JAVA Question Bank

1.	Maximum SubArray, Kadanes's Algorithm	[Easy]

Given an integer array nums, find the
subarray
with the largest sum, and return *its sum*.

Example 1:
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
Example 2:
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
Example 3:
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

- $1 <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$

Follow up: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.
32

```
import java.util.*;

public class Main {
   public static long maxSubarraySum(int[] arr, int n) {
      long maxi = Long.MIN_VALUE; // maximum sum
      long sum = 0;

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

         sum += arr[i];

         if (sum > maxi) {
            maxi = sum;
         }
```

```
            // If sum < 0: discard the sum calculated
            if (sum < 0) {
                sum = 0;
            }
        }

        // To consider the sum of the empty subarray
        // uncomment the following check:

        //if (maxi < 0) maxi = 0;

        return maxi;
    }

    public static void main(String args[]) {
        int[] arr = { -2, 1, -3, 4, -1, 2, 1, -5, 4};
        int n = arr.length;
        long maxSum = maxSubarraySum(arr, n);
        System.out.println("The maximum subarray sum is: " + maxSum);

    }

}
```

Complexity Analysis

Time Complexity: O(N), where N = size of the array.
Reason: We are using a single loop running N times.

Space Complexity: O(1) as we are not using any extra space.

2.	Move Zeros	[Easy]
	Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements. Note that you must do this in-place without making a copy of the array. Example 1: Input: nums = [0,1,0,3,12] Output: [1,3,12,0,0] Example 2:	

	Input: nums = [0] Output: [0] Constraints: • $1 <= nums.length <= 10^4$ • $-2^{31} <= nums[i] <= 2^{31} - 1$

```
import java.util.*;

public class Demo {
   public static int[] moveZeros(int n, int []a) {
      int j = -1;
      //place the pointer j:
      for (int i = 0; i < n; i++) {
         if (a[i] == 0) {
            j = i;
            break;
         }
      }

      //no non-zero elements:
      if (j == -1) return a;

      //Move the pointers i and j
      //and swap accordingly:
      for (int i = j + 1; i < n; i++) {
         if (a[i] != 0) {
            //swap a[i] & a[j]:
            int tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
            j++;
         }
      }
      return a;
   }

   public static void main(String[] args) {
      int[] arr = {1, 0, 2, 3, 2, 0, 0, 4, 5, 1};
```

```
        int n = 10;
        int[] ans = moveZeros(n, arr);
        for (int i = 0; i < n; i++) {
            System.out.print(ans[i] + " ");
        }
        System.out.println("");
    }
}
```

Time Complexity: O(N), N = size of the array.
Reason: We have used 2 loops and using those loops, we are basically traversing the array once.
Space Complexity: O(1) as we are not using any extra space to solve this problem.

3.	Missing Number	[Easy]
	Given an array *'a'* of size *'n'*-1 with elements of range 1 to 'n'. The array does not contain any duplicates. Your task is to find the missing number. For example: Input: 'a' = [1, 2, 4, 5], 'n' = 5 Output : 3 Explanation: 3 is the missing value in the range 1 to 5. Detailed explanation (Input/output format, Notes, Images) Sample Input 1 : 4 1 2 3 Sample Output 1: 4 Explanation Of Sample Input 1: 4 is the missing value in the range 1 to 4. Sample Input 2: 8 1 2 3 5 6 7 8 Sample Output 2: 4 Explanation Of Sample Input 2: 4 is the missing value in the range 1 to 8. Expected time complexity: The expected time complexity is O(n).	

	Constraints: 1 <= 'n' <= 10^6 1 <= 'a'[i] <= 'n' Time Limit: 1 sec

```
import java.util.*;

public class Demo {
   public static int missingNumber(int []a, int N) {

      //Summation of first N numbers:
      int sum = (N * (N + 1)) / 2;

      //Summation of all array elements:
      int s2 = 0;
      for (int i = 0; i < N - 1; i++) {
         s2 += a[i];
      }

      int missingNum = sum - s2;
      return missingNum;
   }

   public static void main(String args[]) {
      int N = 5;
      int a[] = {1, 2, 4, 5};

      int ans = missingNumber(a, N);
      System.out.println("The missing number is: " + ans);
   }
}
```

Time Complexity: O(N), where N = size of array+1.
Reason: Here, we need only 1 loop to get the sum of the array elements. The loop runs for approx. N times. So, the time complexity is O(N).

Space Complexity: O(1) as we are not using any extra space.

