💡 **Question 1** Given three integer arrays arr1, arr2 and arr3 **sorted** in **strictly increasing** order, return a sorted array of **only** the integers that appeared in **all** three arrays.

**Example 1:**

Input: arr1 = [1,2,3,4,5], arr2 = [1,2,5,7,9], arr3 = [1,3,4,5,8]

Output: [1,5]

**Explanation:** Only 1 and 5 appeared in the three arrays.

**CODE :**

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


public class Main {
   public static void main(String[] args) {
      int[] array1 = {1, 2, 4, 5, 6};
      int[] array2 = {2, 3, 5, 7};

      Set<Integer> set1 = new HashSet<>();
      for (int num : array1) {
         set1.add(num);
      }

      Set<Integer> interSec = new HashSet<>();
      for (int num : array2) {
         if (set1.contains(num)) {
            interSec.add(num);
         }
      }

      System.out.println("interSec: " + interSec);
   }
}
```

**Question 2**

**Given two 0-indexed integer arrays nums1 and nums2, return *a list* answer *of size* 2 *where:***

- **answer[0] *is a list of all distinct integers in* nums1 *which are not present in* nums2*.***
- **answer[1] *is a list of all distinct integers in* nums2 *which are not present in* nums1.**

**Note that the integers in the lists may be returned in any order.**

**Example 1:**

**Input: nums1 = [1,2,3], nums2 = [2,4,6]**

**Output: [[1,3],[4,6]]**

**Explanation:**

**For nums1, nums1[1] = 2 is present at index 0 of nums2, whereas nums1[0] = 1 and nums1[2] = 3 are not present in nums2. Therefore, answer[0] = [1,3].**

**For nums2, nums2[0] = 2 is present at index 1 of nums1, whereas nums2[1] = 4 and nums2[2] = 6 are not present in nums2. Therefore, answer[1] = [4,6].**

**Code ::**

```
import java.util.*;
public class DistinctArrays {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      System.out.println("Enter the size of the first array:");
      int size1 = scanner.nextInt();
      int[] nums1 = new int[size1];
      System.out.println("Enter the elements of the first array:");
      for (int i = 0; i < size1; i++) {
         nums1[i] = scanner.nextInt();
      }

      System.out.println("Enter the size of the second array:");
      int size2 = scanner.nextInt();
      int[] nums2 = new int[size2];
      System.out.println("Enter the elements of the second array:");
      for (int i = 0; i < size2; i++) {
```

```
        nums2[i] = scanner.nextInt();
    }

    List<List<Integer>> answer = findDistinctArrays(nums1, nums2);
    System.out.println("Distinct integers in nums1 not present in nums2: " + answer.get(0));
    System.out.println("Distinct integers in nums2 not present in nums1: " + answer.get(1));
}

public static List<List<Integer>> findDistinctArrays(int[] nums1, int[] nums2) {
    List<Integer> distinctNums1 = new ArrayList<>();
    List<Integer> distinctNums2 = new ArrayList<>();

    Set<Integer> set = new HashSet<>();
    for (int num : nums2) {
        set.add(num);
    }

    for (int num : nums1) {
        if (!set.contains(num)) {
            distinctNums1.add(num);
        }
    }

    set.clear();
    for (int num : nums1) {
        set.add(num);
    }

    for (int num : nums2) {
        if (!set.contains(num)) {
            distinctNums2.add(num);
```

```
        }
      }

      List<List<Integer>> answer = new ArrayList<>();
      answer.add(distinctNums1);
      answer.add(distinctNums2);
      return answer;
    }
}
```

**Question 3** A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container.

If such an arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

● For example, the next permutation of arr = [1,2,3] is [1,3,2]. ● Similarly, the next permutation of arr = [2,3,1] is [3,1,2]. ● While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums. The replacement must be in place and use only constant extra memory.

**Example 1:** Input: nums = [1,2,3] Output: [1,3,2]

**Code** :

```
import java.util.Arrays;

import java.util.Scanner;



public class NextPermutation {
```

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter the size of the array:");
    int size = scanner.nextInt();
    int[] nums = new int[size];
    System.out.println("Enter the elements of the array:");
    for (int i = 0; i < size; i++) {
        nums[i] = scanner.nextInt();
    }

    nextPermutation(nums);
    System.out.println("Next permutation: " + Arrays.toString(nums));
}

public static void nextPermutation(int[] nums) {
        int i = nums.length - 2;
    while (i >= 0 && nums[i] >= nums[i + 1]) {
        i--;
    }
        int j = nums.length - 1;
        while (j >= 0 && nums[j] <= nums[i]) {
            j--;
        }
        swap(nums, i, j);
```

```
        }


            reverse(nums, i + 1);
    }


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


    private static void reverse(int[] nums, int start) {
        int i = start;
        int j = nums.length - 1;
        while (i < j) {
            swap(nums, i, j);
            i++;
            j--;
        }
    }
}
```

