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

**1- Square root of a number**

Given an integer **x,** find the square root of x. If **x** is not a perfect square, then return floor(√x).

**Example 1:**

**Input:**

x = 5

**Output:** 2

**Explanation:** Since, 5 is not a perfect

square, floor of square\_root of 5 is 2.

**Example 2:**

**Input:**

x = 4

**Output:** 2

**Explanation:** Since, 4 is a perfect

square, so its square root is 2.

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **floorSqrt**() which takes x as the input parameter and return its square root.  
**Note:**Try Solving the question without using sqrt Function.

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

**Constraints:**  
1 ≤ x ≤ 107

long long int floorSqrt(long long int x)

{ if(x==0)

return 0;

long long int l=1,r=x,m,sol=0;

while(l<=r)

{ m=l+(r-l)/2;

if(m <= x/m)

{ sol=m;

l=m+1;

}

else

r=m-1;

}

return sol;

}

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

**2- Binary Search**

Given a sorted array of size N and an integer K, find the position at which K is present in the array using binary search.

**Example 1:**

**Input:**

N = 5

arr[] = {1 2 3 4 5}

K = 4

**Output:** 3

**Explanation:** 4 appears at index 3.

**Example 2:**

**Input:**

N = 5

arr[] = {11 22 33 44 55}

K = 445

**Output:** -1

**Explanation:** 445 is not present.

**Your Task:**  
You dont need to read input or print anything. Complete the function **binarysearch()** which takes arr[], N and K as input parameters and returns the index of K in the array. If K is not present in the array, return -1.

**Expected Time Complexity:** O(LogN)  
**Expected Auxiliary Space:** O(LogN) if solving recursively and O(1) otherwise.

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

int binarysearch(int arr[], int n, int k){

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

while(l<=r)

{ m=l+(r-l)/2;

if(arr[m]==k) return m;

else if(arr[m]<k) l=m+1;

else r=m-1;

}

return -1;

}

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

**3- Peak element**

A peak element in an array is the one that is not smaller than its neighbours.  
Given an array **arr[]** of size **N**, find the index of any one of its peak elements.  
**Note:**The generated output will always be 1 if the index that you return is correct. Otherwise output will be 0.

**Example 1:**

**Input:**

N = 3

arr[] = {1,2,3}

**Output:** 2

**Explanation:** index 2 is 3.

It is the peak element as it is

greater than its neighbour 2.

**Example 2:**

**Input:**

N = 2

arr[] = {3,4}

**Output:** 1

**Explanation:** 4 (at index 1) is the

peak element as it is greater than

its only neighbour element 3.

**Your Task:**  
You don't have to read input or print anything. Complete the function **peakElement**() which takes the array arr[] and its size N as input parameters and return the index of any one of its peak elements.

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

**Constraints:**  
1 ≤ N ≤ 105  
1 ≤ A[] ≤ 106

int peakElement(int arr[], int n)

{ if(arr[0]>arr[1])

return 0;

if(arr[n-1]>arr[n-2])

return n-1;

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

while(l<=r)

{ m=l+(r-l)/2;

if(arr[m]>arr[m-1] && arr[m]>arr[m+1])

return m;

else if(arr[m]<arr[m+1])

l=m+1;

else

r=m-1;

}

}

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

**4- Search an element in sorted and rotated array**

Given a sorted and rotated array **A**of N distinct elements which is rotated at some point, and given an element **K**. The task is to find the index of the given element K in the array A.

**Example 1:**

**Input:**

N = 9

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

K = 10

**Output:** 5

**Explanation:** 10 is found at index 5.

**Example 1:**

**Input:**

N = 3

A[] = {3,1,2}

K = 1

**Output:** 1

**User Task:**  
Complete **Search()**function and return the index of the element K if found in the array. If the element is not present, then return -1.

