**Unique number of occurences**

**Ap 1:**

1. Sort the array to same element together
2. Store count of each element in array called frequency and sort it
3. Now iterate over frequency array if two adjacent element of frequency array become same then return false
4. Otherwise return true

Time complexity: O(nlogn) Space complexity: O(n)

**Ap 2: using map and set**

1. Create an unordered map freq to store the frequency of each element in the array.
2. Create an unordered set s to store unique frequencies.
3. Iterate over freq and check if element already present in set, I.e., (duplicate)
   1. Return false
4. Return true

Time complexity: O(n) Space complexity: O(n)

public boolean uniqueOccurrences(int[] arr) {

        if(arr.length<=1){

            return true;

        }

        Map<Integer, Integer> freq = new HashMap<>();

        for(int e:arr){

            freq.put(e, freq.getOrDefault(e,0)+1);

        }

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

        for(int e:freq.values()){

            if(s.contains(e)){

                return false;

            }

            else{

                s.add(e);

            }

        }

        return true;

    }

# **Find duplicates in an array without using extra space**

public static ArrayList<Integer> duplicates(int arr[], int n) {

    // current element value = going to that index

    // updating that index value by adding length of arr

    for(int i=0; i<n; i++){

        int index = arr[i]%n;

        arr[index]+=n;

    }

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

    int flag = 0;

    for(int i=0; i<n; i++){

        // duplicates element

        if(arr[i]/n > 1){

            // index i will be the duplicate element

            ans.add(i);

            flag=1;

        }

    }

    if(flag!=1){

        ans.add(-1);

    }

    return ans;

}

1. **Sort 0, 1 and 2. Sort the array without using any sorting algo**

**ap 1: By counting frequency**

1. Count the frequency of 0s, 1s and 2s in the array.
2. Start from i=0 and first fill the number of 0s present in the array.
3. Then fill the number of 1s present in the array.
4. Then fill the number of 2s present in the array.

Time complexity: O(n) Space complexity: O(1)

**Ap 2: 3-Pointers approach**

**Initialize Pointers:**

**current:** Represents the position of the current element during the array iteration. Initialized to 0.

**zeroPos:** Holds the index where 0s will be pushed. Initialized to 0.

**twoPos:** Holds the index where 2s will be pushed. Initialized to n - 1

**Array Iteration:**

Iterate through the array using the current pointer.

**Handling 0s (arr[current] = 0):**

Swap arr[current] with arr[zeroPos].

Increment both current and zeroPos by 1.

**Handling 1s (arr[current] = 1):**

Simply increment current by 1.

No swapping is needed for 1s

**Handling 2s (arr[current] = 2):**

Swap arr[current] with arr[twoPos].

Decrement twoPos by 1.

Note: Do not increment current in this case.

**Termination Condition:**

The loop terminates when current reaches the value of twoPos.

The elements before zeroPos are 0s, and those after twoPos are 2s. The remaining elements between zeroPos and twoPos are 1s and need to be sorted.

**Reasoning:**

zeroPos and twoPos effectively mark the boundaries of 0s and 2s in the array.

Elements before zeroPos are 0s, and elements after twoPos are 2s.

The loop continues until the remaining unsorted elements (between current and twoPos) are sorted.

**Note on Incrementing current with arr[current] = 2:**

We cannot blindly increment current after swapping with twoPos because the value originally at twoPos could be 0, 1, or 2. This uncertainty requires checking the swapped value.

Time complexity: O(n) Space complexity: O(1)

 public static void sort012(int[] arr) {

        int nextZero = 0;

        int nextTwo = arr.length - 1;

        int i = 0;

        while (i <= nextTwo) {

            // If current element is 0, place it at the 'nextZero' pointer.

            if (arr[i] == 0) {

                int temp = arr[nextZero];

                arr[nextZero] = arr[i];

                arr[i] = temp;

                i++;

                nextZero++;

            }

            // Else if element = 2, place it at the 'nextTwo' pointer.

            else if (arr[i] == 2) {

                int temp = arr[nextTwo];

                arr[nextTwo] = arr[i];

                arr[i] = temp;

                nextTwo--;

            }

            // Else, if element = 1, let it remain in mid position.

            else {

                i++;

            }

        }

}

**10.Move all the negative elements to one side of the array**

**ap 1: sorting**

Time complexity: O(nlog(n)) Space complexity: O(1)

**Ap 2: 2-pointers**

Time complexity: O(n) Space complexity: O(1)

public static int[] separateNegativeAndPositive(int a[]) {

        int s=0;

        int e=a.length-1;

        while(s<e){

            if(a[s]>=0 && a[e]<=0){

                int t=a[s];

                a[s]=a[e];

                a[e]=t;

            }

            if(a[s]<0){

                s++;

            }

            if(a[e]>=0){

                e--;

            }

        }

        return a;

    }

**11\_Union of the two sorted arrays**

**Ap 1: using set**

1. Define a set
2. Iterate over 1st array and insert into set
3. Iterate over 2nd array and add into se
4. Now set contains only all unique ans
5. Retrieve from set and stored in ans array and sort the array

**TC : O( n1 log(n) + n2 log(n) ) + o(n1+n2) ;** log(n) used to store element in set

**SC : O(n1+n2)**

**Ap 2 : two pointers**

1. Initialize two pointers I=0 and j =0
2. Until I<arr1.length and j<arr2.length
   1. While inserting checking if it is same as the last element of unionArray
      1. If not , add into unionArray
   2. Increment both pointer accordingly

**TC : O ( n1+n2 )**

**SC : O ( n1+n2) ;** which is the result array

public static int doUnion(int a[], int n, int b[], int m)

    {

        int count=0;

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

        int i = 0;

        int j = 0;

        while(i<n && j<m){

            if(a[i]<=b[j]){

                if(unionList.size()==0 || unionList.get(unionList.size()-1) != a[i]){

                    unionList.add(a[i]);

                    count++;

                }

                i++;

            }

            else{

                if(unionList.size()==0 || unionList.get(unionList.size()-1) != b[j]){

                    unionList.add(b[j]);

                    count++;

                }

                j++;

            }

        }

        while(i<n){

            if(unionList.size()==0 || unionList.get(unionList.size()-1) != a[i]){

                    unionList.add(a[i]);

                    count++;

                }

                i++;

        }

        while(j<m){

            if(unionList.size()==0 || unionList.get(unionList.size()-1) != b[j]){

                    unionList.add(b[j]);

                    count++;

                }

                j++;

        }

        return count;

    }

**Intersection of two sorted arrays**

class Solution {

    public int[] intersection(int[] nums1, int[] nums2) {

        HashSet<Integer> set1 = new HashSet<>();

        HashSet<Integer> set2 = new HashSet<>();

        // Add elements of nums1 to set1

        for (int num : nums1) {

            set1.add(num);

        }

        // Add elements of nums2 to set2

        for (int num : nums2) {

            set2.add(num);

        }

        // Intersection set to store common elements

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

        // Find common elements

        for (int num : set1) {

            if (set2.contains(num)) {

                intersection.add(num);

            }

        }

        // Convert the intersection set to an array

        int[] ans = new int[intersection.size()];

        int index = 0;

        for (int num : intersection) {

            ans[index++] = num;

        }

        return ans;

    }}

**Kadane's Algo [V.V.V.V.V IMP]**

1. We will run a loop(say i) to iterate the given array.
2. Now, while iterating we will add the elements to the sum variable and consider the maximum one.
3. If at any point the sum becomes negative we will set the sum to 0 as we are not going to consider it as a part of our answer.

long maxi = Long.MIN\_VALUE; // maximum sum

        long sum = 0;

        for (int i = 0; i < n; i++) {

            sum += arr[i];

            if (sum > maxi) {

                maxi = sum;

            }

            // If sum < 0: discard the sum calculated

            if (sum < 0) {

                sum = 0;

            }

        }

        // To consider the sum of the empty subarray

        // uncomment the following check:

        //if (maxi < 0) maxi = 0;

        return maxi;