4.	Two Sum	[Easy]
	Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order. Example 1: Input: nums = [2,7,11,15], target = 9 Output: [0,1] Explanation: Because nums[0] + nums[1] == 9, we return [0, 1]. Example 2: Input: nums = [3,2,4], target = 6 Output: [1,2] Example 3: Input: nums = [3,3], target = 6 Output: [0,1] Constraints: 2 <= nums.length <= 104 -109 <= nums[i] <= 109 -109 <= target <= 109 Only one valid answer exists. Follow-up: Can you come up with an algorithm that is less than O(n2) time complexity?	

```
import java.util.*;

public class Main {
   public static String twoSum(int n, int []arr, int target) {
      Arrays.sort(arr);
      int left = 0, right = n - 1;
      while (left < right) {
         int sum = arr[left] + arr[right];
         if (sum == target) {
            return "YES";
         } else if (sum < target) left++;
         else right--;
      }
      return "NO";
```

```
    }

    public static void main(String args[]) {
        int n = 5;
        int[] arr = {2, 6, 5, 8, 11};
        int target = 14;
        String ans = twoSum(n, arr, target);
        System.out.println("This is the answer for variant 1: " + ans);

    }

}
```

Time Complexity: O(N) + O(N*logN), where N = size of the array.
Reason: The loop will run at most N times. And sorting the array will take N*logN time complexity.
Space Complexity: O(1) as we are not using any extra space.

5.	Remove Duplicates	[Easy]
	You are given a sorted integer array 'arr' of size 'n'. You need to remove the duplicates from the array such that each element appears only once. Return the length of this new array. Note: Do not allocate extra space for another array. You need to do this by modifying the given input array in place with O(1) extra memory. For example: 'n' = 5, 'arr' = [1 2 2 2 3]. The new array will be [1 2 3]. So our answer is 3. Sample input 1: 10 1 2 2 3 3 3 4 4 5 5 Sample output 1: 5 Explanation of sample input 1: The new array will be [1 2 3 4 5]. So our answer is 5. Sample input 2: 9 1 1 2 3 3 4 5 5 5 Sample output 2: 5	

	Expected time complexity: The expected time complexity is O(n). Constraints : 1 <= 'n' <= 10^6 -10^9 <= 'arr[i]' <=10^9 Where 'arr[i]' is the value of elements of the array.

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
      int arr[] = {1,1,2,2,2,3,3};
       int k = removeDuplicates(arr);
       System.out.println("The array after removing duplicate elements is ");
       for (int i = 0; i < k; i++) {
          System.out.print(arr[i] + " ");
       }
   }
   static int removeDuplicates(int[] arr) {
       int i = 0;
       for (int j = 1; j < arr.length; j++) {
          if (arr[i] != arr[j]) {
             i++;
             arr[i] = arr[j];
          }
       }
       return i + 1;
   }
}
```

Time Complexity: O(N)
Space Complexity: O(1)

6.	SubArray Sum Equals K	[Medium]
	Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k. A subarray is a contiguous non-empty sequence of elements within an array. Example 1: Input: nums = [1,2,3], k = 3 Output: 2 Example 2:	

	Input: nums = [1,2,3,-3, 1,1,1,4,2,-3], k = 3 Output: 2 Constraints: 1 <= nums.length <= 2 * 104 -1000 <= nums[i] <= 1000 -107 <= k <= 107

```
import java.util.*;

public class Demo{
   public static int findAllSubarraysWithGivenSum(int arr[], int k) {
      int n = arr.length; // size of the given array.
      Map mpp = new HashMap();
      int preSum = 0, cnt = 0;

      mpp.put(0, 1); // Setting 0 in the map.
      for (int i = 0; i < n; i++) {
         // add current element to prefix Sum:
         preSum += arr[i];

         // Calculate x-k:
         int remove = preSum - k;

         // Add the number of subarrays to be removed:
         cnt += mpp.getOrDefault(remove, 0);

