Array's Problem

1.Reverse an Array

You are given an array of integers **arr[]**. Your task is to **reverse** the given array.

Note: Modify the array in place.

Examples:

Input: arr = [1, 4, 3, 2, 6, 5]
Output: [5, 6, 2, 3, 4, 1]
Explanation: The elements of the array are 1 4 3 2 6 5. After reversing the array, the first element goes to the last position, the second element goes to the second last position and so on. Hence, the answer is 5 6 2 3 4 1.

Input: arr = [4, 5, 2]
Output: [2, 5, 4]
Explanation: The elements of the array are 4 5 2. The reversed array will be 2 5 4.

Input: arr = [1]
Output: [1]
Explanation: The array has only single element, hence the reversed array is same as the original.

```java
class Solution {
    public void reverseArray(int arr[]) {
        // code here
        int n= arr.length;
        int start=0;
        int end=n-1;

        while(start<end){
            swap(arr,start,end);
            start++;
            end--;
        }
    }
    public void swap(int arr[],int start,int end){
        int temp=arr[start];
        arr[start]=arr[end];
        arr[end]=temp;
    }
}
```

2.Min and max in Array

Given an array **arr**. Your task is to find the **minimum** and **maximum** elements in the array.

Note: Return a Pair that contains **two** elements the first one will be a minimum element and the second will be a maximum.

Examples:

Input: arr[] = [3, 2, 1, 56, 10000, 167]
Output: 1 10000
Explanation: minimum and maximum elements of array are 1 and 10000.

Input: arr[] = [1, 345, 234, 21, 56789]
Output: 1 56789
Explanation: minimum and maximum element of array are 1 and 56789.

Input: arr[] = [56789]
Output: 56789 56789
Explanation: Since the array contains only one element so both min & max are same.

```
class Compute
{
    static pair getMinMax(long a[], long n)
    {
        //Write your code here

        long min=a[0];
        long max=a[0];

        for(int i=0;i<n;i++)
        {
            if(a[i]<min)
            {
                min=a[i];
            }
            if(a[i]>max)
            {
                max=a[i];
            }
        }
        pair p=new pair(min,max);

        return p;
    }
}
```

3.Kth Smallest Element

Given an array **arr[]** and an integer **k** where k is smaller than the size of the array, your task is to find the k^{th} **smallest** element in the given array.

Follow up: Don't solve it using the inbuilt sort function.

Examples :

Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3
Output: 7
Explanation: 3rd smallest element in the given array is 7.

Input: arr[] = [2, 3, 1, 20, 15], k = 4
Output: 15
Explanation: 4th smallest element in the given array is 15.

```java
// User function Template for Java

class Solution {
    public static int kthSmallest(int[] arr, int k) {
        // Your code here
        Arrays.sort(arr);

        int n =arr.length;
        return arr[k-1];
    }
}
```

4.Sort 0's,1's and 2s

Given an array **arr[]** containing only **0s, 1s, and 2s.** Sort the array in ascending order.

You need to solve this problem without utilizing the built-in sort function.

Examples:

Input: arr[] = [0, 1, 2, 0, 1, 2]
Output: [0, 0, 1, 1, 2, 2]
Explanation: 0s 1s and 2s are segregated into ascending order.

Input: arr[] = [0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1]
Output: [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2]
Explanation: 0s 1s and 2s are segregated into ascending order.

Follow up: Could you come up with a one-pass algorithm using only constant extra space?

```java
class Solution {
    // Function to sort an array of 0s, 1s, and 2s
    public void sort012(int[] arr) {
        // code here
        int low=0;
        int mid=0;
        int high=arr.length-1;

            while(mid<=high)
            {
                if(arr[mid]==0)
                {
                    swap(low,mid, arr);
                    mid++;
                    low++;
                }
                else if(arr[mid]==2)
                {
                    swap(mid,high, arr);
                        high--;
                }
                else if(arr[mid]==1)
                {
                    mid++;
                }
            }

    }
    void swap(int a,int b,int [] arr)
    {
            int t1=arr[a];
            arr[a]=arr[b];
            arr[b]=t1;
    }
}
```

5.Move all Negative element to the End

Given an unsorted array **arr[ ]** having both negative and positive integers. The task is to place all negative elements at the end of the array without changing the order of positive elements and negative elements.

Note: Don't return any array, just in-place on the array.

Examples:

Input : arr[] = [1, -1, 3, 2, -7, -5, 11, 6]
Output : [1, 3, 2, 11, 6, -1, -7, -5]
Explanation: By doing operations we separated the integers without changing the order.

Input : arr[] = [-5, 7, -3, -4, 9, 10, -1, 11]
Output : [7, 9, 10, 11, -5, -3, -4, -1]

```
//User function Template for Java

class Solution {

    public void segregateElements(int arr[], int n)
    {
        // Your code goes here
        int x=0;
        int temp[]=new int[n];

        for(int i=0;i<n;i++)
        {
            if(arr[i]>0)
            {
                temp[x++]=arr[i];
            }
        }
        for(int i=0;i<n;i++)
        {
            if(arr[i]<0)
            {
                temp[x++]=arr[i];
            }
        }
        for(int i=0;i<n;i++)
        {
            arr[i]=temp[i];
        }

    }
}
```

6.Union of array with Duplicates

Given two arrays **a[]** and **b[]**, the task is to find the number of elements in the union between these two arrays.

The Union of the two arrays can be defined as the set containing distinct elements from both arrays. If there are repetitions, then only one element occurrence should be there in the union.

Note: Elements of **a[]** and **b[]** are not necessarily distinct.

Examples

Input: a[] = [1, 2, 3, 4, 5], b[] = [1, 2, 3]
Output: 5
Explanation: Union set of both the arrays will be 1, 2, 3, 4 and 5. So count is 5.

