<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) {
    int[][] res= new int[m][n];
    if(original.length != m*n) return new int[0][0];
    int k=0;
    for(int i= 0; i<m; i++){
      for(int j= 0; j< n; j++){
        res[i][j]= original[k++];
      }
    }
    return res;

  }
}
```

<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*.

</aside>

```
class Solution {
  public int arrangeCoins(int n) {
    int coin = n;
    for(int i=1;i<=n;i++){
      coin=coin-i;
      if(coin<0){
        return i-1;
      }
    }

    return 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[] nums) {
    int n= nums.length;
    int index= n-1;
    int l=0, r= index;
    int[] res= new int[n]; // 2pointer approach
    while(l<=r){
      if(Math.abs(nums[l])< Math.abs(nums[r])){
        res[index--]= nums[r]*nums[r];
        r--;
      } else{
        res[index--]= nums[l]*nums[l];
        l++;

      }
    }
    return res;
}
}
```

<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 {
  public List<List<Integer>> findDifference(int[] nums1, int[]
nums2) {
    List<List<Integer>> al= new ArrayList<>();
    HashSet<Integer> set1= new HashSet<>();
    HashSet<Integer> set2= new HashSet<>();
    List<Integer> list1= new ArrayList<>();
    List<Integer> list2= new ArrayList<>();
    for(int i: nums1) set1.add(i);

    for(int i: nums2) set2.add(i);

    for(int i: set1){
      if(set2.contains(i)) continue;
      else{
         list1.add(i);
      }

    }
    al.add(list1);
    for(int i: set2){

      if(set1.contains(i)) continue;
      else{

        list2.add(i);
      }
    }
    al.add(list2);
    return al;
  }
}
```

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) {
   Set<Integer> numSet = new HashSet<>();
    int count = 0;

    for (int num : arr2) {
      numSet.add(num);
    }

    for (int num : arr1) {
      boolean found = true;
      for (int num2 : numSet) {
        if (Math.abs(num - num2) <= d) {
          found = false;
          break;
        }
      }
      if (found) {
        count++;
      }
    }

    return count;

  }
}
```

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 List<Integer> findDuplicates(int[] nums) {
    HashSet<Integer> set= new HashSet<>();
    List<Integer> list= new ArrayList<>();

    for(int n: nums){
     if(!set.add(n)) list.add(n);
     set.add( n);
    }
    return list;
  }
}
```

Question 7

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [3,4,5,1,2]

Output: 1

Explanation:

The original array was [1,2,3,4,5] rotated 3 times.

</aside>

```
class Solution {
    public int findMin(int[] nums) {
        int l = 0;
        int r= nums.length-1;
        if(nums[l]<= nums[r]){
            //already sorted array
            return nums[0];        }
        while(l<=r){
            int mid = (l+r) /2;

            if(nums[mid]>nums[mid+1])
                return nums[mid+1];
            else if(nums[mid]< nums[mid-1])
                return nums[mid];
            else if(nums[l]<= nums[mid]){
                // left part is sorted means search in 
right part
                l = mid+1;
            }
            else{
                // right part is sorted
                 r= mid-1;
            }


        }
        return -1;


    }
}
```

<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) {
  if (changed.length % 2 != 0) {
      return new int[0]; // If the length is odd, it's not
possible to form a doubled ///array
    }

    int[] original = new int[changed.length / 2];
    Map<Integer, Integer> countMap = new HashMap<>();

    for (int num : changed) {
      countMap.put(num, countMap.getOrDefault(num, 0) + 1);
    }

    Arrays.sort(changed); // Sort the array in ascending order

    int index = 0;
    for (int num : changed) {
      if (countMap.getOrDefault(num, 0) <= 0) {
        continue;
      }
      int doubleNum = num * 2;
      if (countMap.getOrDefault(doubleNum, 0) <= 0) {
        return new int[0]; // If the doubled value doesn't
exist, it's not a doubled //array
      }

      original[index] = num;
      index++;
      countMap.put(num, countMap.get(num) - 1);
      countMap.put(doubleNum, countMap.get(doubleNum) - 1);
    }

    return original;
  }

}
```