         // Update the count of prefix sum
         // in the map.
         mpp.put(preSum, mpp.getOrDefault(preSum, 0) + 1);
      }
      return cnt;
   }

   public static void main(String[] args) {
      int[] arr = {3, 1, 2, 4};
      int k = 6;
      int cnt = findAllSubarraysWithGivenSum(arr, k);
      System.out.println("The number of subarrays is: " + cnt);
   }
}
```

Time Complexity: O(N) or O(N*logN) depending on which map data structure we are using, where N = size of the array.
Space Complexity: O(N) as we are using a map data structure.

7.	Check Sorted Array	[Easy]

You have been given an array *'a'* of *'n'* non-negative integers.You have to check whether the given array is sorted in the non-decreasing order or not.

Your task is to return 1 if the given array is sorted. Else, return 0.

Example :
Input: 'n' = 5, 'a' = [1, 2, 3, 4, 5]
Output: 1

The given array is sorted in non-decreasing order; hence the answer will be 1.
Detailed explanation (Input/output format, Notes, Images)
Sample Input 1 :
4
0 0 0 1
Sample Output 1 :
1
Explanation For Sample Input 1 :
The given array is sorted in non-decreasing order; hence the answer will be 1.
Sample Input 2 :
5
4 5 4 4 4
Sample Output 2 :
0
Expected Time Complexity:
O(n), Where 'n' is the size of an input array 'a'.
Constraints:
1 ≤ 'n' ≤ 5*10^6
0 ≤ 'a'[i] ≤ 10^9

Time limit: 1 sec

```
class Demo {
  static boolean isSorted(int arr[], int n) {
    for (int i = 1; i < n; i++) {
      if (arr[i] < arr[i - 1])
        return false;
```

```
    }

    return true;
  }

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

    System.out.println(isSorted(arr, n));
  }
}
```

Time Complexity: O(N)
Space Complexity: O(1)

8.	Sort Colors	[Medium]
	Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue. We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively. You must solve this problem without using the library's sort function. Example 1: Input: nums = [2,0,2,1,1,0] Output: [0,0,1,1,2,2] Example 2: Input: nums = [2,0,1] Output: [0,1,2] Constraints: • n == nums.length • 1 <= n <= 300 • nums[i] is either 0, 1, or 2. Follow up: Could you come up with a one-pass algorithm using only constant extra space?	

```
import java.util.*;

public class Main {
   public static void sortArray(ArrayList<Integer> arr, int n) {
```

```
        int low = 0, mid = 0, high = n - 1; // 3 pointers

        while (mid <= high) {
            if (arr.get(mid) == 0) {
                // swapping arr[low] and arr[mid]
                int temp = arr.get(low);
                arr.set(low, arr.get(mid));
                arr.set(mid, temp);

                low++;
                mid++;

            } else if (arr.get(mid) == 1) {
                mid++;

            } else {
                // swapping arr[mid] and arr[high]
                int temp = arr.get(mid);
                arr.set(mid, arr.get(high));
                arr.set(high, temp);

                high--;
            }
        }
    }

    public static void main(String args[]) {
        int n = 6;
        ArrayList<Integer> arr = new ArrayList<>(Arrays.asList(new Integer[] {0, 2, 1, 2, 0,
1}));
        sortArray(arr, n);
        System.out.println("After sorting:");
        for (int i = 0; i < n; i++) {
            System.out.print(arr.get(i) + " ");
        }
        System.out.println();

    }

}
```

Time Complexity: O(N), where N = size of the given array.
Space Complexity: O(1) as we are not using any extra space.

9.	Maximum Consecutive Ones	[Easy]
	Given a binary array nums, return *the maximum number of consecutive* 1*'s in the array*. Example 1: Input: nums = [1,1,0,1,1,1] Output: 3 Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3. Example 2: Input: nums = [1,0,1,1,0,1] Output: 2 Constraints: • 1 <= nums.length <= 10^5 • nums[i] is either 0 or 1.	

```
import java.util.*;
public class Main {
   static int findMaxConsecutiveOnes(int nums[]) {
      int cnt = 0;
      int maxi = 0;
      for (int i = 0; i < nums.length; i++) {
         if (nums[i] == 1) {
            cnt++;
         } else {
            cnt = 0;
         }

