<aside>

💡 **Question 1**

Convert 1D Array Into 2D Array

You are given a **0-indexed** 1-dimensional (1D) integer array original, and two integers, m and n. You are tasked with creating a 2-dimensional (2D) array with m rows and n columns using **all** the elements from original.

The elements from indices 0 to n - 1 (**inclusive**) of original should form the first row of the constructed 2D array, the elements from indices n to 2 * n - 1 (**inclusive**) should form the second row of the constructed 2D array, and so on.

Return *an* m x n *2D array constructed according to the above procedure, or an empty 2D array if it is impossible*.

</aside>

```
class Solution {
  public int[][] construct2DArray(int[] original, int m, int n) {
    if (original.length != m * n)
      return new int[][] {};

    int[][] ans = new int[m][n];

    for (int i = 0; i < original.length; ++i)
      ans[i / n][i % n] = original[i];

    return ans;
  }
}
```

<aside>

💡 **Question 2**

You have n coins and you want to build a staircase with these coins. The staircase consists of k rows where the ith row has exactly i coins. The last row of the staircase **may be** incomplete.

Given the integer n, return *the number of **complete rows** of the staircase you will build*.

Example 1:

</aside>

```
class Solution {
public int arrangeCoins(int n) {

   long start=1;
   long sum=1;
   while( sum <= n){
      sum+= ++start;
   }

   return (int) start-1;

}
}
```

<aside>

💡 **Question 3**

Given an integer array nums sorted in **non-decreasing** order, return *an array of **the squares of each number** sorted in non-decreasing order*.

Example 1:

Input: nums = [-4,-1,0,3,10]

Output: [0,1,9,16,100]

Explanation: After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

</aside>

```
class Solution {
  public int[] sortedSquares(int[] A) {
    for (int i = 0; i < A.length; i++)
      A[i] = A[i] * A[i];
    Arrays.sort(A);
    return A;
  }
}
```

<aside>

💡 **Question 4**

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].

</aside>

```
class Solution {
   // Returns the elements in the first arg nums1 that don't exist in the second arg nums2.
   List<Integer> getElementsOnlyInFirstList(int[] nums1, int[] nums2) {
      Set<Integer> onlyInNums1 = new HashSet<> ();
```

```
        // Store nums2 elements in an unordered set.
        Set<Integer> existsInNums2 = new HashSet<> ();
        for (int num : nums2) {
            existsInNums2.add(num);
        }

        // Iterate over each element in the list nums1.
        for (int num : nums1) {
            if (!existsInNums2.contains(num)) {
                onlyInNums1.add(num);
            }
        }

        // Convert to vector.
        return new ArrayList<>(onlyInNums1);
    }

    public List<List<Integer>> findDifference(int[] nums1, int[] nums2) {
        return Arrays.asList(getElementsOnlyInFirstList(nums1, nums2),
getElementsOnlyInFirstList(nums2, nums1));
    }
}
```

<aside>

💡 **Question 5**

Given two integer arrays arr1 and arr2, and the integer d, *return the distance value between the two arrays*.

The distance value is defined as the number of elements arr1[i] such that there is not any element arr2[j] where |arr1[i]-arr2[j]| <= d.

Example 1:

Input: arr1 = [4,5,8], arr2 = [10,9,1,8], d = 2

Output: 2

Explanation:

For arr1[0]=4 we have:

|4-10|=6 > d=2

|4-9|=5 > d=2

|4-1|=3 > d=2

|4-8|=4 > d=2

For arr1[1]=5 we have:

|5-10|=5 > d=2

|5-9|=4 > d=2

|5-1|=4 > d=2

|5-8|=3 > d=2

For arr1[2]=8 we have:

|8-10|=2 <= d=2

|8-9|=1 <= d=2

|8-1|=7 > d=2

|8-8|=0 <= d=2

</aside>

```
class Solution {
    public int findTheDistanceValue(int[] arr1, int[] arr2, int d) {
        int count = 0;
        int length1 = arr1.length, length2 = arr2.length;
        for (int i = 0; i < length1; i++) {
            int num1 = arr1[i];
            boolean flag = true;
            for (int j = 0; j < length2; j++) {
                int num2 = arr2[j];
                if (Math.abs(num1 - num2) <= d) {
                    flag = false;
                    break;
                }
            }
            if (flag)
                count++;
        }
        return count;
    }
}
```

<aside>

💡 **Question 6**

Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears **once** or **twice**, return *an array of all the integers that appears **twice***.

You must write an algorithm that runs in O(n) time and uses only constant extra space.

Example 1:

Input: nums = [4,3,2,7,8,2,3,1]

Output:

[2,3]

</aside>

```
class Solution {
    public int findMin(int[] nums) {
        int n = nums.length;
        if (nums[0] <= nums[n - 1]) {
            return nums[0];
        }
        int left = 0, right = n - 1;
        while (left < right) {
            int mid = (left + right) >> 1;
            if (nums[0] <= nums[mid]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return nums[left];
    }
}
```

<aside>

💡 **Question 8**

An integer array original is transformed into a **doubled** array changed by appending **twice the value** of every element in original, and then randomly **shuffling** the resulting array.

Given an array changed, return original *if* changed *is a **doubled** array. If* changed *is not a **doubled** array, return an empty array. The elements in* original *may be returned in **any** order*.

Example 1:

Input: changed = [1,3,4,2,6,8]

Output: [1,3,4]

Explanation: One possible original array could be [1,3,4]:

- Twice the value of 1 is 1 * 2 = 2.
- Twice the value of 3 is 3 * 2 = 6.
- Twice the value of 4 is 4 * 2 = 8.

Other original arrays could be [4,3,1] or [3,1,4].

</aside>

```
class Solution {
  public int[] findOriginalArray(int[] changed) {
    List<Integer> ans = new ArrayList<>();
```

```
    Queue<Integer> q = new ArrayDeque<>();

    Arrays.sort(changed);

    for (final int num : changed)
      if (!q.isEmpty() && num == q.peek()) {
        q.poll();
      } else {
        q.offer(num * 2);
        ans.add(num);
      }

    return q.isEmpty() ? ans.stream().mapToInt(Integer::intValue).toArray() : new int[] {};
  }
}
```