Input: a[] = [85, 25, 1, 32, 54, 6], b[] = [85, 2]
Output: 7
Explanation: Union set of both the arrays will be 85, 25, 1, 32, 54, 6, and 2. So count is 7.

Input: a[] = [1, 2, 1, 1, 2], b[] = [2, 2, 1, 2, 1]
Output: 2
Explanation: We need to consider only distinct. So count of elements in union set will be 2.

```java
// User function Template for Java

class Solution {
    public static int doUnion(int arr1[], int arr2[]) {
        // Your code here

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

        for(int arr : arr1){
            set.add(arr);
        }

        for(int arr: arr2){
            set.add(arr);
        }

        return set.size();
    }
}
```

7.Rotate array by one

Given an array **arr**, rotate the array by one position in clockwise direction.

Examples:

Input: arr[] = [1, 2, 3, 4, 5]
Output: [5, 1, 2, 3, 4]
Explanation: If we rotate arr by one position in clockwise 5 come to the front and remaining those are shifted to the end.

Input: arr[] = [9, 8, 7, 6, 4, 2, 1, 3]
Output: [3, 9, 8, 7, 6, 4, 2, 1]
Explanation: After rotating clock-wise 3 comes in first position.

Constraints:

$1<=arr.size()<=10^5$

$0<=arr[i]<=10^5$

```
class Compute {

    public void rotate(int arr[], int n)
    {
        int temp=arr[n-1];
        for(int i=n-1;i>=1;i--)
        {
            arr[i]=arr[i-1];
        }
        arr[0]=temp;
    }
}
```

8.Kadane’s Algorithm

Given an integer array **arr[].** You need to find the **maximum** sum of a subarray.

Examples:

Input: arr[] = [2, 3, -8, 7, -1, 2, 3]
Output: 11
Explanation: The subarray {7, -1, 2, 3} has the largest sum 11.

Input: arr[] = [-2, -4]
Output: -2
Explanation: The subarray {-2} has the largest sum -2.

Input: arr[] = [5, 4, 1, 7, 8]
Output: 25
Explanation: The subarray {5, 4, 1, 7, 8} has the largest sum 25.

Constraints:

$1 \leq arr.size() \leq 10^5$

```java
// User function Template for Java
class Solution {

    // arr: input array
    // Function to find the sum of contiguous subarray with maximum sum.
    int maxSubarraySum(int[] arr) {

        // Your code here
        int max=Integer.MIN_VALUE;
        int sum=0;

        for(int i=0;i<arr.length;i++){
            sum +=arr[i];

            if(max<sum){
                max=sum;
            }
            if(sum<0){
                sum=0;
            }
        }
        return max;
    }
}
```

9.Minimise the heights II

Given an array **arr[]** denoting heights of **N** towers and a positive integer **K.**

For **each** tower, you must perform **exactly one** of the following operations **exactly once**.

- **Increase** the height of the tower by **K**
- **Decrease** the height of the tower by **K**

Find out the **minimum** possible difference between the height of the shortest and tallest towers after you have modified each tower.

You can find a slight modification of the problem here.
Note: It is **compulsory** to increase or decrease the height by K for each tower. **After** the operation, the resultant array should **not** contain any **negative integers**.

Examples :

Input: k = 2, arr[] = {1, 5, 8, 10}
Output: 5
Explanation: The array can be modified as {1+k, 5-k, 8-k, 10-k} = {3, 3, 6, 8}.The difference between the largest and the smallest is 8-3 = 5.

Input: k = 3, arr[] = {3, 9, 12, 16, 20}
Output: 11
Explanation: The array can be modified as {3+k, 9+k, 12-k, 16-k, 20-k} -> {6, 12, 9, 13, 17}.The difference between the largest and the smallest is 17-6 = 11.

```java
class Solution {
    int getMinDiff(int[] arr, int k) {
        // code here

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

        int diff=arr[n-1]-arr[0];
        int maxi,mini;

        for(int i=1;i<n;i++){
            if(arr[i]-k<0){
                continue;
            }
            maxi=Math.max(arr[i-1]+k,arr[n-1]-k);
            mini=Math.min(arr[0]+k,arr[i]-k);

            diff=Math.min(diff,maxi-mini);
        }
    return diff;
    }
}
```

10.Minimum Jumps

You are given an array **arr[]** of non-negative numbers. Each number tells you the **maximum number of steps** you can jump forward from that position.

For example:

- If **arr[i] = 3**, you can jump to index **i + 1**, **i + 2**, or **i + 3** from position **i**.
- If **arr[i] = 0**, you **cannot jump forward** from that position.

Your task is to find the **minimum number of jumps** needed to move from the **first** position in the array to the **last** position.

Note: Return **-1** if you can't reach the end of the array.

Examples :

Input: arr[] = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]
Output: 3
Explanation: First jump from 1st element to 2nd element with value 3. From here we jump to 5th element with value 9, and from here we will jump to the last.

Input: arr = [1, 4, 3, 2, 6, 7]
Output: 2
Explanation: First we jump from the 1st to 2nd element and then jump to the last element.

Input: arr = [0, 10, 20]
Output: -1
Explanation: We cannot go anywhere from the 1st element.

```java
class Solution {
    static int minJumps(int[] arr) {
        // your code here
        int jump=0;
        int curr=0;
        int farthestInd=0;
        int i=0;

        while(i<arr.length){
            if(i>farthestInd){
                return -1;
            }
            farthestInd =Math.max(farthestInd,arr[i]+i);

            if(i==curr){
                curr=farthestInd;
                jump++;
                if(curr>=arr.length-1)
                break;
            }
            i++;
        }
        return jump;
    }
}
```

11.Find The Duplicate number

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and using only constant extra space.