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

**Constraints:**  
1 ≤ N ≤ 107  
0 ≤ Ai ≤ 108  
1 ≤ K ≤ 108

int Search(vector<int> arr, int k) {

int n=arr.size();

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

while(l<=r)

{ m=l+(r-l)/2;

if(arr[m]==k)

return m;

else if(arr[m]>=arr[l])

{ if(arr[m]>=k && arr[l]<=k)

r=m-1;

else

l=m+1;

}

else

{ if(arr[m]<=k && arr[r]>=k)

l=m+1;

else

r=m-1;

}

}

return -1;

}

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

**5- Merge Sort**

Given an array arr[], its starting position l and its ending position r. Sort the array using merge sort algorithm.  
**Example 1:**

**Input:**

N = 5

arr[] = {4 1 3 9 7}

**Output:**

1 3 4 7 9

**Example 2:**

**Input:**

N = 10

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

**Output:**

1 2 3 4 5 6 7 8 9 10

**Your Task:**  
You don't need to take the input or print anything. Your task is to complete the function **merge()** which takes arr[], l, m, r as its input parameters and modifies arr[] in-place such that it is sorted from position l to position r, and function **mergeSort()** which uses merge() to sort the array in ascending order using merge sort algorithm.  
  
**Expected Time Complexity:** O(nlogn)

**Expected Auxiliary Space:** O(n)

**Constraints:**  
1 <= N <= 105  
1 <= arr[i] <= 103

void merge(int arr[], int l, int m, int r)

{ int x=m-l+1;

int y=r-m;

int a[x],b[y];

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

a[i]=arr[l+i];

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

b[i]=arr[m+1+i];

int i=0,j=0,k=l;

while(i<x && j<y)

{ if(a[i]<b[j])

{ arr[k]=a[i];

i++;

k++;

}

else

{ arr[k]=b[j];

j++;

k++;

}

}

while(i<x)

{ arr[k]=a[i];

i++;

k++;

}

while(j<y)

{ arr[k]=b[j];

j++;

k++;

}

}

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

{ if(l<r)

{ int m=l+(r-l)/2;

mergeSort(arr,l,m);

mergeSort(arr,m+1,r);

merge(arr,l,m,r);

}

}

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

**6- Quick Sort**

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.  
Given an array arr[], its starting position low and its ending position high.

Implement the partition() and quickSort() functions to sort the array.

**Example 1:**

**Input:**

N = 5

arr[] = { 4, 1, 3, 9, 7}

**Output:**

1 3 4 7 9

**Example 2:**

**Input:**

N = 9

arr[] = { 2, 1, 6, 10, 4, 1, 3, 9, 7}

**Output:**

1 1 2 3 4 6 7 9 10

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the functions **partition()**and **quickSort()**which takes the array arr[], low and high as input parameters and partitions the array. Consider the last element as the pivot such that all the elements less than(or equal to) the pivot lie before it and the elements greater than it lie after the pivot.

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

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

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

{ if(l<r)

{ int p=partition(arr,l,r);

quickSort(arr,l,p-1);

quickSort(arr,p+1,r);

}

}

int partition (int arr[], int l, int r)

{ int j=l-1;

int temp=arr[r],t;

for(int i=l;i<r;i++)

{ if(arr[i]<=temp && i!=j)

{ j++;

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

}

}

j++;

swap(arr[r],arr[j]);

return j;

}

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

**7- Union of Two Sorted Arrays**

Union of two arrays can be defined as the common and distinct elements in the two arrays.  
Given two sorted arrays of size **n** and **m** respectively, find their union.

**Example 1:**

**Input**:

n = 5, arr1[] = {1, 2, 3, 4, 5}

m = 3, arr2 [] = {1, 2, 3}

**Output**: 1 2 3 4 5

**Explanation**: Distinct elements including

both the arrays are: 1 2 3 4 5.

**Example 2:**

**Input**:

n = 5, arr1[] = {2, 2, 3, 4, 5}

m = 5, arr2[] = {1, 1, 2, 3, 4}

**Output**: 1 2 3 4 5

**Explanation**: Distinct elements including

both the arrays are: 1 2 3 4 5.

**Example 3:**

**Input**:

n = 5, arr1[] = {1, 1, 1, 1, 1}

m = 5, arr2[] = {2, 2, 2, 2, 2}

**Output**: 1 2

**Explanation**: Distinct elements including

both the arrays are: 1 2.

**Your Task:**  
You do not need to read input or print anything. Complete the **function findUnion()**that takes two arrays **arr1[]**, **arr2[],** and their size **n and m**as input parameters and returns a list containing the **union of the two arrays**. 

