**############################################################**

**Remove duplicate elements from sorted Array**

Given a sorted array A of size N, delete all the duplicates elements from A.

**Example 1:**

**Input:**

N = 5

Array = {2, 2, 2, 2, 2}

**Output:** 2

**Explanation:** After removing all the duplicates

only one instance of 2 will remain.

**Example 2:**

**Input:**

N = 3

Array = {1, 2, 2}

**Output:** 1 2

**Your Task:**  
You dont need to read input or print anything. Complete the function **remove\_duplicate()** which takes the array A[] and its size N as input parameters and modifies it in place to delete all the duplicates. The function must return an integer X denoting the new modified size of the array.   
**Note:** The generated output will print all the elements of the modified array from index 0 to X-1.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= N <= 104  
1 <= A[i] <= 106

int remove\_duplicate(int a[],int n){

int j=1;

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

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

{ a[j]=a[i];

j++;

}

}

return j;

}

**############################################################**

**Reverse array in groups**

Given an array arr[] of positive integers of size N. Reverse every sub-array group of size K.

**Example 1:**

**Input:**

N = 5, K = 3

arr[] = {1,2,3,4,5}

**Output:** 3 2 1 5 4

**Explanation:** First group consists of elements

1, 2, 3. Second group consists of 4,5.

**Example 2:**

**Input:**

N = 4, K = 3

arr[] = {5,6,8,9}

**Output:** 8 6 5 9

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **reverseInGroups**() which takes the array, N and K as input parameters and modifies the array in-place.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ N, K ≤ 107  
1 ≤ A[i] ≤ 1018

void reverseInGroups(vector<long long>& arr, int n, int k){

int i,j;

for(int s=0;s<n;s+=k)

{ if(s+k<n)

i=s,j=s+k-1;

else

i=s,j=n-1;

while(i<j)

{ swap(arr[i],arr[j]);

i++;

j--;

}

}

}

**############################################################**

**Rotate Array**

Given an unsorted array **arr[]** of size **N**, rotate it by **D** elements in the counter-clockwise direction.

**Example 1:**

**Input:**

N = 5, D = 2

arr[] = {1,2,3,4,5}

**Output:** 3 4 5 1 2

**Explanation:** 1 2 3 4 5  when rotated

by 2 elements, it becomes 3 4 5 1 2.

**Example 2:**

**Input:**

N = 10, D = 3

arr[] = {2,4,6,8,10,12,14,16,18,20}

**Output:** 8 10 12 14 16 18 20 2 4 6

**Explanation:** 2 4 6 8 10 12 14 16 18 20

when rotated by 3 elements, it becomes

8 10 12 14 16 18 20 2 4 6.

**Your Task:**  
Complete the function **rotateArr**() which takes the array, D and N as input parameters and rotates the array by D elements. The array must be modified in-place without using extra space.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= N <= 107  
1 <= D <= N  
0 <= arr[i] <= 105

void rotate(int arr[], int l, int r)

{

while(l<r)

{ swap(arr[l],arr[r]);

l++;

r--;

}

}

//Function to rotate an array by d elements in counter-clockwise direction.

void rotateArr(int arr[], int d, int n){

// code here

rotate(arr,0,d-1);

rotate(arr,d,n-1);

rotate(arr,0,n-1);

}

**############################################################**

**Who has the majority?**

Given an array arr[] of size N and two elements x and y, use counter variables to find which element appears most in the array, x or y. If both elements have the same frequency, then return the smaller element.  
**Note:**  We need to return the element, not its count.

**Example 1:**

**Input:**

N = 11

arr[] = {1,1,2,2,3,3,4,4,4,4,5}

x = 4, y = 5

**Output:** 4

**Explanation:**

frequency of 4 is 4.

frequency of 5 is 1.

**Example 2:**

**Input:**

N = 8

arr[] = {1,2,3,4,5,6,7,8}

x = 1, y = 7

**Output:** 1

**Explanation:**

frequency of 1 is 1.

frequency of 7 is 1.

