💡 **Q1.** 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: Input: nums = [2,7,11,15], target = 9 Output0 [0,1]

Explanation: Because nums[0] + nums[1] == 9, we return [0, 1][

```
public class TwoSum {

    public int[] twoSum(int[] nums, int target) {

        int[] result = new int[2];


        // Iterate through each element in the array

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

            // Iterate through the remaining elements to find a complement

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

                if (nums[i] + nums[j] == target) {

                    // Found the two numbers that add up to the target

                    result[0] = i;

                    result[1] = j;

                    return result;

                }

            }

        }


        // No solution found, return an empty array

        return new int[]{};
    }


    public static void main(String[] args) {
        TwoSum solution = new TwoSum();
        int[] nums = {2, 7, 11, 15};
        int target = 9;
        int[] result = solution.twoSum(nums, target);
        System.out.println("[" + result[0] + ", " + result[1] + "]");
    }
}
```

Q.2 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]]

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


public class FourSum {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();


        Arrays.sort(nums);
        int n = nums.length;


        for (int i = 0; i < n - 3; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) {
```

```
    continue;
}

for (int j = i + 1; j < n - 2; j++) {
    if (j > i + 1 && nums[j] == nums[j - 1]) {
        continue;
    }

    int left = j + 1;
    int right = n - 1;

    while (left < right) {
        int sum = nums[i] + nums[j] + nums[left] + nums[right];

        if (sum == target) {
            result.add(Arrays.asList(nums[i], nums[j], nums[left], nums[right]));

            while (left < right && nums[left] == nums[left + 1]) {
                left++;
            }
            while (left < right && nums[right] == nums[right - 1]) {
                right--;
            }

            left++;
            right--;
```

```
                    } else if (sum < target) {
                        left++;
                    } else {
                        right--;
                    }
                }
            }
        }

        return result;
    }

    public static void main(String[] args) {
        FourSum solution = new FourSum();
        int[] nums = {1, 0, -1, 0, -2, 2};
        int target = 0;
        List<List<Integer>> result = solution.fourSum(nums, target);
        System.out.println(result);
    }
}
```

💡 **Q.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]

```
public class NextPermutation {

    public void nextPermutation(int[] nums) {

        // Find the first decreasing element from the right

        int i = nums.length - 2;

        while (i >= 0 && nums[i] >= nums[i + 1]) {

            i--;

        }



        // If a decreasing element is found

        if (i >= 0) {

            // Find the smallest element larger than nums[i] towards the right

            int j = nums.length - 1;

            while (j > i && nums[j] <= nums[i]) {

                j--;
        }

        // Swap nums[i] and nums[j]
        swap(nums, i, j);
    }

    // Reverse the elements after index i
    reverse(nums, i + 1);
}

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

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

```
    public static void main(String[] args) {

        NextPermutation solution = new NextPermutation();

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

        solution.nextPermutation(nums);

        for (int num : nums) {

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

        }

    }

}
```

Q.3 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

```
public class SearchInsertPosition {

    public int searchInsert(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;
            }
        }

        // At this point, the target value is not found in the array
        // The left pointer represents the index where it would be inserted
        return left;
    }

    public static void main(String[] args) {
        SearchInsertPosition solution = new SearchInsertPosition();
        int[] nums = {1, 3, 5, 6};
        int target = 5;
        int position = solution.searchInsert(nums, target);
        System.out.println(position);
    }
}
```

💡 **Q.4** 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].

```
public class PlusOne {

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

        int n = digits.length;


        // Iterate from right to left

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

            // Increment the current digit by 1

            digits[i]++;


            // If the digit becomes 10, set it to 0 and carry over the 1

            if (digits[i] == 10) {

                digits[i] = 0;

            } else {

                // If the digit is less than 10, no need to carry over, return the digits

                return digits;

            }

        }


        // If we reach here, it means all digits were 9s, so we need to add an extra digit

        int[] result = new int[n + 1];

        result[0] = 1;
        return result;

    }


    public static void main(String[] args) {

        PlusOne solution = new PlusOne();

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

        int[] result = solution.plusOne(digits);

        for (int digit : result) {

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

        }

    }

}
```

💡 **Q.5** You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in non-decreasing order.

