1.) public class SortColors {

public void sortColors(int[] nums) {

int low = 0;

int high = nums.length - 1;

int current = 0;

while (current <= high) {

if (nums[current] == 0) {

swap(nums, current, low);

current++;

low++;

} else if (nums[current] == 2) {

swap(nums, current, high);

high--;

} else {

current++;

}

}

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

public static void main(String[] args) {

int[] colors = {2, 0, 2, 1, 1, 0};

SortColors sorter = new SortColors();

sorter.sortColors(colors);

for (int color : colors) {

System.out.print(color + " ");

}

}

}

2.) public class NobleInteger {

public static int findNobleInteger(int[] nums) {

for (int num : nums) {

int count = 0;

for (int other : nums) {

if (other > num) {

count++;

}

}

if (count == num) {

return num;

}

}

return -1; // No noble integer found

}

public static void main(String[] args) {

int[] nums = {3, 1, 4, 2, 0, 6};

int noble = findNobleInteger(nums);

if (noble != -1) {

System.out.println("Noble Integer: " + noble);

} else {

System.out.println("No Noble Integer found");

}

}

}

3.) import java.util.HashMap;

import java.util.Map;

public class LongestArithmeticProgression {

public static int longestAPLength(int[] nums) {

if (nums == null || nums.length == 0) {

return 0;

}

int n = nums.length;

int maxLength = 2; // The minimum length of an arithmetic progression is 2

// dp[i][j] represents the length of the longest arithmetic progression ending with nums[i] and nums[j]

// The key is the difference between nums[i] and nums[j], and the value is the length of the AP

Map<Integer, Integer>[] dp = new HashMap[n];

for (int i = 0; i < n; i++) {

dp[i] = new HashMap<>();

for (int j = 0; j < i; j++) {

int diff = nums[i] - nums[j];

dp[i].put(diff, dp[j].getOrDefault(diff, 1) + 1);

maxLength = Math.max(maxLength, dp[i].get(diff));

}

}

return maxLength;

}

public static void main(String[] args) {

int[] nums = {1, 7, 10, 15, 27, 29};

int length = longestAPLength(nums);

System.out.println("Length of Longest Arithmetic Progression: " + length);

}

}

4.) import java.util.Arrays;

import java.util.Comparator;

public class LargestNumber {

public static String largestNumber(int[] nums) {

if (nums == null || nums.length == 0) {

return "0";

}

// Convert the integer array to an array of strings

String[] numStrings = new String[nums.length];

for (int i = 0; i < nums.length; i++) {

numStrings[i] = Integer.toString(nums[i]);

}

// Sort the array of strings using a custom comparator

Arrays.sort(numStrings, new Comparator<String>() {

public int compare(String a, String b) {

String order1 = a + b;

String order2 = b + a;

return order2.compareTo(order1); // Compare in reverse order

}

});

// Handle the case where the largest number is 0

if (numStrings[0].equals("0")) {

return "0";

}

// Concatenate the sorted strings to form the largest number

StringBuilder result = new StringBuilder();

for (String numString : numStrings) {

result.append(numString);

}

return result.toString();

}

public static void main(String[] args) {

int[] nums = {3, 30, 34, 5, 9};

String largest = largestNumber(nums);

System.out.println("Largest Number: " + largest);

}

}

5.) public class RemoveElement {

public static int removeElement(int[] nums, int val) {

int newIndex = 0; // Index to track the new position for elements that are not equal to 'val'

for (int i = 0; i < nums.length; i++) {

if (nums[i] != val) {

nums[newIndex] = nums[i];

newIndex++;

}

}

return newIndex;

}

public static void main(String[] args) {

int[] nums = {3, 2, 2, 3};

int val = 3;

int newLength = removeElement(nums, val);

System.out.println("New Length of Array: " + newLength);

}

}

6.) public class MergeSortedArrays {

public static int[] mergeArrays(int[] arr1, int[] arr2) {

int len1 = arr1.length;

int len2 = arr2.length;

int[] merged = new int[len1 + len2];

int i = 0, j = 0, k = 0; // Pointers for arr1, arr2, and merged array

while (i < len1 && j < len2) {

if (arr1[i] < arr2[j]) {

merged[k] = arr1[i];

i++;