Since 1 < 7, return 1.

**Your Task:**  
You don't need to read input or print anything. Complete the function **majorityWins()** that takes**array, n, x, y**as input parametersand return the element with higher frequency.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= n <= 103  
0 <= arri , x , y <= 108

int majorityWins(int arr[], int n, int x, int y) {

int a=0,b=0,i=0;

while(i<n)

{ if(arr[i]==x)

a++;

if(arr[i]==y)

b++;

i++;

}

if(a>b) return x;

else if(b>a) return y;

else

{ if(x<y) return x;

else return y;

}

}

**############################################################**

**Find Transition Point**

Given a sorted array containing only 0s and 1s, find the transition point.

**Example 1:**

**Input:**

N = 5

arr[] = {0,0,0,1,1}

**Output:** 3

**Explanation:** index 3 is the transition

point where 1 begins.

**Example 2:**

**Input:**

N = 4

arr[] = {0,0,0,0}

**Output:** -1

**Explanation:** Since, there is no "1",

the answer is -1.

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **transitionPoint()** that takes array and N as input parameters and returns the 0 based index of the position where "0" ends and "1" begins. If array does not have any 1s, return -1. If array does not have any 0s, return 0.

**Expected Time Complexity:** O(LogN)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ N ≤ 500000  
0 ≤ arr[i] ≤ 1

int transitionPoint(int arr[], int n) {

if(arr[0]==1)

return 0;

if(n==1 && arr[0]==0)

return -1;

int l=0,r=n-1,m;

while(l<=r)

{ m=(l+r)/2;

if(arr[m]==1)

if(arr[m-1]==0)

return m;

else

r=m-1;

else if(arr[m]==0)

if(arr[m+1]==1)

return m+1;

else

l=m+1;

}

if(l>r)

return -1;

else

return m;

}

**############################################################**

**Equilibrium Point**

Given an array A of n positive numbers. The task is to find the first Equilibium Point in the array.   
Equilibrium Point in an array is a position such that the sum of elements before it is equal to the sum of elements after it.

**Example 1:**

**Input:**

n = 1

A[] = {1}

**Output:** 1

**Explanation:** Since its the only

element hence its the only equilibrium

point.

**Example 2:**

**Input:**

n = 5

A[] = {1,3,5,2,2}

**Output:** 3

**Explanation:** For second test case

equilibrium point is at position 3

as elements before it (1+3) =

elements after it (2+2).

**Your Task:**  
The task is to complete the function **equilibriumPoint()** which takes the array and n as input parameters and returns the point of equilibrium. Return -1 if no such point exists.

**Expected Time Complexity:**O(n)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= n <= 106  
1 <= A[i] <= 108

int equilibriumPoint(long long a[], int n) {

// Your code here

int sum=0;

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

sum+=a[i];

int left=0;

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

{ sum=sum-a[i];

if(left==sum)

return i+1;

else if(left>sum)

return -1;

else

left+=a[i];

}

}

**############################################################**

**Leaders in an array**

Given an array A of positive integers. Your task is to find the leaders in the array. An element of array is leader if it is greater than or equal to all the elements to its right side. The rightmost element is always a leader.

**Example 1:**

**Input:**

n = 6

A[] = {16,17,4,3,5,2}

**Output:** 17 5 2

**Explanation:** The first leader is 17

as it is greater than all the elements

to its right.  Similarly, the next

leader is 5. The right most element

is always a leader so it is also

included.

**Example 2:**

**Input:**

n = 5

A[] = {1,2,3,4,0}

**Output:** 4 0

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **leader**() which takes array A and n as input parameters and returns an array of leaders in order of their appearance.

**Expected Time Complexity:** O(n)  
**Expected Auxiliary Space:** O(n)

**Constraints:**  
1 <= n <= 107  
0 <= Ai <= 107

vector<int> leaders(int a[], int n){

// Code here

vector<int> ans;

ans.push\_back(a[n-1]);

int g=a[n-1];

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

{ if(a[i]>=g)

{ ans.push\_back(a[i]);

g=a[i];

}

}

reverse(ans.begin(),ans.end());

return ans;

}

**############################################################**

**Wave Array**

Given a sorted array **arr[]** of distinct integers. Sort the array into a wave-like array and return it. In other words, arrange the elements into a sequence such that a1 >= a2 <= a3 >= a4 <= a5..... (considering the increasing lexicographical order).