Example 1:

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

Example 2:

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

Example 3:

> **Input:** nums = [3,3,3,3,3]
> **Output:** 3

```java
class Solution {
    public int findDuplicate(int[] nums) {
    HashSet<Integer> hs = new HashSet<>();
        for (int num : nums) {
            if (hs.contains(num)) {
                return num;
            }
            hs.add(num);
        }
        return -1;



    }
}
```

```java
// TC: O(n), SC: O(1)
public int findDuplicate_CyclicSortWay(int[] nums) {
    int i = 0;
    while (i < nums.length) {
        if (nums[i] != nums[nums[i]]) swap(nums, i, nums[i]);
        else i++;
    }
    return nums[0];
}

        public void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}



// TC: O(n), SC: O(1)
public int findDuplicate_FloydCycleDetection_AnotherWay(int[] nums) {
    if (nums.length == 0) return -1;
    int slow = nums[0], fast = nums[nums[0]];
    while (slow != fast) {
        slow = nums[slow];
        fast = nums[nums[fast]];
    }
    slow = 0;
    while (slow != fast) {
        slow = nums[slow];
        fast = nums[fast];
    }
    return slow;
}

// TC: O(n), SC: O(1)
public int findDuplicate_FloydCycleDetection(int[] nums) {
    int slow = nums[0], fast = nums[0];
    do {
        slow = nums[slow];
        fast = nums[nums[fast]];
    } while (slow != fast);
    slow = nums[0];

    while (slow != fast) {
        slow = nums[slow];
        fast = nums[fast];
    }
    return slow;
```

```
}

// TC: O(n), SC: O(1)
public int findDuplicate_ModifyInput(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        int val = abs(nums[i]);
        if (nums[val] < 0) return val;
        nums[val] *= -1;
    }
    return -1;
}

// TC: O(n), SC: O(n)
public int findDuplicate_SetApproach(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        if (set.contains(num)) return num;
        set.add(num);
    }
    return 0;
}

// TC: O(nlogn), SC: O(1)
public int findDuplicate_SortApproach(int[] nums) {
    Arrays.sort(nums);
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] == nums[i - 1]) return nums[i];
    }
    return -1;
}
```

12.Merge Without Extra space

Given two sorted arrays **a[]** and **b[]** of size **n** and **m** respectively, the task is to merge them in sorted order without using any **extra space**. Modify **a[]** so that it contains the first **n** elements and modify **b[]** so that it contains the last **m** elements.

Examples:

Input: a[] = [2, 4, 7, 10], b[] = [2, 3]
Output:
2 2 3 4
7 10
Explanation: After merging the two non-decreasing arrays, we get, 2 2 3 4 7 10

Input: a[] = [1, 5, 9, 10, 15, 20], b[] = [2, 3, 8, 13]
Output:
1 2 3 5 8 9
10 13 15 20
Explanation: After merging two sorted arrays we get 1 2 3 5 8 9 10 13 15 20.

Input: a[] = [0, 1], b[] = [2, 3]
Output:
0 1
2 3
Explanation: After merging two sorted arrays we get 0 1 2 3.

```java
class Solution {
    // Function to merge the arrays.
    public void mergeArrays(int a[], int b[]) {
        // code here
        int i=a.length-1;
        int j=0;
        while(i>=0 && j<b.length){
            if(a[i]>b[j]){
                int temp=a[i];
                a[i]=b[j];
                b[j]=temp;
            }
            i--;
            j++;
        }
        Arrays.sort(a);
        Arrays.sort(b);
    }
}
```

13.Merge Intervals

Given an array of `intervals` where `intervals[i] = [start_i, end_i]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input.*

Example 1:

> **Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]
> **Output:** [[1,6],[8,10],[15,18]]
> **Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

Example 2:

> **Input:** intervals = [[1,4],[4,5]]
> **Output:** [[1,5]]
> **Explanation:** Intervals [1,4] and [4,5] are considered overlapping.

```java
class Solution {
    public int[][] merge(int[][] intervals) {

        int n=intervals.length;
        Arrays.sort(intervals,new Comparator<int[]>(){
            public int compare(int a[],int b[]){
                return a[0]-b[0];
            }
        });
        List<List<Integer>> res=new ArrayList<>();
        for(int i=0;i<n;i++){
            if(res.isEmpty() || intervals[i][0]>res.get(res.size()-1).get(1)){
                res.add(Arrays.asList(intervals[i][0],intervals[i][1]));
            }else{
                res.get(res.size()-1).set(1,Math.max(res.get(res.size()-1).get(1),intervals[i][1]));

            }
        }
        int result[][]=new int[res.size()][2];
        for(int i=0;i<res.size();i++){
            result[i][0]=res.get(i).get(0);
            result[i][1]=res.get(i).get(1);
        }
        return result;
    }
```

14.Next Permutation

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 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]

Example 2:

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

Example 3:

> **Input:** nums = [1,1,5]
> **Output:** [1,5,1]

```java
class Solution {
    public void nextPermutation(int[] arr) {
        int ind=-1;
        int n=arr.length;
        for(int i=n-2;i>=0;i--){
            if(arr[i]<arr[i+1]){
                ind=i;
                break;
            }
        }
        if(ind==-1){
            reverse(arr,0,n-1);
            return;
        }
        for(int i=n-1;i>ind;i--){
            if(arr[i]>arr[ind]){
                int temp=arr[i];
                arr[i]=arr[ind];
                arr[ind]=temp;
                break;
            }

        }
        reverse (arr,ind+1,n-1);
```

```java
        }
        reverse (arr,ind+1,n-1);

    }
    public static void reverse(int arr[],int start,int end){
        while(start<end){
            int temp=arr[start];
            arr[start]=arr[end];
            arr[end]=temp;
            start++;end--;
        }
    }
}
```

15.Count Inversion

Given an array of integers **arr[]**. Find the **Inversion Count** in the array.
Two elements arr[i] and arr[j] form an inversion if **arr[i] > arr[j]** and **i < j**.