} else {

merged[k] = arr2[j];

j++;

}

k++;

}

while (i < len1) {

merged[k] = arr1[i];

i++;

k++;

}

while (j < len2) {

merged[k] = arr2[j];

j++;

k++;

}

return merged;

}

public static void main(String[] args) {

int[] arr1 = {1, 3, 5, 7};

int[] arr2 = {2, 4, 6, 8};

int[] mergedArray = mergeArrays(arr1, arr2);

System.out.print("Merged Sorted Array: ");

for (int num : mergedArray) {

System.out.print(num + " ");

}

}

}

7.) public class KthSmallestElement {

public static int findKthSmallest(int[] nums, int k) {

if (nums == null || k <= 0 || k > nums.length) {

throw new IllegalArgumentException("Invalid input");

}

return quickSelect(nums, 0, nums.length - 1, k - 1);

}

private static int quickSelect(int[] nums, int low, int high, int k) {

int pivotIndex = partition(nums, low, high);

if (pivotIndex == k) {

return nums[pivotIndex];

} else if (pivotIndex < k) {

return quickSelect(nums, pivotIndex + 1, high, k);

} else {

return quickSelect(nums, low, pivotIndex - 1, k);

}

}

private static int partition(int[] nums, int low, int high) {

int pivot = nums[high];

int i = low;

for (int j = low; j < high; j++) {

if (nums[j] <= pivot) {

swap(nums, i, j);

i++;

}

}

swap(nums, i, high);

return i;

}

private static void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

public static void main(String[] args) {

int[] nums = {7, 10, 4, 3, 20, 15};

int k = 3;

int kthSmallest = findKthSmallest(nums, k);

System.out.println("Kth Smallest Element: " + kthSmallest);

}

}

8.) public class ConsecutiveArray {

public static boolean isConsecutive(int[] nums) {

int n = nums.length;

if (n <= 1) {

return true;

}

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

int actualSum = 0;

for (int num : nums) {

min = Math.min(min, num);

max = Math.max(max, num);

actualSum += num;

}

int expectedSum = (max \* (max + 1) - min \* (min - 1)) / 2;

return expectedSum == actualSum && max - min == n - 1;

}

public static void main(String[] args) {

int[] nums = {5, 3, 1, 2, 4};

boolean result = isConsecutive(nums);

System.out.println("Array contains consecutive elements: " + result);

}

}

9.) public class MaxMod {

public static int maxModValue(int[] nums) {

int maxMod = 0;

for (int i = 0; i < nums.length; i++) {

for (int j = i + 1; j < nums.length; j++) {

maxMod = Math.max(maxMod, nums[i] % nums[j]);

}

}

return maxMod;

}

public static void main(String[] args) {

int[] nums = {1, 2, 3, 4, 5};

int result = maxModValue(nums);

System.out.println("Maximum value of (arr[i] % arr[j]): " + result);

}

}

10.) import java.util.Arrays;

public class ChocolateDistribution {

public static int minDifference(int[] arr, int n, int k) {

if (k == 0 || n == 0) {

return 0;

}

if (k > n) {

return -1; // Invalid input

}

Arrays.sort(arr);

int minDifference = Integer.MAX\_VALUE;

for (int i = 0; i + k - 1 < n; i++) {

int difference = arr[i + k - 1] - arr[i];

minDifference = Math.min(minDifference, difference);

}

return minDifference;

}

public static void main(String[] args) {

int[] arr = {3, 4, 1, 9, 56, 7, 9, 12};

int k = 5;

int n = arr.length;

int result = minDifference(arr, n, k);

if (result == -1) {

System.out.println("Invalid input");

} else {

System.out.println("Minimum Difference: " + result);

}

}

}

11.) public class InversionCount {

public static int mergeAndCount(int[] arr, int[] temp, int left, int mid, int right) {

int i = left; // Pointer for the left subarray

int j = mid + 1; // Pointer for the right subarray

int k = left; // Pointer for the merged array

int invCount = 0; // Initialize the inversion count

while (i <= mid && j <= right) {

if (arr[i] <= arr[j]) {

temp[k++] = arr[i++];

