is a number with a difference of 1 in frequency
            if (frequencyMap.containsKey(num + 1)) {
                int currentSubsequenceLength = frequencyMap.get(num) + frequencyMap.get(num +
1);
                longestSubsequenceLength = Math.max(longestSubsequenceLength,
currentSubsequenceLength);
            }
        }

        return longestSubsequenceLength;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 2, 2, 5, 2, 3, 7};
        int longestSubsequenceLength = findLHS(nums);
        System.out.println("Length of the longest harmonious subsequence: " +
longestSubsequenceLength);
    }
}
```

Answer 4)

```
public class FlowerPlanting {
    public static boolean canPlaceFlowers(int[] flowerbed, int n) {
        int count = 0;
        int i = 0;
        while (i < flowerbed.length) {
            // Check if the current plot and its adjacent plots are empty
```

```
            if (flowerbed[i] == 0 && (i == 0 || flowerbed[i - 1] == 0) && (i == flowerbed.length - 1 ||
flowerbed[i + 1] == 0)) {
                flowerbed[i] = 1; // Plant a flower
                count++; // Increment the count of planted flowers
            }
            i++; // Move to the next plot
        }
        return count >= n; // Return true if the count of planted flowers is greater than or equal to n
    }

    public static void main(String[] args) {
        int[] flowerbed = {1, 0, 0, 0, 1};
        int n = 1;
        boolean canPlant = canPlaceFlowers(flowerbed, n);
        System.out.println("Can plant " + n + " flowers: " + canPlant);
    }
}
```

Answer 5)

```
import java.util.Arrays;

public class MaximumProduct {
    public static int maximumProduct(int[] nums) {
        Arrays.sort(nums); // Sort the array in ascending order
        int n = nums.length;
        // The maximum product can be either the product of the three largest numbers or the
product of the two smallest numbers and the largest number
        return Math.max(nums[n - 1] * nums[n - 2] * nums[n - 3], nums[0] * nums[1] * nums[n - 1]);
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        int maxProduct = maximumProduct(nums);
        System.out.println("Maximum product: " + maxProduct);
    }
}
```

Answer 6)

```
public class BinarySearch {
    public static 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; // Target found, return the index
            } else if (nums[mid] < target) {
                left = mid + 1; // Target is in the right half
            } else {
                right = mid - 1; // Target is in the left half
            }
        }

        return -1; // Target not found
    }

    public static void main(String[] args) {
        int[] nums = {-1, 0, 3, 5, 9, 12};
        int target = 9;
        int index = search(nums, target);
        System.out.println("Index of target " + target + ": " + index);
    }
}
```

Answer 7)

```
public class MonotonicArray {
    public static boolean isMonotonic(int[] nums) {
        boolean increasing = true;
        boolean decreasing = true;

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < nums[i - 1]) {
                increasing = false;
            }
            if (nums[i] > nums[i - 1]) {
                decreasing = false;
            }
        }

        return increasing || decreasing;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 2, 3};
        boolean isMonotonic = isMonotonic(nums);
        System.out.println("Is the array monotonic? " + isMonotonic);
```

```
    }
}
```

Answer 8)

```
import java.util.Arrays;

public class MinimumScore {
    public static int minScore(int[] nums, int k) {
        int n = nums.length;
        Arrays.sort(nums); // Sort the array in ascending order
        int minScore = nums[n - 1] - nums[0]; // Initialize the minimum score with the difference between the maximum and minimum elements

        // Check the range of x from -k to k for each element in nums
        for (int i = 0; i < n - 1; i++) {
            int min = Math.min(nums[0] + k, nums[i + 1] - k);
            int max = Math.max(nums[i] + k, nums[n - 1] - k);
            minScore = Math.min(minScore, max - min); // Update the minimum score if a smaller difference is found
        }

        return minScore;
    }

    public static void main(String[] args) {
        int[] nums = {1};
        int k = 0;
        int minScore = minScore(nums, k);
        System.out.println("Minimum score: " + minScore);
    }
}
```