**find Largest sum contiguous Subarray [V. IMP]**

**Ap 1: using Kadane algorithm**

1. Calculate first k elements sum
2. Then, start iterating from i = k to end of array
3. Update current sum by adding current elemnt and subtracting (i-k)th element
4. Store max of maxSum and curSum in maxSum

Time complexity: O(n) Space complexity: O(1)

static int maxContiguousSubarraySum(int[] arr,int n,int k){

        int curSum=0;

        int maxSum=0;

        // sum of first k elements

        for(int i=0; i<k; i++){

            curSum+=arr[i];

        }

        // this loop starts from k+1 pos

        maxSum=Math.max(curSum, maxSum);

        for(int i=k; i<n; i++){

            curSum = curSum+arr[i]-arr[i-k];

            maxSum=Math.max(curSum, maxSum);

        }

        return maxSum;

    }

**Minimise the maximum difference between heights [V.IMP]**

**Ap 1:**

1. Sort ‘Arr’.
2. Create a variable, ‘ans’, and initialize it to the difference between the last and first terms of ‘Arr’.
3. Create two variables, ‘minimum’ and ‘maximum’ and initialize them to ‘Arr[0] + K’ and ‘Arr[N - 1] - K’, respectively.
4. Create two variables, ‘current\_minimum’ and ‘current\_maximum’.
5. Iterate using a for loop from i = ‘0’ to ‘N - 1’.
   1. Update ‘current\_minimum’ to min(minimum, A[i + 1] - K).
   2. Update ‘current\_maximum’ to max(maximum, A[i] + K).
   3. If ‘current\_minimum’ is non-negative.
      1. Update ‘ans’ to min(ans, current\_maximum - current\_minimum).
6. return ‘ans’.

Time complexity: O(nlog(n)) Space complexity: O(1)

public static int minimizeIt(int[] A, int K) {

        int n = A.length;

        Arrays.sort(A);

        int ans = A[n - 1] - A[0];

        int minimum = A[0] + K;

        int maximum = A[n - 1] - K;

        int current\_minimum = 0, current\_maximum = 0;

        for (int i = 0; i < n - 1; i++)

        {

            // Finding the minimum and the maximum value if we perform the decrement operation on all the

            // indexes greater than index i and increment operation until the index i.

            current\_minimum = Math.min(A[i + 1] - K, minimum);

            current\_maximum = Math.max(A[i] + K, maximum);

            // Checking if minimum value is non-negative or not.

            if (current\_minimum >= 0)

            {

                ans = Math.min(ans, current\_maximum - current\_minimum);

            }

        }

        return ans;

}

**Find similarities between two arrays.**

public static ArrayList<Integer> findSimilarity(ArrayList<Integer> arr1, ArrayList<Integer> arr2, int n, int m) {

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

        HashSet<Integer>set1=new HashSet<>();

        HashSet<Integer>set2=new HashSet<>();

        // storing elements of arr1

        for(int i=0;i<n;i++){

            set1.add(arr1.get(i));

        }

        // storing elements of arr2

        for(int i=0;i<m;i++){

            set2.add(arr2.get(i));

        }

        // to count common eleemnts(intersection)

        int similar=0;

        for(int i=0;i<n;i++){

            if(set2.contains(arr1.get(i))){

                similar++;

            }

        }

        // size of intersection of two list

        ans.add(similar);

        // size for union of two list

        ans.add(m+n-similar);

        return ans;

}

**find duplicate in an array of N+1 Integers**

Only 1 duplicate will be present

**ap1:** sorting : after sorting we just have to compare ith element with (i-1)th element.If equal then that is duplicate

Tc=O(nlogn)

**Ap2:** using set : iterate and add element into set.before adding check if already present or not in set.If present then that is duplicate.

Tc=O(n) sc=O(n)

**Ap 3 : bitwise xor**

public static int findDuplicate(ArrayList<Integer> arr) {

        int ans=0;

        for(int i : arr){

            ans^=i;

        }

        for(int i=0;i<arr.size();i++){

            ans^=i;

        }

        return ans;

    }

**Ap3:** Inplace modification(optimal) : For each number we go to its index position and multiply it with ’-1’, thus making it negative. In case of duplicate, it will visit twice and hence will become positive, which will be returned.

    Time complexity: O(N)

    Space complexity: O(1)

    Where N is the length of the array.

\*/

import java.util.ArrayList;

public class Solution{

    public static int findDuplicate(ArrayList<Integer> arr, int n){

        // Traverse through the array.

        for(int i = 0; i < n; i++) {

            // Use array indices to store visited state of each element.

            int index = Math.abs(arr.get(i)) - 1;

            // Mark as visited by multiplying with '-1'.

            arr.set(index, arr.get(index) \* -1);

            // In case of duplicate, this will become +ve.

            if(arr.get(index) > 0){

                // Return duplicate element.

                return Math.abs(arr.get(i));

            }

        }

        return -1;

    }

}

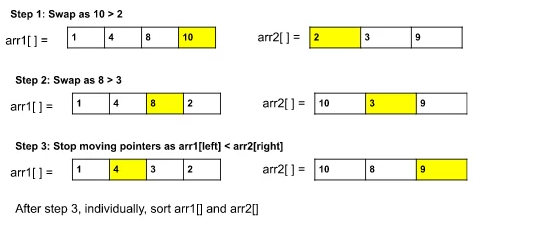
**Merge 2 sorted arrays without using Extra space**

Ap 1 :

1. Initialize a pointer to last element of 1st array and nother at the first element of 2nd array
2. Check if arr1[I]>arr2[j]
   1. Swap until arr1[left] is greater than arr2[right]
   2. increase pointers accordingly
3. Sort two arrays

**TC : O(nlog(n))**

**SC : O(1)**



// Declare 2 pointers:

int left = n - 1;

int right = 0;

// Swap the elements until arr1[left] is

// smaller than arr2[right]:

while (left >= 0 && right < m) {

    if (arr1[left] > arr2[right]) {

        long temp = arr1[left];

        arr1[left] = arr2[right];

        arr2[right] = temp;

        left--;

        right++;

    } else {

        break;

    }

}

// Sort arr1[] and arr2[] individually:

Arrays.sort(arr1);

Arrays.sort(arr2);

**Ap 2 : GAP method**

This gap method is based on a sorting technique called shell sort.

1. First, assume the two arrays as a single array and calculate the gap value i.e. ceil((size of arr1[] + size of arr2[]) / 2).
2. We will perform the following operations for each gap until the value of the gap becomes 0:
   1. Place two pointers in their correct position like the left pointer at index 0 and the right pointer at index (left+gap).
   2. Again we will run a loop until the right pointer reaches the end i.e. (n+m). Inside the loop, there will be 3 different cases:
      1. ****If the left pointer is inside arr1[] and the right pointer is in arr2[]:****We will compare arr1[left] and arr2[right-n] and swap them if arr1[left] > arr2[right-n].
      2. ****If both the pointers are in arr2[]:****We will compare arr1[left-n] and arr2[right-n] and swap them if arr1[left-n] > arr2[right-n].
      3. ****If both the pointers are in arr1[]:****We will compare arr1[left] and arr2[right] and swap them if arr1[left] > arr2[right].
   3. After the right pointer reaches the end, we will decrease the value of the gap and it will become ceil(current gap / 2).
3. Finally, after performing all the operations, we will get the merged sorted array.

public static void swapIfGreater(long[] arr1, long[] arr2, int ind1, int ind2) {

    if (arr1[ind1] > arr2[ind2]) {

        long temp = arr1[ind1];

        arr1[ind1] = arr2[ind2];

        arr2[ind2] = temp;

    }

}

public static void merge(long[] arr1, long[] arr2, int n, int m) {

    // len of the imaginary single array:

    int len = n + m;

    // Initial gap:

    int gap = (len / 2) + (len % 2);

    while (gap > 0) {

        // Place 2 pointers:

        int left = 0;

        int right = left + gap;

        while (right < len) {

            // case 1: left in arr1[]

            //and right in arr2[]:

            if (left < n && right >= n) {

                swapIfGreater(arr1, arr2, left, right - n);

            }

            // case 2: both pointers in arr2[]:

            else if (left >= n) {

                swapIfGreater(arr2, arr2, left - n, right - n);