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

**Constraints:**  
1 <= n, m <= 105  
1 <= arr[i], brr[i] <= 106

vector<int> findUnion(int arr1[], int arr2[], int n, int m)

{ vector<int> v;

int i=0,j=0,k=0,t=-1;

while(i<n && j<m)

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

{ if(arr1[i]==t)

i++;

else

{ v.push\_back(arr1[i]);

t=arr1[i];

i++;

k++;

}

}

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

{ if(arr2[j]==t)

j++;

else

{ v.push\_back(arr2[j]);

t=arr2[j];

j++;

k++;

}

}

else

{ if(arr1[i]==t)

{ i++;

j++;

}

else

{ v.push\_back(arr1[i]);

t=arr1[i];

i++;

j++;

k++;

}

}

}

while(i<n)

{ if(arr1[i]==t)

i++;

else

{ v.push\_back(arr1[i]);

t=arr1[i];

i++;

k++;

}

}

while(j<m)

{ if(arr2[j]==t)

j++;

else

{ v.push\_back(arr2[j]);

t=arr2[j];

j++;

k++;

}

}

return v;

}

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

**8- Intersection of two sorted arrays**

The intersection of two arrays contains the elements common to both the arrays. The intersection should not count duplicate elements.  
Given two sorted arrays **arr1**[] and **arr**2[] of sizes **N** and **M** respectively. Find their **intersection**

**Example 1:**

**Input**:

N = 4, arr1[] = {1, 2, 3, 4}

M = 5, arr2 [] = {2, 4, 6, 7, 8}

**Output**: 2 4

**Explanation**: 2 and 4 are only common

elements in both the arrays.

**Example 2:**

**Input**:

N = 5, arr1[] = {1, 2, 2, 3, 4}

M = 6, arr2[] = {2, 2, 4, 6, 7, 8}

**Output**: 2 4

**Explanation**: 2 and 4 are the only

common elements.

**Example 3:**

**Input**:

N = 2, arr1[] = {1, 2}

M = 2, arr2[] = {3, 4}

**Output**: -1

**Explanation**: No common elements.

**Your Task:**  
You do not need to read input or print anything. Complete the **function printIntersection()**that takes arr1,arr2,  N and M as input parameters and return a list of integers containing the intersection of two arrays. If the intersection is empty then then list should contain -1.  
  
  
**Expected Time Complexity:** O(N + M).  
**Expected Auxiliary Space:** O(N + M).

**Constraints:**  
1 <= N, M <= 105  
1 <= arr[i], brr[i] <= 106

vector<int> printIntersection(int arr1[], int arr2[], int n, int m)

{ vector<int> v;

int i=0,j=0,k=0,t=-1;

while(i<n && j<m)

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

{ if(arr1[i]==t)

i++;

else

{ i++;

k++;

}

}

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

{ if(arr2[j]==t)

j++;

else

{ j++;

k++;

}

}

else

{ if(arr1[i]==t)

{ i++;

j++;

}

else

{ v.push\_back(arr1[i]);

t=arr1[i];

i++;

j++;

k++;

}

}

}

return v;

}

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

**9- Minimum Swaps to Sort**

Given an array of n distinct elements. Find the minimum number of swaps required to sort the array in strictly increasing order.

**Example 1:**

**Input:**

nums = {2, 8, 5, 4}

**Output:**

1

**Explaination:**

swap 8 with 4.

**Example 2:**

**Input:**

nums = {10, 19, 6, 3, 5}

**Output:**

2

**Explaination:**

swap 10 with 3 and swap 19 with 5.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **minSwaps()**which takes the **nums**as input parameter and returns an integer denoting the minimum number of swaps required to sort the array. If the array is already sorted, return 0.

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

**Constraints:**  
1 ≤ n ≤ 105  
1 ≤ numsi ≤ 106

void swap(vector<int> &nums, int i, int j)

{ int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

int minSwaps(vector<int>&nums)

{ int n=nums.size();

pair<int, int> index[n];

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

{ index[i].first = nums[i];

index[i].second = i;

}

sort(index,index+n);

vector<bool> b(n,false);

int ans=0;

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

{ if(b[i] || index[i].second==i)

continue;

int c=0;

int j=i;

while(!b[j])

{ b[j]=1;

j=index[j].second;

c++;

}

if(c>0)

ans+=(c-1);

}

return ans;

}

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

**10- Minimum Platforms**

Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.  
Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms,

**Example 1:**

**Input**: n = 6

arr[] = {0900, 0940, 0950, 1100, 1500, 1800}

dep[] = {0910, 1200, 1120, 1130, 1900, 2000}

**Output**: 3

**Explanation**:

Minimum 3 platforms are required to

safely arrive and depart all trains.