} else {

// If arr[i] > arr[j], it's an inversion

temp[k++] = arr[j++];

invCount += (mid - i + 1); // Count inversions

}

}

while (i <= mid) {

temp[k++] = arr[i++];

}

while (j <= right) {

temp[k++] = arr[j++];

}

// Copy the merged array back to the original array

for (i = left; i <= right; i++) {

arr[i] = temp[i];

}

return invCount;

}

public static int mergeSortAndCount(int[] arr, int[] temp, int left, int right) {

int invCount = 0;

if (left < right) {

int mid = (left + right) / 2;

invCount += mergeSortAndCount(arr, temp, left, mid);

invCount += mergeSortAndCount(arr, temp, mid + 1, right);

invCount += mergeAndCount(arr, temp, left, mid, right);

}

return invCount;

}

public static int inversionCount(int[] arr) {

int n = arr.length;

int[] temp = new int[n];

return mergeSortAndCount(arr, temp, 0, n - 1);

}

public static void main(String[] args) {

int[] arr = {1, 20, 6, 4, 5};

int inversions = inversionCount(arr);

System.out.println("Number of Inversions: " + inversions);

}

}

12.) import java.util.Arrays;

import java.util.Comparator;

public class LargestNumber {

public static String largestNumber(int[] arr) {

// Convert the array of integers to an array of strings

String[] strArr = new String[arr.length];

for (int i = 0; i < arr.length; i++) {

strArr[i] = Integer.toString(arr[i]);

}

// Sort the array of strings using a custom comparator

Arrays.sort(strArr, new Comparator<String>() {

@Override

public int compare(String a, String b) {

String order1 = a + b;

String order2 = b + a;

return order2.compareTo(order1); // Compare in reverse order to get the largest number

}

});

// Handle the case where the largest number is 0

if (strArr[0].equals("0")) {

return "0";

}

// Concatenate the sorted strings to form the largest number

StringBuilder largestNumber = new StringBuilder();

for (String numStr : strArr) {

largestNumber.append(numStr);

}

return largestNumber.toString();

}

public static void main(String[] args) {

int[] arr = {10, 2};

String result = largestNumber(arr);

System.out.println("Output: " + result);

}

}

13.) import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

public class UniqueElements {

public static List<Integer> getUniqueElements(int[] arr) {

Set<Integer> uniqueSet = new HashSet<>();

List<Integer> uniqueList = new ArrayList<>();

for (int num : arr) {

if (!uniqueSet.contains(num)) {

uniqueSet.add(num);

uniqueList.add(num);

}

}

return uniqueList;

}

public static void main(String[] args) {

int[] arr = {1, 2, 3, 2, 1, 5};

List<Integer> uniqueElements = getUniqueElements(arr);

System.out.print("Output: [");

for (int i = 0; i < uniqueElements.size(); i++) {

System.out.print(uniqueElements.get(i));

if (i < uniqueElements.size() - 1) {

System.out.print(", ");

}

}

System.out.println("]");

}

}

14.) public class ReversePairs {

public static int countReversePairs(int[] nums) {

return mergeSortCount(nums, 0, nums.length - 1);

}

private static int mergeSortCount(int[] nums, int left, int right) {

if (left >= right) {

return 0;

}

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

int count = mergeSortCount(nums, left, mid) + mergeSortCount(nums, mid + 1, right);

int j = mid + 1;

for (int i = left; i <= mid; i++) {

while (j <= right && nums[i] > 2L \* nums[j]) {

j++;

}

count += (j - (mid + 1));

}

merge(nums, left, mid, right);

return count;

}

private static void merge(int[] nums, int left, int mid, int right) {

int[] temp = new int[right - left + 1];

int i = left, j = mid + 1, k = 0;

while (i <= mid && j <= right) {

if (nums[i] <= nums[j]) {

temp[k++] = nums[i++];

} else {

temp[k++] = nums[j++];