            }

            // case 3: both pointers in arr1[]:

            else {

                swapIfGreater(arr1, arr1, left, right);

            }

            left++; right++;

        }

        // break if iteration gap=1 is completed:

        if (gap == 1) break;

        // Otherwise, calculate new gap:

        gap = (gap / 2) + (gap % 2);

    }

}

**Merge Intervals**

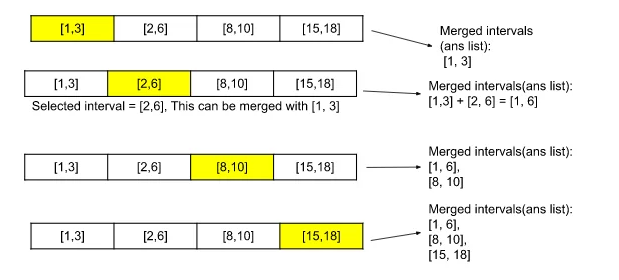
We will start traversing the given array with a single loop. At the first index, as our answer list is empty, we will insert the first element into the answer list. While traversing afterward we can find two different cases:

****Case 1: If the current interval can be merged with the last inserted interval of the answer list:****In this case, we will update the end of the last inserted interval with the ***maximum(current interval’s end, last inserted interval’s end)***and continue moving afterward***.***

****Case 2: If the current interval cannot be merged with the last inserted interval of the answer list:****In this case, we will insert the current interval in the answer array as it is.

TC : O(nlog(n))

SC : O(n)



class Solution {

    // Method to merge overlapping intervals

    public int[][] overlappedInterval(int[][] intervals) {

        // Get the number of intervals

        int n = intervals.length;

        // Sort the intervals based on the start time using custom comparator

        Arrays.sort(intervals, new Comparator<int[]>() {

            public int compare(int[] a, int[] b) {

                // Compare based on the start time (first element of each interval)

                return a[0] - b[0];

            }

        });

        // List to store the merged intervals

        List<int[]> ans = new ArrayList<>();

        // Iterate through each interval

        for (int i = 0; i < n; i++) {

            // If the merged interval list is empty or the current interval does not overlap with the last interval in the list

            if (ans.size() == 0 || ans.get(ans.size() - 1)[1] < intervals[i][0]) {

                // Add the current interval to the list

                ans.add(new int[]{intervals[i][0], intervals[i][1]});

            } else {

                // Merge the current interval with the last interval in the list

                ans.get(ans.size() - 1)[1] = Math.max(ans.get(ans.size() - 1)[1], intervals[i][1]);

            }

        }

        // Convert the list of merged intervals to a 2D array and return

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

        arr = ans.toArray(arr);

        return arr;

    }

}

**Next Permutation**

**Ap 1 : brute force**

Step 1: Find all possible permutations of elements present and store them.

Step 2: Search input from all possible permutations.

Step 3: Print the next permutation present right after it.

TC : O( n! \* n )

SC : O (1 )

**Ap 2 : optimal**

1. ****Find the break-point, i:****Break-point means the ***first index i from the back of the given array*** where arr[i] becomes smaller than arr[i+1].  
   For example, if the given array is {2,1,5,4,3,0,0}, the break-point will be index 1(*0-based indexing*). Here from the back of the array, index 1 is the first index where arr[1] i.e. 1 is smaller than arr[i+1] i.e. 5.  
   *To find the break-point, using a loop we will traverse the array backward and store the index i where arr[i] is less than the value at index (i+1) i.e. arr[i+1].*
2. ****If such a break-point does not exist i.e. if the array is sorted in decreasing order,****the given permutation is the last one in the sorted order of all possible permutations. So, the next permutation must be the first i.e. the permutation in increasing order.  
   So, ***in this case, we will reverse the whole array and will return it as our answer.***
3. ****If a break-point exists:****
   1. Find the smallest number i.e. > arr[i] and in the right half of index i(i.e. from index i+1 to n-1) and swap it with arr[i].
   2. Reverse the entire right half(i.e. from index i+1 to n-1) of index i. And finally, return the array.

TC : O ( n )

SC : O ( 1 )

int n = A.size(); // size of the array.

        // Step 1: Find the break point:

        int ind = -1; // break point

        for (int i = n - 2; i >= 0; i--) {

            if (A.get(i) < A.get(i + 1)) {

                // index i is the break point

                ind = i;

                break;

            }

        }

        // If break point does not exist:

        if (ind == -1) {

            // reverse the whole array:

            Collections.reverse(A);

            return A;

        }

        // Step 2: Find the next greater element

        //         and swap it with arr[ind]:

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

            if (A.get(i) > A.get(ind)) {

                int tmp = A.get(i);

                A.set(i, A.get(ind));

                A.set(ind, tmp);

                break;

            }

        }

        // Step 3: reverse the right half:

        List<Integer> sublist = A.subList(ind + 1, n);

        Collections.reverse(sublist);

        return A;

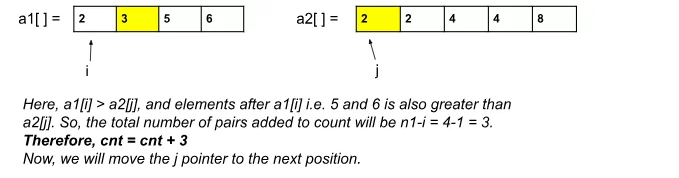
**Count Inversion**

What is an inversion of an array? Definition: for all i & j < size of array, if i < j then you have to find pair (A[i],A[j]) such that A[j] < A[i].

Ap 1 : brute force

Nested loops

Ap 2 : using merge sort



Both the arrays are sorted. So, if a element from left arr is greater than a lement of right array then all the rest elements of 1st array will also be greater than element of right array.

TC : O(nlog(n))

SC : O( n) ; if we don’t want to change the original array, we are keeping an extra array to store elements sorted manner.

private static int merge(int[] arr, int low, int mid, int high) {

    ArrayList<Integer> temp = new ArrayList<>(); // temporary array

    int left = low;      // starting index of left half of arr

    int right = mid + 1;   // starting index of right half of arr

    //Modification 1: cnt variable to count the pairs:

    int cnt = 0;

    //storing elements in the temporary array in a sorted manner//

    while (left <= mid && right <= high) {

        if (arr[left] <= arr[right]) {

            temp.add(arr[left]);

            left++;

        } else {

            temp.add(arr[right]);

            cnt += (mid - left + 1); //Modification 2

            right++;

        }

    }

    // if elements on the left half are still left //

    while (left <= mid) {

        temp.add(arr[left]);

        left++;

    }

    //  if elements on the right half are still left //

    while (right <= high) {

        temp.add(arr[right]);

        right++;

    }

    // transfering all elements from temporary to arr //

    for (int i = low; i <= high; i++) {

        arr[i] = temp.get(i - low);

    }

    return cnt; // Modification 3

}

public static int mergeSort(int[] arr, int low, int high) {

    int cnt = 0;

    if (low >= high) return cnt;

    int mid = (low + high) / 2 ;

    cnt += mergeSort(arr, low, mid);  // left half

    cnt += mergeSort(arr, mid + 1, high); // right half

    cnt += merge(arr, low, mid, high);  // merging sorted halves

    return cnt;

}

public static int numberOfInversions(int[] a, int n) {

    // Count the number of pairs:

    return mergeSort(a, 0, n - 1);

}

**Best time to buy and Sell stock**

* Create a variable maxPro and store 0 initially.
* Create a variable minPrice and store some larger value(ex: MAX\_VALUE) value initially.
* Run a for loop from 0 to n.
* Update the minPrice if it is greater than the current element of the array
* Take the difference of the minPrice with the current element of the array and compare and maintain it in maxPro.
* Return the maxPro.

public int maxProfit(int[] prices) {

        int minTillNow = Integer.MAX\_VALUE;

        int maxPr = 0;

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

            minTillNow = Math.min(minTillNow, prices[i]);

            maxPr = Math.max(maxPr, prices[i]-minTillNow);

        }

        return maxPr;