         maxi = Math.max(maxi, cnt);
      }
      return maxi;
   }
   public static void main(String args[]) {
      int nums[] = { 1, 1, 0, 1, 1, 1 };
      int ans = findMaxConsecutiveOnes(nums);
      System.out.println("The maximum  consecutive 1's are " + ans);
   }
}
```

Time Complexity: O(N) since the solution involves only a single pass.
Space Complexity: O(1) because no extra space is used.

10.	Find the number that appears ones and other numbers twice	[Easy]
	Given a non-empty array of integers nums, every element appears *twice* except for one. Find that single one. You must implement a solution with a linear runtime complexity and use only constant extra space. Example 1: Input: nums = [2,2,1] Output: 1 Example 2: Input: nums = [4,1,2,1,2] Output: 4 Example 3: Input: nums = [1] Output: 1 Constraints: • $1 <= nums.length <= 3 * 10^4$ • $-3 * 10^4 <= nums[i] <= 3 * 10^4$ • Each element in the array appears twice except for one element which appears only once.	

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

public class Main {
   public static int findSingle(int[] arr) {
      Map<Integer, Integer> freq = new HashMap<>();

      // Store the frequency of each element
      for (int num : arr) {
         freq.put(num, freq.getOrDefault(num, 0) + 1);
      }

      // Find and return the element that appears only once
      for (Map.Entry<Integer, Integer> entry : freq.entrySet()) {
         if (entry.getValue() == 1) {
            return entry.getKey();
         }
```

```
        }

        // If no single element is found, return -1
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = { 2, 3, 5, 4, 5, 3, 4 };
        System.out.println(findSingle(arr));
    }
}
```

Time Complexity: O(N)
Space Complexity: O(N)

11.	Longest Subarray with given Sum K(Positives)	[Medium]

You are given an array *'a'* of size *'n'* and an integer *'k'*.

Find the length of the longest subarray of 'a' whose sum is equal to 'k'.

Example :
Input: ‘n’ = 7 ‘k’ = 3
‘a’ = [1, 2, 3, 1, 1, 1, 1]

Output: 3

Explanation: Subarrays whose sum = ‘3’ are:

[1, 2], [3], [1, 1, 1] and [1, 1, 1]

Here, the length of the longest subarray is 3, which is our final answer.
Detailed explanation (Input/output format, Notes, Images)
Sample Input 1 :
7 3
1 2 3 1 1 1 1

Sample Output 1 :
3

Explanation Of Sample Input 1 :
Subarrays whose sum = ‘3’ are:

[1, 2], [3], [1, 1, 1] and [1, 1, 1]
Here, the length of the longest subarray is 3, which is our final answer.

Sample Input 2 :
4 2
1 2 1 3

Sample Output 2 :
1

Sample Input 3 :
5 2
2 2 4 1 2

Sample Output 3 :
1

Expected time complexity :
The expected time complexity is O(n).

Constraints :
1 <= 'n' <= 5 * 10 ^ 6
1 <= 'k' <= 10^18
0 <= 'a[i]' <= 10 ^ 9

Time Limit: 1-second

```
import java.util.*;

public class Demo {
   public static int getLongestSubarray(int []a, long k) {
      int n = a.length; // size of the array.

      int len = 0;
      for (int i = 0; i < n; i++) { // starting index
         long s = 0; // Sum variable
         for (int j = i; j < n; j++) { // ending index
            // add the current element to
            // the subarray a[i...j-1]:
```

```
            s += a[j];

            if (s == k)
                len = Math.max(len, j - i + 1);
        }
    }
    return len;
}

public static void main(String[] args) {
    int[] a = {2, 3, 5, 1, 9};
    long k = 10;
    int len = getLongestSubarray(a, k);
    System.out.println("The length of the longest subarray is: " + len);
}
}
```

Time Complexity: $O(N^2)$ approx., where N = size of the array.
Space Complexity: O(1) as we are not using any extra space.

12.	Merge 2 Sorted Array	[Medium]

Given two sorted arrays, *'a'* and *'b'*, of size *'n'* and *'m'*, respectively, return the union of the arrays.

The union of two sorted arrays can be defined as an array consisting of the common and the distinct elements of the two arrays. The final array should be sorted in ascending order.

Note: 'a' and 'b' may contain duplicate elements, but the union array must contain unique elements.

Example:
Input: 'n' = 5 'm' = 3
'a' = [1, 2, 3, 4, 6]
'b' = [2, 3, 5]

Output: [1, 2, 3, 4, 5, 6]

Explanation: Common elements in 'a' and 'b' are: [2, 3]
Distinct elements in 'a' are: [1, 4, 6]
Distinct elements in 'b' are: [5]