Inversion Count: For an array, inversion count indicates how far (or close) the array is from being sorted. If the array is already sorted then the inversion count is 0.

If an array is sorted in the reverse order then the inversion count is the maximum.

Examples:

Input: arr[] = [2, 4, 1, 3, 5]
Output: 3
Explanation: The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

Input: arr[] = [2, 3, 4, 5, 6]
Output: 0
Explanation: As the sequence is already sorted so there is no inversion count.

Input: arr[] = [10, 10, 10]
Output: 0
Explanation: As all the elements of array are same, so there is no inversion count.

```java
// Driver Code Ends
class Solution {
    // Function to count inversions in the array.
    static int inversionCount(int arr[]) {
        int l = 0;
        int r = arr.length-1;
        int count[] = {0};
        mergeSort(arr,l,r,count);
        return count[0];
    }
    static void mergeSort(int arr[], int l, int r, int[] count) {
        int mid = (l+r)/2;
        if(l==r){
            return;
        }
        mergeSort(arr,l,mid, count);
        mergeSort(arr, mid+1,r,count);
        conqure (arr, l, mid, r, count);
    }
    static void conqure (int[] arr, int l, int mid, int r, int[] count){
        int left = l;
        int right = mid+1;
        int i=0;
        int merge[] = new int[r-l+1];
        while(left<=mid && right<=r) {
            if(arr[left]<=arr[right]){
```

```
    static void conqure (int[] arr, int l, int mid, int r, int[] count){
        int left = l;
        int right = mid+1;
        int i=0;
        int merge[] = new int[r-l+1];
        while(left<=mid && right<=r) {
            if(arr[left]<=arr[right]){
                merge[i++] = arr[left++];
            }
            else{
                merge[i++] = arr[right++];
                count[0]+=(mid-left+1);
            }
        }
        while(left<=mid) {
            merge[i++] = arr[left++];
        }
        while(right<=r){
            merge[i++] = arr[right++];
        }
        for(int k=l,h=0;h<merge.length;h++,k++){
            arr[k] = merge[h];
        }
    }
}
```

16.Best Time to buy and sell stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i^th` day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

> **Input:** prices = [7,6,4,3,1]
> **Output:** 0
> **Explanation:** In this case, no transactions are done and the max profit = 0.

```java
class Solution {
    public int maxProfit(int[] prices) {
        int buy=prices[0];
        int profit=0;

        for(int i=1;i<prices.length;i++){
            if(buy<prices[i]){
                profit=Math.max(prices[i]-buy,profit);
            }else{
                buy=prices[i];
            }
        }
        return profit;
    }
}
```

View less

17.Two sum pair with 0 sum

Given an integer array **arr**, return all the **unique** pairs [arr[i], arr[j]] such that **i != j** and **arr[i] + arr[j] == 0.**

Note: The pairs must be returned in **sorted** order, the solution array should also be **sorted**, and the answer must not contain any **duplicate** pairs.

Examples:

Input: arr = [-1, 0, 1, 2, -1, -4]
Output: [[-1, 1]]
Explanation: arr[0] + arr[2] = (-1)+ 1 = 0.
arr[2] + arr[4] = 1 + (-1) = 0.
The distinct pair are [-1,1].

Input: arr = [6, 1, 8, 0, 4, -9, -1, -10, -6, -5]
Output: [[-6, 6],[-1, 1]]
Explanation: The distinct pairs are [-1, 1] and [-6, 6].

Expected Time Complexity: O(n log n)
Expected Auxiliary Space: O(n).

```java
// User function Template for Java

class Solution {
    int getPairsCount(int[] arr, int k) {
        // code here
        int res = 0;
        int n=arr.length;
        HashMap<Integer,Integer> freqMap = new HashMap<>();

        for(int i=0;i<n;i++){
            if(arr[i]>k){
                continue;
            }
            int secondVal = k-arr[i];
            res += freqMap.getOrDefault(secondVal,0);
            freqMap.put(arr[i],freqMap.getOrDefault(arr[i],0)+1);
        }
        return res;

    }
}
```

18.Common in 3 sorted array

Given three sorted arrays in **non-decreasing** order, print all common elements in **non-decreasing** order across these arrays. If there are no such elements return an empty array. In this case, the output will be -1.

Note: can you handle the duplicates without using any additional Data Structure?

Examples :

Input: arr1 = [1, 5, 10, 20, 40, 80] , arr2 = [6, 7, 20, 80, 100] , arr3 = [3, 4, 15, 20, 30, 70, 80, 120]
Output: [20, 80]
Explanation: 20 and 80 are the only common elements in arr, brr and crr.

Input: arr1 = [1, 2, 3, 4, 5] , arr2 = [6, 7] , arr3 = [8,9,10]
Output: [-1]
Explanation: There are no common elements in arr, brr and crr.

Input: arr1 = [1, 1, 1, 2, 2, 2], B = [1, 1, 2, 2, 2], arr3 = [1, 1, 1, 1, 2, 2, 2, 2]
Output: [1, 2]
Explanation: We do not need to consider duplicates

```
49
50  class Solution {
51      // Function to find common elements in three arrays.
52      public List<Integer> commonElements(List<Integer> arr1, List<Integer> arr2,
53                                           List<Integer> arr3) {
54          // Code Here
55          Set<Integer>set=new HashSet<>(arr1);
56          Set<Integer>set1=new HashSet<>(arr2);
57          TreeSet<Integer> set3 =new TreeSet<>(arr3);
58          List<Integer>list=new ArrayList<>();
59          for (int i:set3){
60              if (set1.contains(i)&&set.contains(i)){
61                  list.add(i);
62              }
63          }
64          return list;
65
66      }
67  }
```

19.Subarray with 0 sum

Given an array of integers, **arr[]**. Find if there is a **subarray** (of size at least one) with **0 sum**. Return true/false depending upon whether there is a subarray present with 0-sum or not.