    }

**Smallest Subarray with sum greater than a given value**

public static int smallestSubWithSum(int a[], int n, int x) {

    // Initialize the answer with the maximum possible value

    int ans = Integer.MAX\_VALUE;

    // Initialize two pointers i and j

    int i = 0, j = 0;

    // Initialize the sum of the current subarray

    int sum = 0;

    // Loop until either pointer exceeds the array bounds

    while (i <= j && j < n) {

        // Expand the subarray to the right while the sum is less than or equal to x

        while (sum <= x && j < n) {

            sum += a[j++];

        }

        // If the current subarray sum is greater than x, update the answer

        while (sum > x && i < j) {

            // Update the answer with the minimum length of subarray found so far

            ans = Math.min(ans, j - i);

            // Shrink the subarray from the left

            sum -= a[i++];

        }

    }

    // If ans is still Integer.MAX\_VALUE, it means no subarray satisfies the condition

    if (ans == Integer.MAX\_VALUE) {

        return 0; // or any other suitable value indicating no valid subarray found

    }

    return ans;

}

**common elements in 3 sorted arrays**

**Ap 1 : using binary search**

1. Iterating over 1st array
   1. Checking curr element is present in second array
   2. If present check for in third array if present
   3. If present then add to ans

public static ArrayList<Integer> findCommonElements(ArrayList<Integer> a, ArrayList<Integer> b, ArrayList<Integer> c)

    {

        // Getting the size of the first array

        int n = a.size();

        // Defining the vector to store common elements

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

        // Iterating through the first array

        for (int i = 0; i < n; i++)

        {

            // Defining the flag variables

            int isPresentInB = 0, isPresentInC = 0;

            // Checking if the current element is present in second array using binary search

            isPresentInB = binarySearch(b , a.get(i));

            // Checking if the current element was not found in second array

            if (isPresentInB == 0)

            {

                continue;

            }

            // Checking if the current element is present in third array using binary search

            isPresentInC = binarySearch(c, a.get(i));

            // Checking if the current element was found in third array

            if (isPresentInC > 0)

            {

                // Adding the current element to output array if it is already not present

                int temp = a.get(i);

                if (ans.size() == 0 || ans.get(ans.size() - 1) != temp)

                {

                    ans.add(a.get(i));

                }

            }

        }

        // Returning the final output array

        return ans;

    }

**Ap 2 : 3 pointer**

public static ArrayList<Integer> commonElements(int A[], int B[], int C[], int n1, int n2, int n3) {

    ArrayList<Integer> ans = new ArrayList<Integer>(); // Initialize an ArrayList to store common elements

    int i = 0, j = 0, k = 0; // Initialize pointers for each array

    // Iterate until any of the pointers reach the end of their respective arrays

    while (i < n1 && j < n2 && k < n3) {

        int x = A[i]; // Current element from array A

        int y = B[j]; // Current element from array B

        int z = C[k]; // Current element from array C

        // If all three elements are equal, add to the result list

        if (x == y && y == z) {

            // Check if the element is already present in the result list

            if (ans.size() == 0 || ans.get(ans.size() - 1) != x) {

                ans.add(x); // Add the element to the result list

            }

            i++; // Move to the next element in array A

            j++; // Move to the next element in array B

            k++; // Move to the next element in array C

        } else if (x <= y && x <= z) {

            i++; // Move to the next element in array A

        } else if (y <= x && y <= z) {

            j++; // Move to the next element in array B

        } else if (z <= x && z <= y) {

            k++; // Move to the next element in array C

        }

    }

    return ans; // Return the list of common elements

}

**Rearrange the array in alternating positive and negative items with O(1) extra space**

**Ap 1 :**

* Maintain a list of positives that store all the positive integers from the array.
* Maintain another list of negatives which store all the negative integers from the array.
* Update the array alternatively.

TC : O(n)

SC : O(n)

public static void rearrange(int[] arr)

    {

        // This will store all positive integers.

        ArrayList<Integer> positives = new ArrayList();

        // This will store all negative integers.

        ArrayList<Integer> negatives = new ArrayList();

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

        {

            if (arr[i] < 0)

            {

                negatives.add(arr[i]);

            }

            else

            {

                positives.add(arr[i]);

            }

        }

        int index = 0;

        int i = 0, j = 0;

        // Fill equal number of positives and negatives.

        while (i < positives.size() && j < negatives.size())

        {

            if (index % 2 == 0)

            {

                arr[index] = negatives.get(j);

                j++;

            }

            else

            {

                arr[index] = positives.get(i);

                i++;

            }

            index++;

        }

        // If positive integers are still left.

        while (i < positives.size())

        {

            arr[index] = positives.get(i);

            index++;

            i++;

        }

        // If negative integers are still left.

        while (j < negatives.size())

        {

            arr[index] = negatives.get(j);

            index++;

            j++;

        }

        return;

    }

**Ap 2 : modified quick sort**

**partition(arr, size):**

* Initialize I and J with 0.
  + While J < size:
    - If arr[I] < 0: swap arr[J] and arr[I], increment I with 1.
    - Increment J with 1.
* Return I, this is the index where the left side contains all negatives.

**rearrange(arr , size):**

* positive = partition(arr, size), the index of the first positive integer.
* Update negative with 1, keeping negative integer index 0 and starting first positive integer from index 1.
* Condition 1: positive and negative are in range.
* Condition 2: negative is less than positive.
* Condition 3: negative index contains a negative integer only, if not, this means we have arranged all the negative integers.
* while(all the above conditions satisfy):
  + Swap (positive index element with negative index)
  + Increment negative by 2.
  + Increment positive by 1.

public static void rearrange(int[] arr)

    {

        // Finding the point where negatives and positives seperate.

        int division = pivot(arr);

        // Positive element's index.

        int pos = division;

        // Negative element's index.

        int neg = 1;

        // 1. Positive index is in range.

        // 2. Negative index is less than positive index.

        // 3. Negative index contains negative  element only.

        while (pos < arr.length && neg < pos && arr[neg] < 0)

        {

            swap(arr, pos, neg);

            neg += 2;

            pos++;

        }

        return;

    }

    public static void swap(int arr[], int i, int j)

    {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

    public static int pivot(int[] arr)

    {

        int i = 0;

        int j = 0;

        while (i < arr.length)

        {

            if (arr[i] < 0)

            {

                swap(arr, i, j);

                j++;

            }

            i++;

        }

        return j;

}

**Find factorial of a large number**

A factorial of 100 has 158 digits. It is not possible to store these many digits even if we use long int.

static void multiply(int n, ArrayList<Integer> ans){

    int carry =0;

    for(int i=0; i<ans.size(); i++){

        // Multiply each digit of the current answer by n and add the carry

        int num = n\*ans.get(i);

        // Update the digit at index i in the answer list with the new value

        ans.set(i, (carry+num)%10); // Store only the unit digit

        // Update carry for the next iteration

        carry=(num+carry)/10; // Calculate the carry for the next digit

    }

    // If there is any remaining carry, add it to the answer

    while(carry!=0){

        ans.add(carry%10); // Add the remaining carry as a new digit to the answer

        carry/=10; // Reduce the carry for further processing

    }

}

static ArrayList<Integer> factorial(int N){

    // Initialize the answer list with a single digit representing 1

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

    ans.add(1);

    // Iterate from 2 to N

    for(int i=2; i<=N; i++){

        // Multiply the current number (i) with the answer so far

        multiply(i,ans);

    }

    // Reverse the answer list to get the correct order of digits

    Collections.reverse(ans);

    // Return the factorial as an ArrayList of integers

    return ans;