**Example 1:**

**Input:**

n = 5

arr[] = {1,2,3,4,5}

**Output:** 2 1 4 3 5

**Explanation:** Array elements after

sorting it in wave form are

2 1 4 3 5.

**Example 2:**

**Input:**

n = 6

arr[] = {2,4,7,8,9,10}

**Output:** 4 2 8 7 10 9

**Explanation:** Array elements after

sorting it in wave form are

4 2 8 7 10 9.

**Your Task:**  
The task is to complete the function **convertToWave**() which converts the given array to wave array.

**Expected Time Complexity:**O(n).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 ≤ n ≤ 106  
0 ≤ A[i] ≤107

void convertToWave(int \*arr, int n){

for(int i=0;i<n-1;i+=2)

swap(arr[i],arr[i+1]);

}

**############################################################**

**Subarray with given sum**

Given an unsorted array **A**of size **N** that contains only non-negative integers, find a continuous sub-array which adds to a given number **S**.

**Example 1:**

**Input:**

N = 5, S = 12

A[] = {1,2,3,7,5}

**Output:** 2 4

**Explanation:** The sum of elements

from 2nd position to 4th position

is 12.

**Example 2:**

**Input:**

N = 10, S = 15

A[] = {1,2,3,4,5,6,7,8,9,10}

**Output:** 1 5

**Explanation:** The sum of elements

from 1st position to 5th position

is 15.

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **subarraySum**() which takes arr, N and S as input parameters and returns a list containing the starting and ending positions of the first such occurring subarray from the left where sum equals to S. The two indexes in the list should be according to 1-based indexing. If no such subarray is found, return an array consisting only one element that is -1.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= N <= 105  
1 <= Ai <= 1010

vector<int> subarraySum(int arr[], int n, int s)

{ int l=0,r=-1,i=0,sum=0;

vector<int> ans;

while(i<n)

{ sum+=arr[i];

r++;

while(sum>s)

{ sum-=arr[l];

l++;

}

if(sum==s)

{ ans.push\_back(l+1);

ans.push\_back(r+1);

break;

}

i++;

}

if(i==n)

return {-1};

return ans;

}

**############################################################**

**First Repeating Element**

Given an array arr[] of size n, find the first repeating element. The element should occurs more than once and the index of its first occurrence should be the smallest.

**Example 1:**

**Input:**

n = 7

arr[] = {1, 5, 3, 4, 3, 5, 6}

**Output:** 2

**Explanation:**

5 is appearing twice and

its first appearence is at index 2

which is less than 3 whose first

occuring index is 3.

**Example 2:**

**Input:**

n = 4

arr[] = {1, 2, 3, 4}

**Output:** -1

**Explanation:**

All elements appear only once so

answer is -1.

**Your Task:**  
You don't need to read input or print anything. Complete the function **firstRepeated()** which takes arr and n as input parameters and return the position of the first repeating element. If there is no such element, return -1.  
The position you return should be according to 1-based indexing.

**Expected Time Complexity:** O(nlogn)  
**Expected Auxilliary Space:** O(n)

**Constraints:**  
1 <= n <= 106

int firstRepeated(int arr[], int n) {

//code here

unordered\_map<int,int> mp;

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

mp[arr[i]]++;

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

{ int k=arr[i];

auto temp=mp.find(k);

if(temp->second > 1)

return i+1;

}

return -1;

}

**############################################################**

**Find Immediate Smaller Than X**

Given an array arr[] of size N containing positive integers and an integer X, find the element in the array which is smaller than X and closest to it.