**Example 2:**

**Input**: n = 3

arr[] = {0900, 1100, 1235}

dep[] = {1000, 1200, 1240}

**Output**: 1

**Explanation**: Only 1 platform is required to

safely manage the arrival and departure

of all trains.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findPlatform()** which takes the array arr[] (denoting the arrival times), array dep[] (denoting the departure times) and the size of the array as inputs and returns the minimum number of platforms required at the railway station such that no train waits.

**Note:** Time intervals are in the 24-hour format(**HHMM) ,** where the first two characters represent hour (between 00 to 23 ) and the last two characters represent minutes (between 00 to 59).

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

**Constraints:**  
1 <= n <= 50000  
0000 <= A[i] < D[i] <= 2359

int findPlatform(int arr[], int dep[], int n)

{ sort(arr,arr+n);

sort(dep,dep+n);

int maxp=1;

int needp=1;

int i=1,j=0;

while(i<n)

{ if(needp<maxp)

needp=maxp;

if(arr[i]>dep[j])

{ maxp--;

j++;

}

if(arr[i]<=dep[j])

{ maxp++;

i++;

}

}

return needp;

}

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

**11- Median of Two sorted arrays**

Given two sorted arrays of sizes **N** and **M** respectively. The task is to find the median of the two arrays when they get merged.

**Example 1:**

**Input:**

N = 5, M = 6

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

brr[] = {3,4,5,6,7,8}

**Output:** 4

**Explanation:** After merging two arrays,

elements will be as 1 2 3 3 4 4 5 5 6 7 8

So, median is 4.

**Example 2:**

**Input:**

N = 2, M = 3

arr[] = {1,2}

brr[] = {2,3,4}

**Output:** 2

**Explanation:** After merging two arrays,

elements will be as 1 2 2 3 4. So,

median is 2.

**Your Task:**  
You do not need to read input or print anything. Complete **findMedian()**function which takes the two arrays and n and m as input parameters and returns their median. If there are total even elements, return floor of average of middle two elements.

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

**Constraints:**  
1 <= N, M <= 106  
1 <= arr[i], brr[i] <= 107

int findMedian(int arr[], int n, int brr[], int m)

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

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

{ if(arr[i]>brr[j])

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

i--;

j++;

}

sort(arr,arr+n);

sort(brr,brr+m);

int mid=(m+n)/2;

if((m+n)%2==0)

{ if(mid<n)

return (double)(arr[mid-1]+arr[mid])/2;

if(mid==n)

return (double)(arr[mid-1]+brr[0])/2;

if(mid>n)

return (double)(brr[mid-1-n]+brr[mid-n])/2;

}

else

{ if(mid<n)

return (double)(arr[mid]);

if(mid>=n)

return (double)(brr[mid-n]);

}

}

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

**12- Sort an array of 0s, 1s and 2s**

Given an array of size N containing only 0s, 1s, and 2s; sort the array in ascending order.

**Example 1:**

**Input:**

N = 5

arr[]= {0 2 1 2 0}

**Output:**

0 0 1 2 2

**Explanation:**

0s 1s and 2s are segregated

into ascending order.

**Example 2:**

**Input:**

N = 3

arr[] = {0 1 0}

**Output:**

0 0 1

**Explanation:**

0s 1s and 2s are segregated

into ascending order.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **sort012()**that takes an array arr and N as input parameters and **sorts the array in-place.**

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

**Constraints:**  
1 <= N <= 10^6  
0 <= A[i] <= 2

void sort012(int a[], int n)

{ int i=0,j=n-1,k=0;

while(k<=j)

{ if(a[k]==0) swap(a[i++],a[k++]);

else if(a[k]==2) swap(a[k],a[j--]);

else k++;

}

}

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