Examples:

Input: arr[] = [4, 2, -3, 1, 6]
Output: true
Explanation: 2, -3, 1 is the subarray with a sum of 0.

Input: arr = [4, 2, 0, 1, 6]
Output: true
Explanation: 0 is one of the element in the array so there exist a subarray with sum 0.

Input: arr = [1, 2, -1]
Output: false

Constraints:

$1 <= arr.size <= 10^4$

$-10^5 <= arr[i] <= 10^5$

```java
class Solution {
    // Function to check whether there is a subarray present with 0-sum or not.
    static boolean findsum(int arr[]) {
        // Your code here
        HashSet<Integer> ans=new HashSet<>();
        int prefix=0;
        for(int num:arr){
            prefix+=num;
            if(prefix==0 || ans.contains(prefix)){
                return true;
            }
            ans.add(prefix);
        }
        return false;
    }
}
```

20.Factorial of large number

Given an integer **n,** find its factorial. Return a list of integers denoting the digits that make up the factorial of n.

Examples:

Input: n = 5
Output: [1, 2, 0]
Explanation: 5! = 1*2*3*4*5 = 120

Input: n = 10
Output: [3, 6, 2, 8, 8, 0, 0]
Explanation: 10! = 1*2*3*4*5*6*7*8*9*10 = 3628800

Input: n = 1
Output: [1]
Explanation: 1! = 1

Constraints:
$1 \leq n \leq 10^3$

```java
class Solution {
    public static ArrayList<Integer> factorial(int n) {
        // code here
        ArrayList<Integer> res = new ArrayList<>();
        res.add(1);
        while(n>1){
            int carry = 0;
            for(int i=res.size()-1;i>=0;i--){
                int mult = res.get(i)*n + carry;
                carry = mult/10;
                res.set(i,mult%10);
            }
            while(carry>0){
                res.add(0,carry%10);
                carry = carry/10;
            }
            n--;
        }
        return res;
    }
}

// } Driver Code Ends
```

```java
import java.math.BigInteger;

class Solution {
    public static BigInteger fact(int n)
    {
        if(n == 1 || n==0)
        {
            return BigInteger.ONE;
        }
        return BigInteger.valueOf(n).multiply(fact(n-1));
    }
    public static ArrayList<Integer> factorial(int n) {
        // code here

        BigInteger result=fact(n);


        ArrayList<Integer> digits = new ArrayList<>();

        for (char c : result.toString().toCharArray()) {
            digits.add(c - '0');
        }
        return digits;
    }
}
```

21.Maximum Product subarray

Given an array **arr[]** that contains positive and negative integers (may contain 0 as well). Find the **maximum** product that we can get in a subarray of **arr[]**.

Note: It is guaranteed that the output fits in a 32-bit integer.

Examples

Input: arr[] = [-2, 6, -3, -10, 0, 2]
Output: 180
Explanation: The subarray with maximum product is {6, -3, -10} with product = 6 * (-3) * (-10) = 180.

Input: arr[] = [-1, -3, -10, 0, 6]
Output: 30
Explanation: The subarray with maximum product is {-3, -10} with product = (-3) * (-10) = 30.

Input: arr[] = [2, 3, 4]
Output: 24
Explanation: For an array with all positive elements, the result is product of all elements.

```java
class Solution {
    // Function to find maximum product subarray
    int maxProduct(int[] arr) {
        // code here
        int n=arr.length;
        int maxi = arr[0];
        int mini = arr[0];
        int ans = arr[0];

        for(int i=1;i<n;i++){
            if(arr[i] < 0)
            {
                int temp = maxi;
                maxi = mini;
                mini = temp;
            }

            maxi = Math.max(arr[i],maxi * arr[i]);
            mini = Math.min(arr[i], mini * arr[i]);

            ans = Math.max(maxi,ans);
        }
        return ans;
    }
}
```

22.Longest Consecutive subsequence

Given an array **arr[]** of non-negative integers. Find the **length** of the longest sub-sequence such that elements in the subsequence are consecutive integers, the **consecutive numbers** can be in **any order.**

Examples:

Input: arr[] = [2, 6, 1, 9, 4, 5, 3]
Output: 6
Explanation: The consecutive numbers here are 1, 2, 3, 4, 5, 6. These 6 numbers form the longest consecutive subsquence.

Input: arr[] = [1, 9, 3, 10, 4, 20, 2]
Output: 4
Explanation: 1, 2, 3, 4 is the longest consecutive subsequence.

Input: arr[] = [15, 13, 12, 14, 11, 10, 9]
Output: 7
Explanation: The longest consecutive subsequence is 9, 10, 11, 12, 13, 14, 15, which has a length of 7.

```java
class Solution {

    // Function to return length of longest subsequence of consecutive integers.
    public int longestConsecutive(int[] arr) {
        // code here
        HashMap<Integer,Boolean> map=new HashMap<>();
        for(int i=0;i<arr.length;i++){
            map.put(arr[i],false);
        }

        for(int key : map.keySet()){
            if(map.containsKey(key-1)==false){
                map.put(key,true);
            }
        }
        int max=0;
        for(int key : map.keySet()){
            int k=1;
            if(map.get(key)==true){
                while(map.containsKey(key+k)==true){
                    k++;
                }
            }
            max=Math.max(max,k);
        }
        return max;
    }
}
```

23.Majority Element II

Given an integer array of size `n`, find all elements that appear more than `⌊ n/3 ⌋` times.