	Union of ‘a’ and ‘b’ is: [1, 2, 3, 4, 5, 6] Detailed explanation (Input/output format, Notes, Images) Sample Input 1 : 5 3 1 2 3 4 6 2 3 5 Sample Output 1 : 1 2 3 4 5 6 Explanation Of Sample Input 1 : Input: ‘n’ = 5 ‘m’ = 3 ‘a’ = [1, 2, 3, 4, 6] ‘b’ = [2, 3, 5] Output: [1, 2, 3, 4, 5, 6] Explanation: Common elements in ‘a’ and ‘b’ are: [2, 3] Distinct elements in ‘a’ are: [1, 4, 6] Distinct elements in ‘b’ are: [5] Union of ‘a’ and ‘b’ is: [1, 2, 3, 4, 5, 6] Sample Input 2: 4 3 1 2 3 3 2 2 4 Sample Output 2: 1 2 3 4 Explanation Of Sample Input 2 : Input: ‘n’ = 5 ‘m’ = 3 ‘a’ = [1, 2, 3, 3] ‘b’ = [2, 2, 4] Output: [1, 2, 3, 4] Explanation: Common elements in ‘a’ and ‘b’ are: [2] Distinct elements in ‘a’ are: [1, 3] Distinct elements in ‘b’ are: [4] Union of ‘a’ and ‘b’ is: [1, 2, 3, 4] Expected Time Complexity: O((N + M)), where 'N' and ‘M’ are the sizes of Array ‘A’ and ‘B’. Constraints : 1 <= 'n', 'm' <= 10^5 -10^9 <= 'a'[i], 'b'[i] <= 10^9

	Time Limit: 1 sec

```
import java.util.*;

class Demo{
static ArrayList<Integer> FindUnion(int arr1[], int arr2[], int n, int m) {
  HashSet <Integer> s=new HashSet<>();
  ArrayList < Integer > Union=new ArrayList<>();
  for (int i = 0; i < n; i++)
    s.add(arr1[i]);
  for (int i = 0; i < m; i++)
    s.add(arr2[i]);
  for (int it: s)
    Union.add(it);
  return Union;
}
public static void main(String args[]) {
  int n = 10, m = 7;
  int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int arr2[] = {2, 3, 4, 4, 5, 11, 12};
  ArrayList<Integer> Union = FindUnion(arr1, arr2, n, m);
  System.out.println("Union of arr1 and arr2 is ");
  for (int val: Union)
    System.out.print(val+" ");
}
}
```

Time Compleixty : O((m+n)log(m+n)) . Inserting an element in a set takes logN time, where N is no of elements in the set. At max set can store m+n elements **{when there are no common elements and elements in arr,arr2 are distntict}.** So Inserting m+n th element takes log(m+n) time. Upon approximation across inserting all elements in worst, it would take O((m+n)log(m+n) time.
Space Complexity : O(m+n) {If Space of Union ArrayList is considered}

13.	Rotate Array by k elements	[Easy]
	Given an integer array nums, rotate the array to the right by k steps, where k is non-negative. Example 1: Input: nums = [1,2,3,4,5,6,7], k = 3 Output: [5,6,7,1,2,3,4]	

Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
Example 2:
Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]

Constraints:

- $1 <= nums.length <= 10^5$
- $-2^{31} <= nums[i] <= 2^{31} - 1$
- $0 <= k <= 10^5$

Follow up:

- Try to come up with as many solutions as you can. There are at least three different ways to solve this problem.
- Could you do it in-place with O(1) extra space?

```
import java.util.*;
public class Main {
  public static void swap(int[] arr, int a, int b, int k) {
    for (int i = 0; i < k; i++) {
      int temp = arr[a + i];
      arr[a + i] = arr[b + i];
      arr[b + i] = temp;
    }
  }
  public static void BlockSwap(int[] arr, int i, int k, int n) {
    if (k == 0 || k == n)
      return;
    // If first part and second part are of same size
    if (k == n - k) {
      swap(arr, i, n - k + i, k);
      return;
    }
    // If first part of array is of smaller size
    else if (k < n - k) {
      swap(arr, i, n - k + i, k);
      BlockSwap(arr, i, k, n - k); // second part of array is taken now
```

```
    }
    // If second part of array is of smaller size
    else {
      swap(arr, i, k, n - k);
      BlockSwap(arr, n - k + i, 2 * k - n, k);
    }
  }
  public static void main(String args[]) {
    int[] arr = {1,2,3,4,5,6,7};
    int n = 7;
    int k = 2;
    BlockSwap(arr, 0, k, n);
    System.out.println("After Rotating the array ");
    for (int i = 0; i < n; i++)
      System.out.print(arr[i] + " ");
  }
}
```

Time Complexity: O(n)
Space Complexity: O(1)

14.	Remove Duplicates from Sorted Array	[Easy]

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return *the number of unique elements in* nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

- Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
- Return k.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
   assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

Example 1:
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
Example 2:
Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).

Constraints:

- $1 <= nums.length <= 3 * 10^4$
- $-100 <= nums[i] <= 100$
- nums is sorted in non-decreasing order.