}

**TC : O(n^2)**

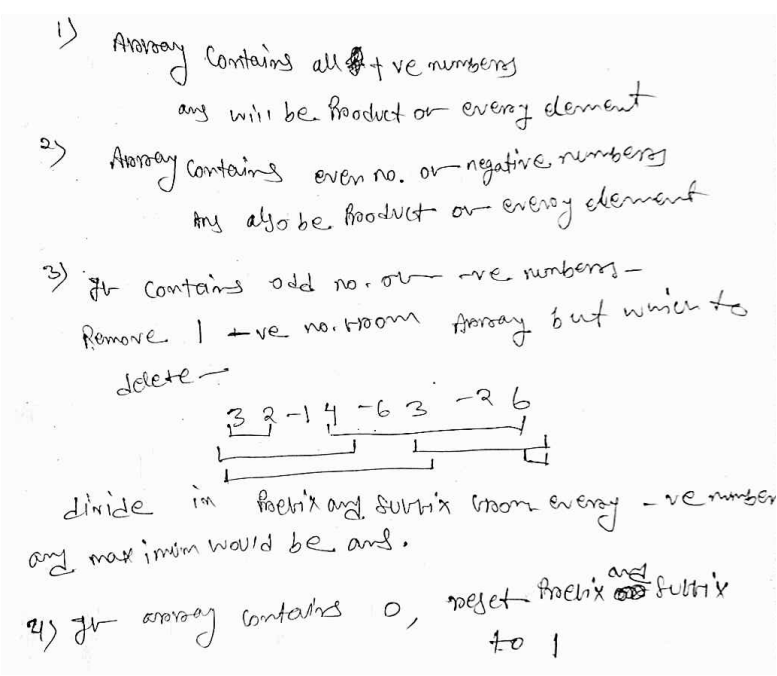
**SC : O(n) ; for storing ans only**

**maximum product subarray**

**Ap 1 : nested loop**

TC : O(n^2)

**Ap 2 : intuition based**



1. We will first declare 2 variables i.e. ‘pre’(*stores the product of the prefix subarray*) and ‘suff’(*stores the product of the suffix subarray*). They both will be initialized with 1(*as we want to store the product*).
2. Now, we will use a loop(say ****i****) that will run from 0 to n-1.
3. We have to check 2 cases to handle the presence of 0:
   1. ****If pre = 0:****This means the previous element was 0. So, we will consider the current element as a part of the new subarray. So, we will set ‘pre’ to 1.
   2. ****If suff = 0:****This means the previous element was 0 in the suffix. So, we will consider the current element as a part of the new suffix subarray. So, we will set ‘suff’ to 1.
4. Next, we will multiply the elements from the starting index with ‘pre’ and the elements from the end with ‘suff’. To incorporate both cases inside a single loop, we will do the following:
   1. We will multiply arr[i] with ‘pre’ i.e. pre \*= arr[i].
   2. We will multiply arr[n-i-1] with ‘suff’ i.e. suff \*= arr[n-i-1].
5. After each iteration, we will consider the maximum among the previous answer, ‘pre’ and ‘suff’ i.e. max(previous\_answer, pre, suff).
6. Finally, we will return the maximum product.

public static int subarrayWithMaxProduct(int[] arr) {

    // Initialize variables to store the maximum product,

    // and prefix and suffix products.

    int maxProduct = Integer.MIN\_VALUE;

    int prefix = 1; // Product of elements from the beginning

    int suffix = 1; // Product of elements from the end

    // Loop through the array

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

        // If prefix becomes 0, reset it to 1

        if (prefix == 0) {

            prefix = 1;

        }

        // If suffix becomes 0, reset it to 1

        if (suffix == 0) {

            suffix = 1;

        }

        // Calculate prefix product by multiplying the current element

        prefix \*= arr[i];

        // Calculate suffix product by multiplying the current element

        // from the end of the array (arr.length - i - 1)

        suffix \*= arr[arr.length - i - 1];

        // Update maxProduct with the maximum of maxProduct, prefix, and suffix

        maxProduct = Math.max(maxProduct, Math.max(prefix, suffix));

    }

    // Return the maximum product

    return maxProduct;

}

**TC : O(n)**

**SC : O(1)**

**longest coinsecutive subsequence**

**Ap 1 : sorting**

* We will consider 3 variables,
  + ****‘lastSmaller’ →****(*to store the last included element of the current sequence*),
  + ****‘cnt’ →****(*to store the length of the current sequence*),
  + ****‘longest’ →****(*to store the maximum length*).

public static int longestSuccessiveElements(int []a) {

        int n = a.length;

        if (n == 0) return 0;

        //sort the array:

        Arrays.sort(a);

        int lastSmaller = Integer.MIN\_VALUE;

        int cnt = 0;

        int longest = 1;

        //find longest sequence:

        for (int i = 0; i < n; i++) {

            if (a[i] - 1 == lastSmaller) {

                //a[i] is the next element of the

                //current sequence.

                cnt += 1;

                lastSmaller = a[i];

            } else if (a[i] != lastSmaller) {

                cnt = 1;

                lastSmaller = a[i];

            }

            longest = Math.max(longest, cnt);

        }

        return longest;

    }

**Ap 2 : using HashSet**

1. First, we will put all the array elements into the set data structure.
2. For every element, x, that can be a starting number(*i.e. x-1 does not exist in the set*) we will do the following:
   1. We will set the length of the current sequence(****cnt****) to 1.
   2. Then, again using the set, we will search for the consecutive elements such as x+1, x+2, and so on, and find the maximum possible length of the current sequence. This length will be stored in the variable ‘cnt’.
   3. After that, we will compare ‘cnt’ and ‘longest’ and update the variable ‘longest’ with the maximum value (i.e. ****longest = max(longest, cnt)****).
3. Finally, we will have the answer i.e. ‘longest’.

public static int longestSuccessiveElements(int []a) {

    // Check if the array is empty

    if(a.length == 0) return 0;

    // At least 1 element would be there

    int ans = 1;

    // Create a HashSet to store unique elements of the array

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

    // Add all elements of the array to the HashSet

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

        s.add(a[i]);

    }

    // Iterate through each unique element in the HashSet

    for(int x : s) {

        // Check if the previous element is not present in the HashSet

        if(!s.contains(x - 1)) {

            // If not present, it indicates the start of a new consecutive sequence

            int count = 1;

            // Count consecutive elements by iterating forward in the sequence

            while(s.contains(x + 1)) {

                x = x + 1;

                count++;

            }

            // Update the answer with the maximum consecutive count found

            ans = Math.max(ans, count);

        }

    }

    return ans;

}

****Time Complexity:****O(N) + O(2\*N) ~ O(3\*N), where N = size of the array.  
****Reason:****O(N) for putting all the elements into the set data structure. After that for every starting element, we are finding the consecutive elements. Though we are using nested loops, the set will be traversed at most twice in the worst case. So, the time complexity is O(2\*N) instead of O(N2).

****Space Complexity:****O(N), as we are using the set data structure to solve this problem.

****Note:**** The time complexity is computed under the assumption that we are using unordered\_set and it is taking O(1) for the set operations.

* If we consider the worst case the set operations will take O(N) in that case and the total time complexity will be approximately O(N2).
* And if we use the set instead of unordered\_set, the time complexity for the set operations will be O(logN) and the total time complexity will be O(NlogN).

**find all elements that appear more than " n/k " times.**

**Ap 1: sorting**

First sort the elements then count the frequency of distinct elements by checking its adjacent elements are equal or not if the frequency of elements is greater than n/k then, print the array element.