Example 1:

> **Input:** nums = [3,2,3]
> **Output:** [3]

Example 2:

> **Input:** nums = [1]
> **Output:** [1]

Example 3:

> **Input:** nums = [1,2]
> **Output:** [1,2]

```java
class Solution {
    public List<Integer> majorityElement(int[] nums) {
        HashMap<Integer,Integer> map=new HashMap<>();
        for(int num : nums){
            map.put(num,map.getOrDefault(num,0)+1);
        }
        int n=nums.length;
        List<Integer> res=new ArrayList<>();
        for(Map.Entry<Integer,Integer> entry : map.entrySet()){
            if(entry.getValue()>n/3){
                res.add(entry.getKey());
            }
        }
        return res;
    }
}
```

24.Best time to sell and buy stock III

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

Find the maximum profit you can achieve. You may complete **at most two transactions**.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

> **Input:** prices = [3,3,5,0,0,3,1,4]
> **Output:** 6
> **Explanation:** Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
> Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

Example 2:

> **Input:** prices = [1,2,3,4,5]
> **Output:** 4
> **Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
> Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

Example 3:

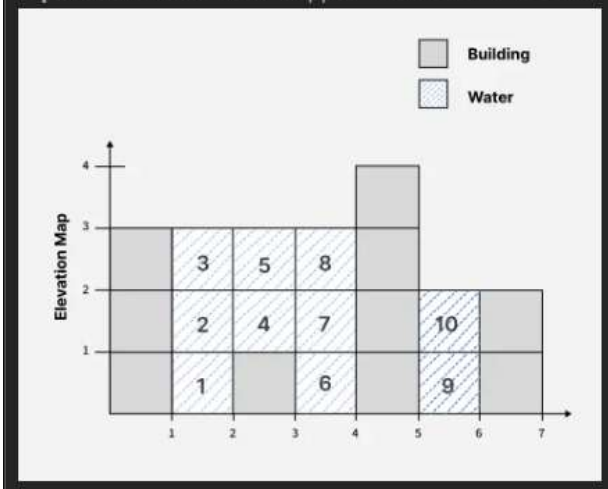
> **Input:** prices = [7,6,4,3,1]
> **Output:** 0
> **Explanation:** In this case, no transaction is done, i.e. max profit = 0.

```java
class Solution {
    public int maxProfit(int[] prices) {
        int sell1 = 0, sell2 = 0, buy1 = Integer.MIN_VALUE, buy2 = Integer.MIN_VALUE;
        for (int i = 0; i < prices.length; i++) {
            buy1 = Math.max(buy1, -prices[i]);
            sell1 = Math.max(sell1, buy1 + prices[i]);
            buy2 = Math.max(buy2, sell1 - prices[i]);
            sell2 = Math.max(sell2, buy2 + prices[i]);
        }
        return sell2;
    }
}
```

25.Trapping Rain Water

Given an array **arr[]** with non-negative integers representing the height of blocks. If the width of each block is 1, compute how much water can be trapped between the blocks during the rainy season.

Input: arr[] = [3, 0, 1, 0, 4, 0 2]
Output: 10
Explanation: Total water trapped = 0 + 3 + 2 + 3 + 0 + 2 + 0 = 10 units.



Input: arr[] = [3, 0, 2, 0, 4]
Output: 7
Explanation: Total water trapped = 0 + 3 + 1 + 3 + 0 = 7 units.

Input: arr[] = [1, 2, 3, 4]
Output: 0
Explanation: We cannot trap water as there is no height bound on both sides.

Input: arr[] = [2, 1, 5, 3, 1, 0, 4]
Output: 9
Explanation: Total water trapped = 0 + 1 + 0 + 1 + 3 + 4 + 0 = 9 units.

```java
class Solution {
    public int maxWater(int arr[]) {
        // code here
        int n=arr.length;
        int left_sum=0;
        int right_sum=0;
        int i=0;
        int j=n-1;
        int res=0;

        while(i<=j){
            if(arr[i]<arr[j]){
                if(arr[i]>left_sum){
                    left_sum=arr[i];
                }else{
                    res+=left_sum-arr[i];
                }
                i++;
            }else{
                if(arr[j]>right_sum){
                    right_sum=arr[j];
                }else{
                    res+=right_sum-arr[j];
                }
                j--;
            }
        }
        return res;
    }
}
```

26.Chocolate distribution problem

Given an array **arr[]** of positive integers, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **m** students, the task is to distribute chocolate packets among **m** students such that -

i. Each student gets **exactly** one packet.

ii. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum and return that minimum possible difference.

Examples:

Input: arr = [3, 4, 1, 9, 56, 7, 9, 12], m = 5
Output: 6
Explanation: The minimum difference between maximum chocolates and minimum chocolates is 9 - 3 = 6 by choosing following m packets :[3, 4, 9, 7, 9].

Input: arr = [7, 3, 2, 4, 9, 12, 56], m = 3
Output: 2
Explanation: The minimum difference between maximum chocolates and minimum chocolates is 4 - 2 = 2 by choosing following m packets :[3, 2, 4].

Input: arr = [3, 4, 1, 9, 56], m = 5
Output: 55
Explanation: With 5 packets for 5 students, each student will receive one packet, so the difference is 56 - 1 = 55.

```java
class Solution {
    public int findMinDiff(ArrayList<Integer> arr, int m) {
        // your code here
        int n=arr.size();
        if(n==0){
            return 0;
        }
        Collections.sort(arr);
        int res=Integer.MAX_VALUE;

        for(int i=0;i<n-m+1;i++){
            int minEle=arr.get(i);
            int maxEle=arr.get(i+m-1);
            res=Math.min(res,maxEle-minEle);
        }
        return res;
    }
}
```

27.Minimum size subarray sum

Given an array of positive integers `nums` and a positive integer `target`, return *the* ***minimal length*** *of a subarray whose sum is greater than or equal to* `target`. If there is no such subarray, return `0` instead.

Example 1:

> **Input:** target = 7, nums = [2,3,1,2,4,3]
> **Output:** 2
> **Explanation:** The subarray [4,3] has the minimal length under the problem constraint.