**Example 1:**

**Input:**

N = 5

arr[] = {4 67 13 12 15}

X = 16

**Output:** 15

**Explanation:** For a given value 16, there

are four values which are smaller than

it. But 15 is the number which is smaller

and closest to it with minimum difference

of 1.

**Example 2:**

**Input:**

N = 5

arr[] = {1 2 3 4 5}

X = 1

**Output:** -1

**Explanation:** No value is smaller than 1.

**Your Task:**  
You don't need to read input or print anything. You need to complete the given function **immediateSmaller()**which takes arr, N and X as input parameters and returns the closest element that is smaller than X. If no such element exists, return -1.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= **N** <= 103  
1 <= **arr[i], X** <= 104

int immediateSmaller(int a[], int n, int x)

{

// your code here

int min=INT\_MAX,f=-1;

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

{ if(a[i]<x && (x-a[i])<min)

{ min=x-a[i];

f=a[i];

}

}

return f;

}

**############################################################**

**Kadane's Algorithm**

Given an array **arr**of **N** integers. Find the contiguous sub-array with maximum sum.

**Example 1:**

**Input:**

N = 5

arr[] = {1,2,3,-2,5}

**Output:**

9

**Explanation:**

Max subarray sum is 9

of elements (1, 2, 3, -2, 5) which

is a contiguous subarray.

**Example 2:**

**Input:**

N = 4

arr[] = {-1,-2,-3,-4}

**Output:**

-1

**Explanation:**

Max subarray sum is -1

of element (-1)

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **maxSubarraySum**() which takes arr and N as input parameters and returns the sum of subarray with maximum sum.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ N ≤ 106  
-107 ≤ A[i] ≤ 107

int maxSubarraySum(int arr[], int n){

int t=0,g=INT\_MIN;

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

{ t+=arr[i];

if(t<arr[i])

t=arr[i];

if(g<t)

g=t;

}

return g;

}

**############################################################**

**Smallest Positive missing number**

Given an array **arr**[] of size **N**, find the smallest positive number missing from the array.

**Example 1:**

**Input:**

N = 5

arr[] = {1,2,3,4,5}

**Output:** 6

**Explanation:** Smallest positive missing

number is 6.

**Example 2:**

**Input:**

N = 5

arr[] = {0,-10,1,3,-20}

**Output:** 2

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **findMissing**() which takes arr and N as input parameters and returns the smallest positive missing number.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= N <= 106  
-106 <= arr[i] <= 106

int split(int arr[], int n)

{ int temp,j=0;

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

{ if(arr[i]<=0)

{ temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

j++;

}

}

return j;

}

int findMissing(int a[], int n) {

int p=split(a,n);

int arr[n-p];

int j=0;

for(int i=p;i<n;i++)

{ arr[j]=a[i];

j++;

}

for(int i=0;i<j;i++)

if(abs(arr[i])-1<j && arr[abs(arr[i])-1]>0)

arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];

int i;

for(i=0;i<j;i++)

if(arr[i]>0)

return i+1;

return j+1;

}

**############################################################**

**Rearrange an array with O(1) extra space**

Given an array **arr[]** of size **N** where every element is in the range from **0 to n-1**. Rearrange the given array so that **arr[i]**becomes **arr[arr[i]]**.

**Example 1:**

**Input:**

N = 2

arr[] = {1,0}

**Output:** 0 1

**Explanation:**

arr[arr[0]] = arr[1] = 0.

arr[arr[1]] = arr[0] = 1.

**Example 2:**

**Input:**

N = 5

arr[] = {4,0,2,1,3}

**Output:** 3 4 2 0 1

**Explanation:**

arr[arr[0]] = arr[4] = 3.

arr[arr[1]] = arr[0] = 4.

and so on.