**TC : O(nlog(n))**

**SC : O(1)**

**Ap 2 : using maps**

Take an unordered map and traverse through the array and keep incrementing the value in the map every time that key appears in the array. At last, compare it with a variable comp which stores the value of n/k. And keep storing the answer in a dynamic array or directly printing it.

int comp = n / k;

HashMap<Integer, Integer > map=new HashMap<>();

for (int i = 0; i < n; i++) {

    map.put(arr[i], map.getOrDefault(arr[i],0)+1);

}

for(int x:map.keySet()){

    if(map.get(x)>comp)

        System.out.print(x+" ");

}

**Chocolate Distribution problem**

public long findMinDiff (ArrayList<Integer> a, int n, int m)

{

    // Sort the ArrayList in ascending order

    Collections.sort(a);

    // Initialize a variable to store the minimum difference

    long ans = Integer.MAX\_VALUE;

    // Iterate through the sorted ArrayList from index 0 to n-m

    for(int i = 0; i + m - 1 < n; i++){

        // Update the minimum difference by comparing the difference between the current element at index i+m-1

        // and the element at index i with the current minimum difference

        ans = Math.min(ans, a.get(i + m - 1) - a.get(i));

    }

    // Return the minimum difference

    return ans;

}

**Three way partitioning of an array around a given value**

1. Initialise 2 variables start=0 and end=n-1 which stores the left-most and right-most available space.
2. Start iterating the array from left until i<=end.
3. If the current element is less than ****a**** then swap the current element with the element at index start and move to the next element in the array. Also, increment start.
4. If the current element is greater than ****b**** then swap the current element with the element at the index end. Decrement end.
5. Else move to the next element in the array.

int l = 0; // Initialize left pointer

int r = array.length - 1; // Initialize right pointer

// Loop through the array from left to right

for (int i = 0; i <= r; i++) {

    // If the current element is less than 'a'

    if (array[i] < a) {

        // Swap the current element with the element at index 'l'

        int t = array[i];

        array[i] = array[l];

        array[l] = t;

        // Move the left pointer to the right

        l++;

    }

    // If the current element is greater than 'b'

    else if (array[i] > b) {

        // Swap the current element with the element at index 'r'

        int t = array[i];

        array[i] = array[r];

        array[r] = t;

        // Move the right pointer to the left

        r--;

        // Since the current element at index 'i' is swapped with a new element,

        // decrement 'i' to recheck the swapped element

        i--;

    }

}

**Find whether an array is a subset of another array**

**Ap 1 : nested loops**

We can use two nested loops. The outer loop iterates over the elements of arr1[] and the inner loop checks for that element in arr2[] by simple linear search.

**TC : O(M\*N)**

**SC : O(1)**

**Ap 2 : sorting and binary search**

We first sort the arr2[] array and then we can use set a loop to traverse the elements of arr1[]  and search for them in arr2[] using binary search( as arr2[] is sorted)

static boolean bSearch(int elem, int arr[], int n) {

    int start = 0;

    int end = n - 1;

    while (start <= end) {

      int mid = (start + end) / 2;

      if (arr[mid] == elem)

        return true;

      else if (arr[mid] < elem)

        start = mid + 1;

      else end = mid - 1;

    }

    return false;

  }

  static boolean isSubset(int arr1[], int m, int arr2[], int n) {

    if (m > n) return false;

    Arrays.sort(arr2);

    for (int i = 0; i < m; i++) {

      boolean present = bSearch(arr1[i], arr2, n);

      if (present == false) return false;

    }

    return true;

  }

**ap 3 : HashMap**

1. If the length of ARR2 is greater than ARR1 then return false, as ARR2 can’t be a subset.
2. Create a map for storing the frequency of elements of ARR1.
3. Iterate array ARR1, and update the frequency of ARR1.
4. Now, iterate through ARR2 and check if the current element of ARR2 is in the map or not, it is not in the map then returns false.
5. Otherwise, decrease the frequency of the current element by 1 and check if the updated frequency is positive or not, if it is negative then return false.
6. If the whole ARR2 is traversed, then return true.

 public static boolean checkSubset(int[] arr1, int[] arr2, int n, int m) {

        // If arr2 is larger, it can't be subset.

        if (m > n) {

            return false;

        }

        // Map to store the frequency of each elements of array arr1.

        HashMap<Integer, Integer> map = new HashMap<>();

        // Store the element of arr1 with its frequency in map.

        for (int i = 0; i < n; i++) {

            if (map.containsKey(arr1[i])) {

                map.put(arr1[i], map.get(arr1[i]) + 1);

            }

            else {

                map.put(arr1[i], 1);

            }

        }

        for (int i = 0; i < m; i++) {

            // If arr2[i] is not in map then return false.

            if (!map.containsKey(arr2[i])) {

                return false;

            }

            // Else decrease its frequency by 1.

            else {

                map.put(arr2[i], map.get(arr2[i]) - 1);

                // If at any point, frequency becomes negative then return false.

                if (map.get(arr2[i]) < 0) {

                    return false;

                }

            }

        }

        return true;

    }

**Pair Sum**

**ap 1: brute force**

1. For each element in the array 'ARR[i]', check if ('ARR[i]' + ‘ARR[j]’), equals to given sum or not, where ‘i’ < ‘j’ < ‘N’.
2. If the condition matches, add the pair('ARR[i]', ‘ARR[j]’) to the list.

Time complexity: O( n^2 ) Space complexity: O( 1 )

**Ap 2: using 2-pointers (optimal)**

1. while left < right
   1. curSum = arr[left] + arr[right]
      1. If curSum==targetSum
         1. Add pair && increment left and right

Ii . If curSum>target

1. Decrement right

Iii. If curSum<target

1. Increment left

Time complexity: O( n ) Space complexity: O( 1 )

public static List<int[]> pairSum(int[] arr, int s) {

       List<int[]> result = new ArrayList<>();

        // Sort the array

        Arrays.sort(arr);

        int left = 0;

        int right = arr.length - 1;

        while (left < right) {

            int currentSum = arr[left] + arr[right];

            if (currentSum == s) {

                // Add the pair to the result list

                result.add(new int[]{arr[left], arr[right]});

                // Move pointers to find other pairs

                left++;

                right--;

            } else if (currentSum < s) {

                // Increase the sum by moving the left pointer to the right

                left++;

            } else {

                // Decrease the sum by moving the right pointer to the left

                right--;

            }

        }

        return result;

    }

**3Sum**

**Ap 1: 2 pointers (optimal)**

1. Now since we want triplets such that x + y + z = ‘K’, we have x+ y = ‘K’ - z and now 2. we can fix z as arr[i]. So we want to find the sum of two numbers x and y as ‘K’ - arr[i] in the array.
2. Let us assume that we are the ith index of the array and initialise variable target to ‘K’ - ‘ARR[i]’. So now we just need to find two elements x, y such that target = x + y.
3. Let the two pointers be ‘FRONT’ and ‘BACK’, where ‘FRONT’ = i + 1 and ‘BACK’ = n - 1. Let ‘SUM’ = x + y, where x = ‘ARR[FRONT]’ and y = ‘ARR[BACK]’.
4. We have to find the triplets such that ‘TARGET’ = ‘SUM’.

While ‘FRONT’ < ‘BACK’, there will be 3 cases:

1. if ('SUM' < ‘TARGET’), increment front pointer
2. Else if ('SUM' > ‘TARGET’), decrease the ‘BACK’ pointer.
3. Else print the triplet and
4. since we want distinct triplets, do the following.

Increment the front pointer until ‘ARR[FRONT]’ = x and ‘FRONT’ < ‘BACK’.

Decrement the back pointer until ‘ARR[BACK]’ = y and ‘FRONT’ < ‘BACK’.

While ‘ARR[i]’ = ‘ARR[i+1]’, keep on incrementing i, this will automatically ensure that we are only finding distinct triplets.

Time complexity: O( N^2 ) Space complexity: O( 1 )

public static ArrayList<ArrayList<Integer>> findTriplets(int[] arr, int n, int k) {

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

        // Sorting the arraylist.