Example 2:

> **Input:** target = 4, nums = [1,4,4]
> **Output:** 1

Example 3:

> **Input:** target = 11, nums = [1,1,1,1,1,1,1,1]
> **Output:** 0

```java
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int i=0;
        int j=0;
        int sum=0;
        int min=Integer.MAX_VALUE;

        while(j<nums.length){
            sum+=nums[j];
            j++;
            while(sum>=target){
                min=Math.min(min,j-i);
                sum-=nums[i];
                i++;
            }
        }
        return min==Integer.MAX_VALUE?0:min;
    }
}
```

28.Median of two sorted array

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be `O(log (m+n))`.

Example 1:

> **Input:** nums1 = [1,3], nums2 = [2]
> **Output:** 2.00000
> **Explanation:** merged array = [1,2,3] and median is 2.

Example 2:

> **Input:** nums1 = [1,2], nums2 = [3,4]
> **Output:** 2.50000
> **Explanation:** merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

```java
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int ans[] =merge(nums1,nums2);

        if(ans.length%2 ==0){
            double ans2=(double)(ans[ans.length/2] + ans[ans.length/2-1])/2;
            return ans2;
        }else{
            double ans2 =(double)(ans[ans.length/2]);

            return ans2;
        }
    }

    public int[] merge(int[] arr1,int[] arr2){
        int ans[] =new int[arr1.length+arr2.length];

        int p1=0;
        int p2=0;
        int p3=0;

        while(p1<arr1.length || p2<arr2.length){
            int val1 =p1<arr1.length ? arr1[p1]:Integer.MAX_VALUE;
            int val2=p2<arr2.length ? arr2[p2]:Integer.MAX_VALUE;
```

```java
        while(p1<arr1.length || p2<arr2.length){
            int val1 =p1<arr1.length ? arr1[p1]:Integer.MAX_VALUE;
            int val2=p2<arr2.length ? arr2[p2]:Integer.MAX_VALUE;

            if(val1<val2){
                ans[p3]=val1;
                p1++;
            }else{
                ans[p3] =val2;
                p2++;
            }
            p3++;
        }
        return ans;
}
```

29.Alternate Positive Negative

Given an unsorted array **arr** containing both **positive** and **negative** numbers. Your task is to rearrange the array and convert it into an array of alternate positive and negative numbers without changing the relative order.

Note:

- Resulting array **should start** with a positive integer (0 will also be considered as a positive integer).
- If any of the positive or negative integers are exhausted, then add the remaining integers in the answer as it is by maintaining the relative order.
- The array **may** or **may not** have the **equal** number of positive and negative integers.

Examples:

Input: arr[] = [9, 4, -2, -1, 5, 0, -5, -3, 2]
Output: [9, -2, 4, -1, 5, -5, 0, -3, 2]
Explanation: The positive numbers are [9, 4, 5, 0, 2] and the negative integers are [-2, -1, -5, -3]. Since, we need to start with the positive integer first and then negative integer and so on (by maintaining the relative order as well), hence we will take 9 from the positive set of elements and then -2 after that 4 and then -1 and so on.

Input: arr[] = [-5, -2, 5, 2, 4, 7, 1, 8, 0, -8]
Output: [5, -5, 2, -2, 4, -8, 7, 1, 8, 0]
Explanation : The positive numbers are [5, 2, 4, 7, 1, 8, 0] and the negative integers are [-5,-2,-8]. According to the given conditions we will start from the positive integer 5 and then -5 and so on. After reaching -8 there are no negative elements left, so according to the given rule, we will add the remaining elements (in this case positive elements are remaining) as it in by maintaining the relative order.

Input: arr[] = [9, 5, -2, -1, 5, 0, -5, -3, 2]
Output: [9, -2, 5, -1, 5, -5, 0, -3, 2]
Explanation: The positive numbers are [9, 5, 5, 0, 2] and the negative integers are [-2, -1, -5, -3]. Since, we need to start with the positive integer first and then negative integer and so on (by maintaining the relative order as well), hence we will take 9 from the positive set of elements and then -2 after that 5 and then -1 and so on.

```java
class Solution {
    void rearrange(ArrayList<Integer> arr) {
        // code here
         List<Integer> pos=new ArrayList<>();
        List<Integer> neg=new ArrayList<>();

        int []res=new int[arr.size()];

        for(int x : arr){

            if(x>=0){
                pos.add(x);
            }
            else{
                neg.add(x);
            }
        }

        int i=0,j=0;
        int x=0;

        while(i<pos.size()  && j<neg.size()){

           arr.remove(x);
           arr.add(x++,pos.get(i++) );
           arr.remove(x);
           arr.add(x++,neg.get(j++) );

        }

        while(i<pos.size()){
           arr.remove(x);
           arr.add(x++, pos.get(i++) );
        }

        while(j<neg.size()){
           arr.remove(x);
           arr.add(x++,neg.get(j++) );
        }
    }
}
```

30.Three way partitioning

Given an **array** and a range **a, b**. The task is to partition the array around the range such that the array is divided into three parts.
1) All elements smaller than **a** come first.
2) All elements in range **a** to **b** come next.
3) All elements greater than **b** appear in the end.
The individual elements of three sets can appear in any order. You are required to return the modified array.

Note: The generated output is true if you modify the given array successfully. Otherwise false.

Geeky Challenge: Solve this problem in O(n) time complexity.

Examples:

Input: arr[] = {1, 2, 3, 3, 4}, a = 1, b = 2
Output: true
Explanation: One possible arrangement is: {1, 2, 3, 3, 4}. If you return a valid arrangement, output will be true.