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
     int arr[] = {1,1,2,2,2,3,3};
      int k = removeDuplicates(arr);
      System.out.println("The array after removing duplicate elements is ");
      for (int i = 0; i < k; i++) {
         System.out.print(arr[i] + " ");
      }
   }
   static int removeDuplicates(int[] arr) {
      int i = 0;
      for (int j = 1; j < arr.length; j++) {
         if (arr[i] != arr[j]) {
            i++;
            arr[i] = arr[j];
         }
      }
      return i + 1;
```

```
    }
}
```

Time Complexity: O(N)
Space Complexity: O(1)

15.	Rearrange Array Elements by Sign	[Medium]

You are given a 0-indexed integer array nums of even length consisting of an equal number of positive and negative integers.
You should return the array of nums such that the the array follows the given conditions:

1. Every consecutive pair of integers have opposite signs.
2. For all integers with the same sign, the order in which they were present in nums is preserved.
3. The rearranged array begins with a positive integer.

Return *the modified array after rearranging the elements to satisfy the aforementioned conditions*.

Example 1:
Input: nums = [3,1,-2,-5,2,-4]
Output: [3,-2,1,-5,2,-4]
Explanation:
The positive integers in nums are [3,1,2]. The negative integers are [-2,-5,-4].
The only possible way to rearrange them such that they satisfy all conditions is [3,-2,1,-5,2,-4].
Other ways such as [1,-2,2,-5,3,-4], [3,1,2,-2,-5,-4], [-2,3,-5,1,-4,2] are incorrect because they do not satisfy one or more conditions.
Example 2:
Input: nums = [-1,1]
Output: [1,-1]
Explanation:
1 is the only positive integer and -1 the only negative integer in nums.
So nums is rearranged to [1,-1].

Constraints:

- $2 <= nums.length <= 2 * 10^5$
- nums.length is even
- $1 <= |nums[i]| <= 10^5$
- nums consists of equal number of positive and negative integers.

It is not required to do the modifications in-place.

```
import java.util.*;
public class Main {
```

```
    public static void main(String[] args) {
        // Array Initialization.
        ArrayList<Integer> A = new ArrayList<>(Arrays.asList(1, 2, -4, -5));
        ArrayList<Integer> ans = RearrangebySign(A);

        for (int i = 0; i < ans.size(); i++) {
            System.out.print(ans.get(i) + " ");
        }
    }

    public static ArrayList<Integer> RearrangebySign(ArrayList<Integer> A) {
        int n = A.size();

        // Define array for storing the ans separately.
        ArrayList<Integer> ans = new ArrayList<>(Collections.nCopies(n, 0));

        // positive elements start from 0 and negative from 1.
        int posIndex = 0, negIndex = 1;
        for (int i = 0; i < n; i++) {

            // Fill negative elements in odd indices and inc by 2.
            if (A.get(i) < 0) {
                ans.set(negIndex, A.get(i));
                negIndex += 2;
            }

            // Fill positive elements in even indices and inc by 2.
            else {
                ans.set(posIndex, A.get(i));
                posIndex += 2;
            }
        }

        return ans;
    }
}
```

Time Complexity: O(N) { O(N) for traversing the array once and substituting positives and negatives simultaneously using pointers, where N = size of the array A}.
Space Complexity: O(N) { Extra Space used to store the rearranged elements separately in an array, where N = size of array A}.

16.	Best Time to Buy and Sell Stock	[Easy]

	You are given an array prices where prices[i] is the price of a given stock on the i^{th} day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0. Example 1: Input: prices = [7,1,5,3,6,4] Output: 5 Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell. Example 2: Input: prices = [7,6,4,3,1] Output: 0 Explanation: In this case, no transactions are done and the max profit = 0. Constraints: • $1 <= prices.length <= 10^5$ • $0 <= prices[i] <= 10^4$

```
import java.util.*;

public class Main {

   public static void main(String[] args) {
   int arr[] = {7,1,5,3,6,4};

   int maxPro = maxProfit(arr);
   System.out.println("Max profit is: " + maxPro);

   }
   static int maxProfit(int[] arr) {
   int maxPro = 0;
   int minPrice = Integer.MAX_VALUE;
   for (int i = 0; i < arr.length; i++) {
      minPrice = Math.min(minPrice, arr[i]);
      maxPro = Math.max(maxPro, arr[i] - minPrice);
   }
```