        Arrays.sort(arr);

        for (int i = 0; i < n; i++) {

            int target = k-arr[i];

            int front = i + 1;

            int back = n - 1;

            while (front < back) {

                int sum = arr[front] + arr[back];

                // Finding answer which starts from arr[i].

                if (sum < target) {

                    front++;

                }

                else if (sum > target) {

                    back--;

                }

                else {

                    int x = arr[front];

                    int y = arr[back];

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

                    list.add(arr[i]);

                    list.add(arr[front]);

                    list.add(arr[back]);

                    ans.add(list);

                    // Incrementing front pointer until we reach a different number.

                    while (front < back && arr[front] == x) {

                        front++;

                    }

                    // Decrementing last pointer until we reach a different number.

                    while (front < back && arr[back] == y) {

                        back--;

                    }

                }

            }

            // Ensuring that we don't encounter duplicate values for arr[i].

            while (i + 1 < n && arr[i] == arr[i + 1]) {

                i++;

            }

        }

        return ans;

}

**Minimum no. of operations required to make an array palindrome**

public static int palindrome(int[] A) {

    // Initialize a variable to keep track of the number of operations

    int ans = 0;

    // Initialize two pointers, l pointing to the start of the array and r pointing to the end

    int l = 0, r = A.length - 1;

    // Iterate until the pointers meet or cross each other

    while (l < r) {

        // If the elements at the current positions are equal

        if (A[l] == A[r]) {

            // Move the pointers towards the center of the array

            l++;

            r--;

        }

        // If the element at index l is greater than the element at index r

        else if (A[l] > A[r]) {

            // Move the right pointer towards the center and adjust the element at index r

            r--;

            A[r] += A[r + 1]; // Add the value of the next element to make A[r] equal to A[l]

            ans++; // Increment the number of operations

        }

        // If the element at index l is less than the element at index r

        else if (A[l] < A[r]) {

            // Move the left pointer towards the center and adjust the element at index l

            l++;

            A[l] += A[l - 1]; // Add the value of the previous element to make A[l] equal to A[r]

            ans++; // Increment the number of operations

        }

    }

    // Return the total number of operations performed

    return ans;

}

**Minimum swaps required bring elements less equal K together**

public static int minSwap (int arr[], int n, int k) {

        // Count the number of elements less than or equal to k

    int count = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] <= k) {

            count++;

        }

    }

    // Count the number of bad elements (elements greater than k) in the first window of size count

    int bad = 0;

    for (int i = 0; i < count; i++) {

        if (arr[i] > k) {

            bad++;

        }

    }

    // Initialize the minimum number of swaps needed

    int minSwap = bad;

    // Slide the window across the array and update the count of bad elements

    for (int i = 0, j = count; j < n; i++, j++) {

        if (arr[i] > k) {

            bad--;

        }

        if (arr[j] > k) {

            bad++;

        }

        minSwap = Math.min(minSwap, bad);

    }

    return minSwap;

}

# Length of the longest subarray with zero Sum

Now let’s say we know that the sum of subarray(i, j) = S, and we also know that the sum of subarray(i, x) = S where i < x < j. We can conclude that the sum of subarray(x+1, j) = 0.

1. First, let us initialize a variable say ****sum**** ****= 0**** which stores the sum of elements traversed so far and another variable says max = 0 which stores the length of the longest subarray with sum zero.
2. Declare a HashMap<Integer, Integer> which stores the prefix sum of every element as a key and its index as a value.
3. Now traverse the array, and add the array element to our sum.

 (i)  If sum = 0, then we can say that the subarray until the current index has a sum = 0,      so we update max with the maximum value of (max, current\_index+1)

(ii)  If the sum is not equal to zero then we check the hashmap if we’ve seen a subarray with this sum before

if HashMap contains sum -> this is where the above-discussed case occurs (subarray with equal sum), so we update our max

else -> Insert (sum, current\_index) into hashmap to store prefix sum until the current index

1. After traversing the entire array our max variable has the length of the longest substring having a sum equal to zero, so return max.

**TC : O(nlog(n)) ; n for number of elements and log(n) for searching from hashmap**

**SC : O(n) ; hashmap**

public static int getLongestZeroSumSubarrayLength(int[] arr) {

    // Initialize variables

    int maxLength = 0; // This will store the length of the longest subarray with zero sum

    int sum = 0; // This will keep track of the cumulative sum of elements

    HashMap<Integer, Integer> map = new HashMap<>(); // This HashMap will store the cumulative sum and its corresponding index

    // Iterate through the array

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

        sum += arr[i]; // Add the current element to the sum

        // If the sum is zero from the beginning of the array till the current index

        if (sum == 0) {

            maxLength = i + 1; // Update the maxLength to current index + 1

        } else {

            // If the sum is not zero

            if (map.containsKey(sum)) {

                // If the sum has been encountered before, calculate the length of the subarray with zero sum

                // by subtracting the index of the previous occurrence of the same sum from the current index

                maxLength = Math.max(maxLength, i - map.get(sum));

            } else {

                // If the sum is encountered for the first time, store the index in the map

                map.put(sum, i);

            }

        }

    }

    return maxLength; // Return the length of the longest subarray with zero sum

}

**----------------HARD--------**

**Maximum profit by buying and selling a share atmost twice**

public static int maxProfit(int[] price, int n) {

    // Array to store the profit for each day

    int[] profit = new int[n];

    // Initialize all elements of the profit array to 0

    for (int i = 0; i < n; i++)

        profit[i] = 0;

    // Calculate the maximum price encountered from right to left

    int max\_price = price[n - 1];

    for (int i = n - 2; i >= 0; i--) {

        // Update the maximum price if a higher price is found

        if (price[i] > max\_price)

            max\_price = price[i];

        // Calculate the profit for the current day based on the maximum price

        profit[i] = Math.max(profit[i + 1], max\_price - price[i]);

    }

    // Calculate the maximum profit considering the second transaction

    int min\_price = price[0];

    for (int i = 1; i < n; i++) {

        // Update the minimum price if a lower price is found

        if (price[i] < min\_price)

            min\_price = price[i];

        // Update the profit for the current day by considering both transactions

        profit[i] = Math.max(profit[i - 1], profit[i] + (price[i] - min\_price));

    }

    // The final element of the profit array contains the maximum profit achievable

    int result = profit[n - 1];

    return result;

}

**Trapping Rain water problem**

**Ap 1 : brute force**

For each index, we have to find the amount of water that can be stored and we have to sum it up.If we observe carefully the amount the water stored at a particular index is the minimum of maximum elevation to the left and right of the index minus the elevation at that index.

**TC : O(n\*n)**

**SC : O(1)**

static int trap(int[] arr) {

    // Get the length of the input array

    int n = arr.length;

    // Initialize the variable to store the total amount of water trapped

    int waterTrapped = 0;

    // Loop through each element in the array

    for (int i = 0; i < n; i++) {

        int j = i;

        // Initialize variables to store the maximum height of the walls to the left and right of the current element

        int leftMax = 0, rightMax = 0;

        // Find the maximum height of the walls to the left of the current element

        while (j >= 0) {

            leftMax = Math.max(leftMax, arr[j]);

            j--;

        }

        // Reset the pointer j to the current element

        j = i;

        // Find the maximum height of the walls to the right of the current element

        while (j < n) {

            rightMax = Math.max(rightMax, arr[j]);

            j++;

        }

        // Calculate the amount of water that can be trapped at the current element

        // by taking the minimum of the maximum heights of the walls to the left and right,

        // and subtracting the height of the current element

        waterTrapped += Math.min(leftMax, rightMax) - arr[i];

    }

    // Return the total amount of water trapped

    return waterTrapped;

}

**Ap 2 : two pointer**

 Take 2 pointers l(left pointer) and r(right pointer) pointing to 0th and (n-1)th index respectively. Take two variables leftMax and rightMax and initialize them to 0. If height[l] is less than or equal to height[r] then if leftMax is less than height[l] update leftMax to height[l] else add leftMax-height[l] to your final answer and move the l pointer to the right i.e l++. If height[r] is less than height[l], then now we are dealing with the right block. If height[r] is greater than rightMax, then update rightMax to height[r] else add rightMax-height[r] to the final answer. Now move r to the left. Repeat these steps till l and r crosses each other.

****Intuition:**** We need a minimum of leftMax and rightMax.So if we take the case when height[l]<=height[r] we increase l++, so we can surely say that there is a block with a height more than height[l] to the right of l. And for the same reason when height[r]<=height[l] we can surely say that there is a block to the left of r which is at least of height[r].