Input: arr[] = {1, 4, 3, 6, 2, 1}, a = 1, b = 3
Output: true
Explanation: One possible arrangement is: {1, 3, 2, 1, 4, 6}. If you return a valid arrangement, output will be true.

```java
class Solution {
    // Function to partition the array around the range such
    // that array is divided into three parts.
    public void threeWayPartition(int arr[], int a, int b) {
        // code here
        int start = 0;
        int end = 0;

        while(end < arr.length){
            if(arr[end] >= a){
                end++;
            }
            else{
                int temp = arr[start];
                arr[start] = arr[end];
                arr[end] = temp;
                start++;
                end++;
            }
        }

        end = start;

         while(end < arr.length){
            if(arr[end] >= b){
                end++;
            }
            else{
                int temp = arr[start];
                arr[start] = arr[end];
                arr[end] = temp;
                start++;
                end++;
            }
        }
    }
}
```

31.Minimum swaps and k Together

Given an array **arr** and a number **k**. One can apply a swap operation on the array any number of times, i.e choose any two index i and j (i < j) and swap arr[i] , arr[j] . Find the **minimum** number of swaps required to bring all the numbers less than or equal to **k** together, i.e. make them a contiguous subarray.

Examples :

Input: arr[] = [2, 1, 5, 6, 3], k = 3
Output: 1
Explanation: To bring elements 2, 1, 3 together, swap index 2 with 4 (0-based indexing), i.e. element arr[2] = 5 with arr[4] = 3 such that final array will be- arr[] = [2, 1, 3, 6, 5]

Input: arr[] = [2, 7, 9, 5, 8, 7, 4], k = 6
Output: 2
Explanation: To bring elements 2, 5, 4 together, swap index 0 with 2 (0-based indexing) and index 4 with 6 (0-based indexing) such that final array will be- arr[] = [9, 7, 2, 5, 4, 7, 8]

Input: arr[] = [2, 4, 5, 3, 6, 1, 8], k = 6
Output: 0

```java
// User function Template for Java

class Solution {
    // Function for finding maximum and value pair
    int minSwap(int[] arr, int k) {
        // Complete the function
        int total = 0;
        int temp = 0;
        int max = 0;
        int n = arr.length;
        for(int i=0; i<n; i++){
            if(arr[i]<=k){
                total++;
                temp++;
            }else{
                max = Math.max(temp, max);
                temp = 0;
            }
        }
        return total == temp ? 0 : total - max;
    }
}
```

32.Minimum Number of moves to make Palindrome

You are given a string `s` consisting only of lowercase English letters.

In one **move**, you can select any two **adjacent** characters of `s` and swap them.

Return *the **minimum number of moves** needed to make `s` a palindrome.*

Note that the input will be generated such that `s` can always be converted to a palindrome.

Example 1:

> **Input:** s = "aabb"
> **Output:** 2
> **Explanation:**
> We can obtain two palindromes from s, "abba" and "baab".
> - We can obtain "abba" from s in 2 moves: "a**ab**b" -> "ab**ab**" -> "abba".
> - We can obtain "baab" from s in 2 moves: "a**ab**b" -> "**ab**ab" -> "baab".
> Thus, the minimum number of moves needed to make s a palindrome is 2.

Example 2:

> **Input:** s = "letelt"
> **Output:** 2
> **Explanation:**
> One of the palindromes we can obtain from s in 2 moves is "lettel".
> One of the ways we can obtain it is "lete**lt**" -> "lete**et**l" -> "lettel".
> Other palindromes such as "tleelt" can also be obtained in 2 moves.
> It can be shown that it is not possible to obtain a palindrome in less than 2 moves.

```java
class Solution {
    public int minMovesToMakePalindrome(String s) {
        int count = 0;

        while(s.length() > 2) {
            char ch1 = s.charAt(0);
            int len = s.length();
            char ch2 = s.charAt(len - 1);

            if (ch1 == ch2) {
                s = s.substring(1, len - 1);
            } else {
                int id1 = s.lastIndexOf(ch1);
                int id2 = s.indexOf(ch2);

                int steps1 = len - id1 - 1;
                int steps2 = id2;

                StringBuilder sb = new StringBuilder();

                if (steps1 > steps2) {
                    count += steps2;
```

```java
                StringBuilder sb = new StringBuilder();

                if (steps1 > steps2) {
                    count += steps2;
                    sb.append(s.substring(0, id2));
                    sb.append(s.substring(id2 + 1, len - 1));
                } else {
                    count += steps1;
                    sb.append(s.substring(1, id1));
                    sb.append(s.substring(id1 + 1));
                }

                s = sb.toString();
            }
        }

        return count;
    }
}
```

33.Median of 2 sorted Array of same size

Given two sorted arrays **a[]** and **b[]** of equal size, find and return the **median** of the combined array after merging them into a single sorted array.

Examples:

Input: a[] = [-5, 3, 6, 12, 15], b[] = [-12, -10, -6, -3, 4]
Output: 0
Explanation: The merged array is [-12, -10, -6, -5, -3, 3, 4, 6, 12, 15]. So the median of the merged array is (-3 + 3) / 2 = 0.

Input: a[] = [2, 3, 5, 7], b[] = [10, 12, 14, 16]
Output: 8.5
Explanation: The merged array is [2, 3, 5, 7, 10, 12, 14, 16]. So the median of the merged array is (7 + 10) / 2 = 8.5.

Input: a[] = [-5], b[] = [-6]
Output: -5.5
Explanation: The merged array is [-6, -5]. So the median of the merged array is (-6 + -5) / 2 = -5.5.

```java
class Solution {
    public double medianOf2(int a[], int b[]) {
        // Your Code Here
        int len=a.length+b.length;
        ArrayList<Integer>res = new ArrayList<Integer>();
        for(int x : a){
            res.add(x);
        }

        for(int x : b){
            res.add(x);
        }
        Collections.sort(res);
        if(res.size() % 2 != 0){
            int i = res.size()/2;
            double result = res.get(i);

            return result;
        }else{
            int i = res.size()/2;
            int j = i-1;

            double result = (res.get(i) + res.get(j)) / 2.0;

            return result;
        }
    }
}
```