```
    return maxPro;
    }
}
```

Time complexity: O(n)
Space Complexity: O(1)

17.	3 Sum	[Medium]

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.
Notice that the solution set must not contain duplicate triplets.

Example 1:
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.
Example 2:
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.
Example 3:
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.

Constraints:

- 3 <= nums.length <= 3000
- -10^5 <= nums[i] <= 10^5

```
import java.util.*;

public class Demo{
    public static List<List<Integer>> triplet(int n, int[] arr) {
        List<List<Integer>> ans = new ArrayList<>();
```

```
    Arrays.sort(arr);

    for (int i = 0; i < n; i++) {
        //remove duplicates:
        if (i != 0 && arr[i] == arr[i - 1]) continue;

        //moving 2 pointers:
        int j = i + 1;
        int k = n - 1;
        while (j < k) {
            int sum = arr[i] + arr[j] + arr[k];
            if (sum < 0) {
                j++;
            } else if (sum > 0) {
                k--;
            } else {
                List<Integer> temp = Arrays.asList(arr[i], arr[j], arr[k]);
                ans.add(temp);
                j++;
                k--;
                //skip the duplicates:
                while (j < k && arr[j] == arr[j - 1]) j++;
                while (j < k && arr[k] == arr[k + 1]) k--;
            }
        }
    }

    return ans;
}

public static void main(String[] args) {
    int[] arr = { -1, 0, 1, 2, -1, -4};
    int n = arr.length;
    List<List<Integer>> ans = triplet(n, arr);
    for (List<Integer> it : ans) {
        System.out.print("[");
        for (Integer i : it) {
            System.out.print(i + " ");
        }
        System.out.print("] ");
    }
    System.out.println();
```

```
    }
}
```

Time Complexity: O(NlogN)+O(N^2), where N = size of the array.
Space Complexity: O(no. of quadruplets), ***This space is only used to store the answer. We are not using any extra space to solve this problem.*** So, from that perspective, space complexity can be written as O(1).

18.	4 Sum	[Medium]
	Given an array nums of n integers, return *an array of all the unique quadruplets* [nums[a], nums[b], nums[c], nums[d]] such that: • 0 <= a, b, c, d < n • a, b, c, and d are distinct. • nums[a] + nums[b] + nums[c] + nums[d] == target You may return the answer in any order. Example 1: Input: nums = [1,0,-1,0,-2,2], target = 0 Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]] Example 2: Input: nums = [2,2,2,2,2], target = 8 Output: [[2,2,2,2]] Constraints: • 1 <= nums.length <= 200 • -10^9 <= nums[i] <= 10^9 • -10^9 <= target <= 10^9	

```
import java.util.*;

public class Demo {
    public static List<List<Integer>> fourSum(int[] nums, int target) {
        int n = nums.length; // size of the array
        List<List<Integer>> ans = new ArrayList<>();

        // sort the given array:
        Arrays.sort(nums);

        // calculating the quadruplets:
```

```
        for (int i = 0; i < n; i++) {
            // avoid the duplicates while moving i:
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            for (int j = i + 1; j < n; j++) {
                // avoid the duplicates while moving j:
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;

                // 2 pointers:
                int k = j + 1;
                int l = n - 1;
                while (k < l) {
                    long sum = nums[i];
                    sum += nums[j];
                    sum += nums[k];
                    sum += nums[l];
                    if (sum == target) {
                        List<Integer> temp = new ArrayList<>();
                        temp.add(nums[i]);
                        temp.add(nums[j]);
                        temp.add(nums[k]);
                        temp.add(nums[l]);
                        ans.add(temp);
                        k++;
                        l--;

                        // skip the duplicates:
                        while (k < l && nums[k] == nums[k - 1]) k++;
                        while (k < l && nums[l] == nums[l + 1]) l--;
                    } else if (sum < target) k++;
                    else l--;
                }
            }
        }

        return ans;
    }

    public static void main(String[] args) {
        int[] nums = {4, 3, 3, 4, 4, 2, 1, 2, 1, 1};
        int target = 9;
        List<List<Integer>> ans = fourSum(nums, target);
        System.out.println("The quadruplets are: ");
```