The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Example 1: Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3 Output: [1,2,2,3,5,6]

Explanation: The arrays we are merging are [1,2,3] and [2,5,6]. The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

```
public class MergeSortedArray {

    public void merge(int[] nums1, int m, int[] nums2, int n) {

        int i = m - 1; // Index of last element in nums1

        int j = n - 1; // Index of last element in nums2

        int k = m + n - 1; // Index of last position in nums1 to merge elements
        // Merge elements from the end, starting from the largest elements
        while (j >= 0) {
            if (i >= 0 && nums1[i] > nums2[j]) {
                nums1[k] = nums1[i];
                i--;
            } else {
                nums1[k] = nums2[j];
                j--;
            }
            k--;
        }
    }

    public static void main(String[] args) {
        MergeSortedArray solution = new MergeSortedArray();
        int[] nums1 = {1, 2, 3, 0, 0, 0};
        int m = 3;
        int[] nums2 = {2, 5, 6};
        int n = 3;
        solution.merge(nums1, m, nums2, n);
        for (int num : nums1) {
            System.out.print(num + " ");
        }
    }
}
```

💡 **Q.6** Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

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

Output: true

```
import java.util.HashSet;

import java.util.Set;


public class ContainsDuplicate {

    public boolean containsDuplicate(int[] nums) {

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


        for (int num : nums) {

            if (set.contains(num)) {

                return true;

            }

            set.add(num);

        }


        return false;

    }


    public static void main(String[] args) {

        ContainsDuplicate solution = new ContainsDuplicate();

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

        boolean containsDuplicate = solution.containsDuplicate(nums);
        System.out.println(containsDuplicate);

    }

}
```

💡 **Q7.** Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the nonzero 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]

```
public class MoveZeros {

    public void moveZeroes(int[] nums) {

        int i = 0; // Index to track the position for the next non-zero element


        // Iterate through the array

        for (int num : nums) {

            if (num != 0) {

                nums[i] = num;

                i++;

            }

        }


        // Fill the remaining positions with zeros
        while (i < nums.length) {

            nums[i] = 0;

            i++;

        }

    }


    public static void main(String[] args) {

        MoveZeros solution = new MoveZeros();

        int[] nums = {0, 1, 0, 3, 12};

        solution.moveZeroes(nums);

        for (int num : nums) {

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

        }

    }

}
```

💡 **Q.8** You have a set of integers s, which originally contains all the numbers from 1 to n. Unfortunately, due to some error, one of the numbers in s got duplicated to another number in the set, which results in repetition of one number and loss of another number.

You are given an integer array nums representing the data status of this set after the error.

Find the number that occurs twice and the number that is missing and return them in the form of an array.

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

```
public class FindErrorNums {

    public int[] findErrorNums(int[] nums) {

        int[] result = new int[2];

        int n = nums.length;
```

```
// Calculate the sum of all numbers from 1 to n
int sum = (n * (n + 1)) / 2;

// Calculate the sum of all elements in nums
int actualSum = 0;
for (int num : nums) {
    actualSum += num;
}

// Calculate the difference between the expected sum and actual sum
int diff = actualSum - sum;

// Create a boolean array to keep track of visited elements
boolean[] visited = new boolean[n + 1];

// Find the duplicate number and missing number
for (int num : nums) {
    if (visited[num]) {
        result[0] = num; // Duplicate number
    } else {
        visited[num] = true;
    }
}

// Find the missing number
```

```
        for (int i = 1; i <= n; i++) {
            if (!visited[i]) {
                result[1] = i; // Missing number
                break;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        FindErrorNums solution = new FindErrorNums();
        int[] nums = {1, 2, 2, 4};
        int[] result = solution.findErrorNums(nums);
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```