}

}

while (i <= mid) {

temp[k++] = nums[i++];

}

while (j <= right) {

temp[k++] = nums[j++];

}

for (int idx = 0; idx < temp.length; idx++) {

nums[left + idx] = temp[idx];

}

}

public static void main(String[] args) {

int[] arr = {1, 3, 2, 3, 1};

int result = countReversePairs(arr);

System.out.println("Output: " + result);

}

}

15.) import java.util.Arrays;

import java.util.PriorityQueue;

public class ClosestPointsToOrigin {

public static int[][] kClosest(int[][] points, int k) {

// Create a max heap priority queue with a custom comparator

PriorityQueue<int[]> maxHeap = new PriorityQueue<>((a, b) ->

(b[0]\*b[0] + b[1]\*b[1]) - (a[0]\*a[0] + a[1]\*a[1])

);

// Add points to the max heap

for (int[] point : points) {

maxHeap.offer(point);

if (maxHeap.size() > k) {

maxHeap.poll(); // Remove the farthest point if size exceeds k

}

}

// Convert the max heap to an array

int[][] result = new int[k][2];

int i = 0;

while (!maxHeap.isEmpty()) {

result[i] = maxHeap.poll();

i++;

}

return result;

}

public static void main(String[] args) {

int[][] points = {{1, 3}, {-2, 2}};

int k = 1;

int[][] result = kClosest(points, k);

System.out.println("Output:");

for (int[] point : result) {

System.out.println(Arrays.toString(point));

}

}

}

16.) import java.util.Arrays;

public class SumOfDifferences {

public static int sumOfAbsoluteDifferences(int[] nums) {

Arrays.sort(nums);

int sum = 0;

for (int i = 0; i < nums.length; i++) {

sum += (i \* nums[i] - (nums.length - i - 1) \* nums[i]);

}

return sum;

}

public static void main(String[] args) {

int[] nums = {1, 2, 3};

int result = sumOfAbsoluteDifferences(nums);

System.out.println("Output: " + result);

}

}

17.) public class QuickSort {

public static void quickSort(int[] arr) {

quickSort(arr, 0, arr.length - 1);

}

private static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void main(String[] args) {

int[] arr = {5, 1, 4, 2, 8};

quickSort(arr);

System.out.print("Output: [");

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i]);

if (i < arr.length - 1) {

System.out.print(", ");

}

}

System.out.println("]");

}

}

18.) import java.util.Arrays;

public class MaximumUnsortedSubarray {

public static int[] findUnsortedSubarray(int[] nums) {

int[] sortedNums = Arrays.copyOf(nums, nums.length);

Arrays.sort(sortedNums);

int start = -1, end = -1;

for (int i = 0; i < nums.length; i++) {

if (nums[i] != sortedNums[i]) {

if (start == -1) {

start = i;

}

end = i;

}

}

if (start == -1) {

// The array is already sorted

return new int[]{};

}

return Arrays.copyOfRange(nums, start, end + 1);

}

public static void main(String[] args) {

int[] nums = {1, 3, 2, 4, 5};

int[] result = findUnsortedSubarray(nums);

System.out.print("Output: [");

for (int i = 0; i < result.length; i++) {

System.out.print(result[i]);

if (i < result.length - 1) {

System.out.print(", ");

}

}

System.out.println("]");

}

}

19.) public class MaxMinMagicExpression {

public static void findMaxMinMagic(int[] arr) {

int n = arr.length;

int maxVal = Integer.MIN\_VALUE;

int minVal = Integer.MAX\_VALUE;

for (int i = 0; i < n; i++) {

for (int j = i; j < n; j++) {

int expressionValue = (arr[i] \* arr[i]) + (arr[j] \* arr[j]) + (i - j) \* (i - j);

maxVal = Math.max(maxVal, expressionValue);

minVal = Math.min(minVal, expressionValue);

}

}

System.out.println("Maximum Value: " + maxVal);

System.out.println("Minimum Value: " + minVal);

}

public static void main(String[] args) {

int[] arr = {1, 2, 3};

findMaxMinMagic(arr);

}

}