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **arrange**() which takes arr and N as input parameters and rearranges the elements in the array in-place.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= N <= 107  
0 <= Arr[i] < N

void arrange(long long arr[], int n) {

for(int i=0;i<n;i++) arr[i]=arr[i]+(arr[arr[i]]%n)\*n;

for(int i=0;i<n;i++) arr[i]=arr[i]/n;

}

**############################################################**

**Trapping Rain Water**

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

**Example 1:**

**Input:**

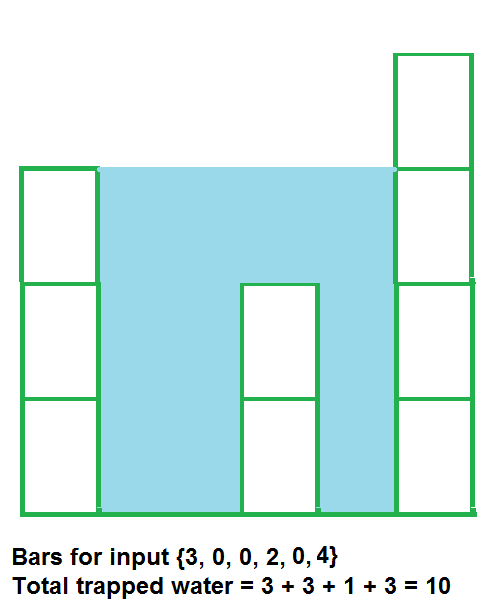
N = 6

arr[] = {3,0,0,2,0,4}

**Output:**

10

**Explanation:**



**Example 2:**

**Input:**

N = 4

arr[] = {7,4,0,9}

**Output:**

10

**Explanation:**

Water trapped by above

block of height 4 is 3 units and above

block of height 0 is 7 units. So, the

total unit of water trapped is 10 units.

**Example 3:**

**Input:**

N = 3

arr[] = {6,9,9}

**Output:**

0

**Explanation:**

No water will be trapped.

**Your Task:**  
You don'y need to read input or print anything. The task is to complete the function **trappingWater**() which takes arr and N as input parameters and returns the total amount of water that can be trapped.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(N)

**Constraints:**  
3 <= N <= 107  
0 <= Ai <= 108

int trappingWater(int arr[], int n){

int maxl=0,maxr=0,l[n],r[n];

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

{ l[i]=maxl;

if(arr[i]>maxl) maxl=arr[i];

}

for(int i=n-1;i>=0;i--)

{ r[i]=maxr;

if(arr[i]>maxr) maxr=arr[i];

}

int sum=0;

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

if(min(l[i],r[i])>arr[i])

sum+=min(l[i],r[i])-arr[i];

return sum;

}

**############################################################**

**Merge Without Extra Space**

Given two sorted arrays **arr1[]** and **arr2[]**ofsizes **n** and **m** in non-decreasing order. Merge them in sorted order without using any extra space. Modify arr1 so that it contains the first N elements and modify arr2 so that it contains the last M elements.

**Example 1:**

**Input**:

n = 4, arr1[] = [1 3 5 7]

m = 5, arr2[] = [0 2 6 8 9]

**Output**:

arr1[] = [0 1 2 3]

arr2[] = [5 6 7 8 9]

**Explanation**:

After merging the two

non-decreasing arrays, we get,

0 1 2 3 5 6 7 8 9.

**Example 2:**

**Input**:

n = 2, arr1[] = [10, 12]

m = 3, arr2[] = [5 18 20]

**Output**:

arr1[] = [5 10]

arr2[] = [12 18 20]

**Explanation**:

After merging two sorted arrays

we get 5 10 12 18 20.

**Your Task:**  
You don't need to read input or print anything. You only need to complete the function**merge()**that takes arr1, arr2, n and m as input parameters and modifies them in-place so that they look like the sorted merged array when concatenated.

**Expected Time Complexity:**  O((n+m) log(n+m))  
**Expected Auxilliary Space:** O(1)

**Constraints:**  
1 <= n, m <= 5\*104  
0 <= arr1i, arr2i <= 107

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

{ long long int temp,i=n-1,j=0;

while(i>=0 && j<m)

{ if(arr1[i]>arr2[j])

swap(arr1[i],arr2[j]); i--; j++;

}

sort(arr1,arr1+n);

sort(arr2,arr2+m);

}

**############################################################**