```
        for (List<Integer> it : ans) {
            System.out.print("[");
            for (int ele : it) {
                System.out.print(ele + " ");
            }
            System.out.print("] ");
        }
        System.out.println();
    }
}
```

Time Complexity: O(N^3), where N = size of the array.
Space Complexity: O(no. of quadruplets), ***This space is only used to store the answer. We are not using any extra space to solve this problem.*** So, from that perspective, space complexity can be written as O(1).

19.	Longest subarray with 0 sum	[Medium]
	given an array *'Arr'* of size *'N'*. You have to help him find the longest subarray of 'Arr', whose sum is 0. You must return the length of the longest subarray whose sum is 0. For Example: For N = 5, and Arr = {1, -1, 0, 0, 1}, We have the following subarrays with zero sums: {{1, -1}, {1, -1, 0}, {1, -1, 0, 0}, {-1, 0, 0, 1}, {0}, {0, 0}, {0}} Among these subarrays, {1, -1, 0, 0} and {-1, 0, 0, 1} are the longest subarrays with their sum equal to zero. Hence the answer is 4. Detailed explanation (Input/output format, Notes, Images) Sample Input 1: 4 1 0 -1 1 Sample Output 1: 3 Explanation of Sample Input 1: The subarrays with sums equal to zero are: {{1, 0, -1}, {0}, {0, -1, 1}, {-1, 1}}. Among these, {1, 0, -1} and {0, -1, 1} are the longest with length equal to 3. Hence the answer is 3. Sample Input 2: 2 1 1 Sample Output 2: 0 Constraints:	

	1 <= N <= 10^5 -10^9 <= Arr[i] <= 10^9 The sum of 'N' over all test cases is less than or equal to 10^5. Time Limit: 1 sec.

```
int maxLen(int A[], int n)
   {
      // Your code here
      HashMap<Integer, Integer> mpp = new HashMap<Integer, Integer>();

      int maxi = 0;
      int sum = 0;

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

         sum += A[i];

         if(sum == 0) {
            maxi = i + 1;
         }
         else {
            if(mpp.get(sum) != null) {

               maxi = Math.max(maxi, i - mpp.get(sum));
            }
            else {

               mpp.put(sum, i);
            }
         }
      }
      return maxi;
   }
```

Time Complexity: O(N), as we are traversing the array only once
Space Complexity: O(N), in the worst case we would insert all array elements prefix sum into our hashmap

20.	Count the number of subarrays with given xor K	[Medium]

Given an array *'A'* consisting of *'N'* integers and an integer *'B'*, find the number of subarrays of array 'A' whose bitwise XOR($\oplus$) of all elements is equal to 'B'.

A subarray of an array is obtained by removing some(zero or more) elements from the front and back of the array.

Example:
Input: 'N' = 4 'B' = 2
'A' = [1, 2, 3, 2]

Output: 3

Explanation: Subarrays have bitwise xor equal to '2' are: [1, 2, 3, 2], [2], [2].
Detailed explanation (Input/output format, Notes, Images)
Sample Input 1:
4 2
1 2 3 2
Sample Output 1 :
3
Explanation Of Sample Input 1:
Input: 'N' = 4 'B' = 2
'A' = [1, 2, 3, 2]

Output: 3

Explanation: Subarrays have bitwise xor equal to '2' are: [1, 2, 3, 2], [2], [2].
Sample Input 2:
4 3
1 2 3 3
Sample Output 2:
4
Sample Input 3:
5 6
1 3 3 3 5
Sample Output 3:
2
Constraints:
1 <= N <= 10^3
1 <= A[i], B <= 10^9

Time Limit: 1-sec

```
import java.util.*;

public class Demo{

   public static int subarraysWithXorK(int []a, int k) {
      int n = a.length; //size of the given array.
      int xr = 0;
      Map<Integer, Integer> mpp = new HashMap<>(); //declaring the map.
      mpp.put(xr, 1); //setting the value of 0.
      int cnt = 0;

      for (int i = 0; i < n; i++) {
         // prefix XOR till index i:
         xr = xr ^ a[i];

         //By formula: x = xr^k:
         int x = xr ^ k;

         // add the occurrence of xr^k
         // to the count:
         if (mpp.containsKey(x)) {
            cnt += mpp.get(x);
         }

         // Insert the prefix xor till index i
         // into the map:
         if (mpp.containsKey(xr)) {
            mpp.put(xr, mpp.get(xr) + 1);
         } else {
            mpp.put(xr, 1);
         }
      }
      return cnt;
   }

   public static void main(String[] args) {
      int[] a = {4, 2, 2, 6, 4};
      int k = 6;
```

```
        int ans = subarraysWithXorK(a, k);
        System.out.println("The number of subarrays with XOR k is: " + ans);
    }
}
```

Time Complexity: O(N) or O(N*logN) depending on which map data structure we are using, where N = size of the array.

Space Complexity: O(N) as we are using a map data structure.