💡 **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][

</aside>

```
public class Solution {
    public int[] twoSum(int[] numbers, int target) {
        HashMap<Integer,Integer> indexMap = new HashMap<Integer,Integer>();
        for(int i = 0; i < numbers.length; i++){
            Integer requiredNum = (Integer)(target - numbers[i]);
            if(indexMap.containsKey(requiredNum)){
                int toReturn[] = {indexMap.get(requiredNum), i};
                return toReturn;
            }

            indexMap.put(numbers[i], i);
        }
        return null;
    }
}
```

<aside>

💡 **Q2.** Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.

Consider the number of elements in nums which are not equal to val 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 elements which are not equal to val. The remaining elements of nums are not important as well as the size of nums.

- Return k.

Example :

Input: nums = [3,2,2,3], val = 3

Output: 2, nums = [2,2,_*,_*]

Explanation: Your function should return k = 2, with the first two elements of nums being 2. It does not matter what you leave beyond the returned k (hence they are underscores)[

</aside>

```
class Solution {
    public int removeElement(int[] nums, int val) {
        int count = 0;
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == val){
                continue;
            }
            nums[count] = nums[i];
            count++;
        }
        return count;
    }
}
```

<aside>

💡 **Q3.** 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

</aside>

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        // Last and First indexes
        int start = 0;
        int end = nums.length - 1;

         // Traverse an array
        while( start <= end ) {

            int mid = (start + end) / 2;

            //if target value found.
            if(nums[mid] == target) {
                return mid;
            }

            // If target value is greater then mid elements's value
            else if (target > nums[mid]) {
                start = mid + 1;
            }

            //otherwise target value is less,
             else {
```

```
            end = mid - 1;
          }
        }
        // Return the insertion position
        return end + 1;
    }
}
```

<aside>

💡 **Q4.** 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].

</aside>

```
class Solution {
    public int[] plusOne(int[] digits) {

        int i = digits.length - 1;

        while (i >=0 && digits[i] == 9) {
            i --;
        }

        if (i == -1) {
            int[] result = new int[digits.length + 1];
```

```
            result[0] = 1;
            return result;
        }


        int[] result = new int[digits.length];

        result[i] = digits[i] + 1;
        for (int j = 0; j < i; j ++) {
            result[j] = digits[j];
        }

        return result;

    }
}
```

<aside>

💡 **Q5.** 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.

</aside>

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        for (int i = m - 1, j = n - 1, k = m + n - 1; j >= 0; --k) {
            nums1[k] = i >= 0 && nums1[i] > nums2[j] ? nums1[i--] : nums2[j--];
        }
    }
}
```

<aside>

💡 **Q6.** 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

</aside>

```
class Solution {
    public boolean containsDuplicate(int[] nums) {

        HashSet<Integer> set= new HashSet<Integer>();
        for(int x :nums){
            if(set.contains(x)) return true;
            set.add(x);
        }
        return false;

    }
}
```

<aside>

💡 **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]

</aside>

```
class Solution {
    public void moveZeroes(int[] a) {

        int i=0,j=0;

        // t stores the index of the last element, which will be (number of elements - 1) as array index starts from 0.
        int t=a.length-1;
        while(i<=t)
        {
            if(a[i]!=0)
            {
                //copying only the non-zero elements in the original array itself as need to keep it in-place
                a[j++]=a[i];
            }
            i++;
        }
```

```
        // counting the number of zeroes, which will be equal to the total numbers - the non zero
numbers.
        j=t-j+1;



        //filling the remaining array with the zeros starting from end.
        while(j!=0)
        {
           a[t--]=0;
           j--;
        }
    }
}
```

<aside>

💡 **Q8.** 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]

</aside>

```
class Solution {
    public int[] findErrorNums(int[] nums) {
        int N = nums.length, sum = N * (N + 1) / 2;
        int[] ans = new int[2];
        boolean[] seen = new boolean[N+1];
        for (int num : nums) {
            sum -= num;
            if (seen[num]) ans[0] = num;
            seen[num] = true;
        }
        ans[1] = sum + ans[0];
        return ans;
    }
}
```