**Question 4** Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You must write an algorithm with O(log n) runtime complexity. **Example 1:** Input: nums = [1,3,5,6], target = 5 Output: 2

**CODE :**

```
import java.util.Scanner;
```

```
public class SERPosition {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      System.out.println("Enter the size of the array:");
      int size = scanner.nextInt();
      int[] nums = new int[size];
      System.out.println("Enter the elements of the :");
      for (int i = 0; i < size; i++) {
         nums[i] = scanner.nextInt();
      }

      System.out.println("Enter the target:");
      int target = scanner.nextInt();

      int index = SER(nums, target);
      System.out.println("Index where the target : " + index);
   }

   public static int SER(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 left;
  }
}
```

**Question 5** You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

**Example 1:** Input: digits = [1,2,3] Output: [1,2,4]

**Explanation:** The array represents the integer 123. Incrementing by one gives 123 + 1 = 124. Thus, the result should be [1,2,4].

Code :

```
import java.util.Arrays;

public class PlusOne {

    public static void main(String[] args) {

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

        int[] result = plusOne(digits);

        System.out.println("Result: " + Arrays.toString(result));

    }

 public static int[] plusOne(int[] digits) {

        int n = digits.length;




        for (int i = n - 1; i >= 0; i--) {
```

```
            if (digits[i] < 9) {
                digits[i]++;
                return digits;
            }



            digits[i] = 0;
        }



        int[] newDigits = new int[n + 1];
        newDigits[0] = 1;
        return newDigits;
    }
}
```

**Question 6** 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

Code:

```
import java.util.Arrays;

public class SingleNumber {
    public static void main(String[] args) {
        int[] nums = {2, 2, 1};
        int result = singleNumber(nums);
```

```
        System.out.println("Single number: " + result);
    }

    public static int singleNumber(int[] nums) {
        int result = 0;
        for (int num : nums) {
            result ^= num;
        }
        return result;
    }
}
```

**Question 7** You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within the inclusive range. A number x is considered missing if x is in the range [lower, upper] and x is not in nums. Return the shortest sorted list of ranges that exactly covers all the missing numbers. That is, no element of nums is included in any of the ranges, and each missing number is covered by one of the ranges.

**Example 1:** Input: nums = [0,1,3,50,75], lower = 0, upper = 99 Output: [[2,2],[4,49],[51,74],[76,99]]

**Explanation:** The ranges are: [2,2] [4,49] [51,74] [76,99]

Code:

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

public class MissingRanges {
    public static void main(String[] args) {
        int[] nums = {0, 1, 3, 50, 75};
        int lower = 0;
```

```
    int upper = 99;
    List<String> result = findRnage(nums, lower, upper);
    System.out.println("Missing ranges: " + result);
}

public static List<String> findRnage(int[] nums, int lower, int upper) {
    List<String> result = new ArrayList<>();

    // Find the missing ranges
    int prev = lower - 1;
    for (int i = 0; i <= nums.length; i++) {
        int curr = (i < nums.length) ? nums[i] : upper + 1;

        if (curr - prev > 1) {
            result.add(getRange(prev + 1, curr - 1));
        }

        prev = curr;
    }

    return result;
}

private static String getRange(int start, int end) {
    if (start == end) {
```

```
            return String.valueOf(start);
        } else {
            return start + "->" + end;
        }
    }
}
```

**Question 8** Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

**Example 1:** Input: intervals = [[0,30],[5,10],[15,20]] Output: false

```
import java.util.Arrays;

public class MeetingRooms {
    public static void main(String[] args) {
        int[][] intervals = {{0, 30}, {5, 10}, {15, 20}};
        boolean attend = canMeet(intervals);
        System.out.println("Can attend all meetings: " + attend);
    }

    public static boolean canMeet(int[][] intervals) {

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


        for (int i = 1; i < intervals.length; i++) {
```

```
            int[] prevIn = intervals[i - 1];
            int[] currInt = intervals[i];

            if (prevIn[1] > currInt[0]) {
                return false;
            }
        }

        return true;
    }
}
```