**TC : O(n)**

**SC : O(1)**

public int trap(int[] height) {

    // Initialize variables to store the result (total water trapped),

    // and the maximum height of walls to the left and right

    int res = 0;

    int leftMax = 0, rightMax = 0;

    // Initialize two pointers, one at the beginning and one at the end of the array

    int l = 0;

    int r = height.length - 1;

    // Loop until the two pointers meet

    while (l < r) {

        // If the height of the wall at the left pointer is less than or equal to the height of the wall at the right pointer

        if (height[l] <= height[r]) {

            // Update the left maximum height if the current height is greater

            if (height[l] > leftMax)

                leftMax = height[l];

            // Otherwise, calculate the trapped water by subtracting the current height from the left maximum height

            else

                res = res + (leftMax - height[l]);

            // Move the left pointer to the right

            l++;

        }

        // If the height of the wall at the left pointer is greater than the height of the wall at the right pointer

        else {

            // Update the right maximum height if the current height is greater

            if (height[r] > rightMax)

                rightMax = height[r];

            // Otherwise, calculate the trapped water by subtracting the current height from the right maximum height

            else

                res = res + (rightMax - height[r]);

            // Move the right pointer to the left

            r--;

        }

    }

    // Return the total amount of water trapped

    return res;

}

**Median of 2 sorted arrays of equal size**

**Median of 2 sorted arrays of different size**

**Ap 1 :**

1. We will call the required indices as ****ind2 =****(n1+n2)/2 and ****ind1 =****((n1+n2)/2)-1. Now we will declare the counter called ‘cnt’ and initialize it with 0.
2. Now, as usual, we will take two pointers i and j, where i points to the first element of arr1[] and j points to the first element of arr2[].
3. Next, using a while loop( while(i < n1 && j < n2)), we will select two elements i.e. arr1[i] and arr2[j], and consider the smallest one among the two. Then, we will increase that specific pointer by 1.  
   In addition to that, in each iteration, we will check if the counter ‘cnt’ hits the indices ****ind1 or ind2.****when ‘cnt’ reaches either index ind1 or ind2, we will store that particular element. We will also increase the ‘cnt’ by 1 every time regardless of matching the conditions.
   1. ****If arr1[i] < arr2[j]:****Check ‘cnt’ to perform necessary operations and increase i and ‘cnt’ by 1.
   2. ****Otherwise:****Check ‘cnt’ to perform necessary operations and increase j and ‘cnt’ by 1.
4. After that, the left-out elements from both arrays will be copied as it is into the third array. While copying we will again check the above-said conditions for the counter, ‘cnt’ and increase it by 1.
5. Now, let’s call the elements at the required indices as ****ind1el(***at ind1***)****and****ind2el(***at ind2***)****:
   1. ****If the total length i.e. (n1+n2) is even:****The median is the average of the two middle elements. median = (ind1el + ind2el) / 2.0.
   2. ****If the total length i.e. (n1+n2) is odd:****median = ind2el.
6. Finally, we will return the value of the median.

public static double median(int[] a, int[] b) {

    // Calculate the lengths of arrays a and b

    int n1 = a.length;

    int n2 = b.length;

    // Calculate the total length of the merged array

    int n = n1 + n2;

    // Calculate the indices for the median elements

    int ind2 = n / 2;

    int ind1 = ind2 - 1;

    // Variables to store the median elements

    int ind1Ele = -1, ind2Ele = -1;

    // Counter to keep track of elements processed

    int cnt = 0;

    // Pointers for iterating through arrays a and b

    int i = 0, j = 0;

    // Merge arrays a and b while maintaining sorted order

    while (i < n1 && j < n2) {

        if (a[i] < b[j]) {

            if (cnt == ind1) ind1Ele = a[i]; // Update ind1Ele if it's at ind1

            if (cnt == ind2) ind2Ele = a[i]; // Update ind2Ele if it's at ind2

            cnt++;

            i++;

        } else {

            if (cnt == ind1) ind1Ele = b[j]; // Update ind1Ele if it's at ind1

            if (cnt == ind2) ind2Ele = b[j]; // Update ind2Ele if it's at ind2

            cnt++;

            j++;

        }

    }

    // Handle remaining elements in array a

    while (i < n1) {

        if (cnt == ind1) ind1Ele = a[i]; // Update ind1Ele if it's at ind1

        if (cnt == ind2) ind2Ele = a[i]; // Update ind2Ele if it's at ind2

        cnt++;

        i++;

    }

    // Handle remaining elements in array b

    while (j < n2) {

        if (cnt == ind1) ind1Ele = b[j]; // Update ind1Ele if it's at ind1

        if (cnt == ind2) ind2Ele = b[j]; // Update ind2Ele if it's at ind2

        cnt++;

        j++;

    }

    // If the total number of elements is odd, return the second median element

    if (n % 2 == 1) {

        return (double) ind2Ele;

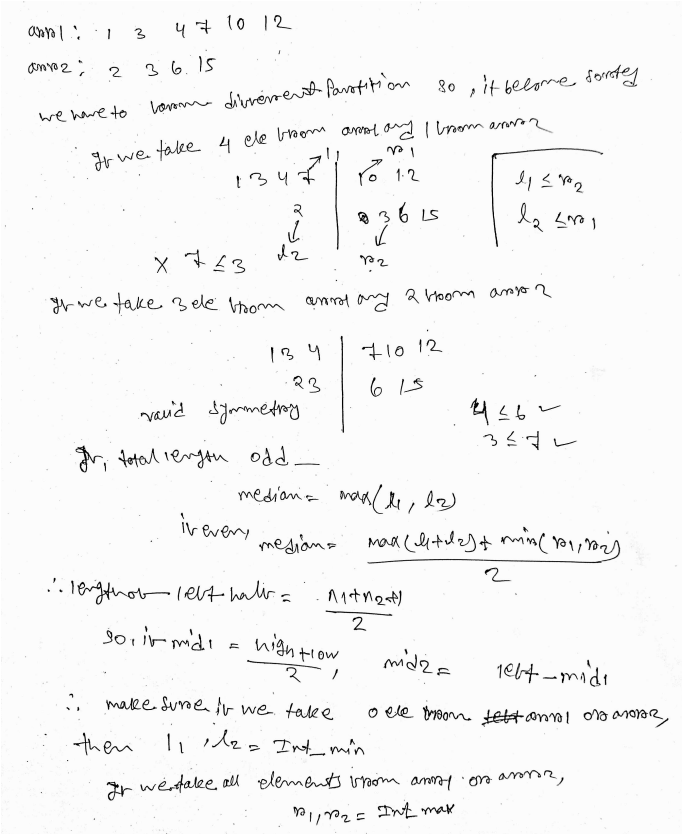
    }

    // If the total number of elements is even, return the average of the two median elements

    return (double) ((double) (ind1Ele + ind2Ele) / 2.0);

}

**Ap 2 : binary search**



public static double median(int[] a, int[] b) {

    // If array a is larger than array b, swap them and call the function again

    if (a.length > b.length) {

        return median(b, a);

    }

    // Lengths of arrays a and b

    int n1 = a.length;

    int n2 = b.length;

    // Total length of the merged array

    int n = n1 + n2;

    // Calculate the number of elements on the left side of the combined array

    int left = (n1 + n2 + 1) / 2;

    // Binary search variables

    int low = 0;

    int high = n1;

    // Perform binary search to find the correct partition point

    while (low <= high) {

        // Calculate midpoints for both arrays

        int mid1 = (low + high) / 2;

        int mid2 = left - mid1;

        // Find left and right elements for both arrays

        int l1 = mid1 > 0 ? a[mid1 - 1] : Integer.MIN\_VALUE;

        int l2 = mid2 > 0 ? b[mid2 - 1] : Integer.MAX\_VALUE;

        int r1 = mid1 < n1 ? a[mid1] : Integer.MAX\_VALUE;

        int r2 = mid2 < n2 ? b[mid2] : Integer.MAX\_VALUE;

        // Check if the partition is correct

        if (l1 <= r2 && l2 <= r1) {

            // If the total number of elements is odd, return the maximum of left elements

            if (n % 2 == 1) {

                return Math.max(l1, l2);

            } else { // If the total number of elements is even, return the average of max(left) and min(right)

                return (double) ((double) Math.max(l1, l2) + Math.min(r1, r2)) / 2.0;

            }

        } else if (l1 > r2) { // Move towards left in array a

            high = mid1 - 1;

        } else { // Move towards right in array a

            low = mid1 + 1;

        }

    }

    // If no median is found, return 0.0

    return 0.0;

}

**TC : O(min(len(arr1),len(arr2)) ; as we are always working with array with min length**

**SC : O(1)**