⛵

# Binary Search Notes

## 1. Binary Search

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with `O(log n)` runtime complexity.

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int l=0;
        int h=nums.size()-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(nums[mid]==target)return mid;
            else if(nums[mid]>target)h=mid-1;
            else l=mid+1;
        }
        return -1;
    }
};
```

## 2.Implement Lower Bound and Upper Bound

Given an unsorted array **Arr[]** of **N** integers and an integer **X**, find floor and ceiling of **X** in **Arr[0..N-1]**.

**Floor** of **X** is the largest element which is smaller than or equal to **X**. Floor of **X** doesn’t exist if **X** is smaller than smallest element of **Arr[]**.

**Ceil** of **X** is the smallest element which is greater than or equal to **X**. Ceil of **X** doesn’t exist if **X** is greater than greatest element of **Arr[]**.

Example—

```
Input:
N = 8, X = 7
Arr[] = {5, 6, 8, 9, 6, 5, 5, 6}
Output: 6 8
Explanation:
Floor of 7 is 6 and ceil of 7
is 8.
```

```
pair<int, int> getFloorAndCeil(int arr[], int n, int x) {
    // code here
    sort(arr,arr+n);

    int lb=-1;//upper bound
    int ub=-1;//upper bound

    int l=0;
    int h=n-1;

    while(l<=h){
        int mid=l+(h-l)/2;
        if(arr[mid]==x){
            return {arr[mid],arr[mid]};
        }else if(arr[mid]>x){
            ub=arr[mid];
            h=mid-1;
        }else{
            lb=arr[mid];
            l=mid+1;
        }
    }

    return {lb,ub};
}
```

## 3.Search Insert Position

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with `O(log n)` runtime complexity.

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int l=0;
        int h=nums.size()-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(nums[mid]==target)return mid;
            else if(nums[mid]<target)l=mid+1;
            else h=mid-1;
        }
        return l;
    }
};
```

## 4.Find First and Last Position of Element in Sorted Array

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given `target` value.

If `target` is not found in the array, return `[-1, -1]`.

You must write an algorithm with `O(log n)` runtime complexity.

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int fi,li;
        fi=li=-1;

        int l=0;
        int h=nums.size()-1;

        //For First Position of Element in Sorted Array
        while(l<=h){
            int mid=l+(h-l)/2;
            if(nums[mid]==target){
                fi=mid;
                h=mid-1;
            }else if(nums[mid]<target){
                l=mid+1;
            }else h=mid-1;
        }

        //For Last Position of Element in Sorted Array
        l=0;
        h=nums.size()-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(nums[mid]==target){
                li=mid;
                l=mid+1;
            }else if(nums[mid]<target){
                l=mid+1;
            }else h=mid-1;
        }

        return {fi,li};
    }
};
```

## 5.Find Peak Element

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that `nums[-1] = nums[n] = -∞`. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

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

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l=0;
        int h=nums.size()-1;

        while(l<=h){
            int mid1=l+(h-l)/2;
            int mid2=mid1+1;
            if(mid2<nums.size()&&nums[mid1]<nums[mid2]){
                //mid2 could possibly be the answer
                l=mid2;
            }else h=mid1-1;
        }
        return l;
    }
};
```

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int n=nums.size();
        int l=0;
        int h=n-1;

        while(l<=h){
            int mid=l+(h-l)/2;
            if((mid-1<0||nums[mid-1]<nums[mid])&&(mid+1>=n||nums[mid]>nums[mid+1])){
                return mid;
            }else if(mid+1<n&&nums[mid+1]>nums[mid]){
                l=mid+1;
            }else h=mid-1;
        }

        return -1;
    }
};
```

## 6.Search in Rotated Sorted Array

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly rotated** at an unknown pivot index `k` (`1 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]`.

Given the array `nums` **after** the possible rotation and an integer `target`, return *the index of* `target` *if it is in* `nums`*, or* `-1` *if it is not in* `nums`.

You must write an algorithm with `O(log n)` runtime complexity.

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n=nums.size();
        int l=0;
        int h=n-1;

        //Find Pivot at which sorted array id rotated
        while(l<h){
           int mid=l+(h-l)/2;
           if(nums[mid]<=nums[h]){
            h=mid;
           }else l=mid+1;
        }

        int pivot=l;


        //Apply BS on first half [0....(pivot-1)]
        l=0;
        h=pivot-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(nums[mid]==target)return mid;
            else if(nums[mid]>target)h=mid-1;
            else l=mid+1;
        }
        //Apply BS on second half [pivot.....(n-1)]
        l=pivot;
        h=n-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(nums[mid]==target)return mid;
            else if(nums[mid]>target)h=mid-1;
            else l=mid+1;
        }

        return -1;
    }
};
```

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n=nums.size();
        int l=0;
        int h=n-1;
        while(l<=h){
            int mid=l+(h-l)/2;

            if(nums[mid]==target)return mid;

            //If left half is sorted
            if(nums[l]<=nums[mid]){
                //If target is in left sorted half->Eliminate right search space
                if(nums[l]<=target&&target<=nums[mid]){
                    h=mid-1;
                }else{
                //If target is not in left sorted part ->Eliminate left space search
                    l=mid+1;
                }
            }
            //If right half is sorted
            else{
                //If target is in right sorted half->Eliminate left search space
                if(nums[mid]<=target&&target<=nums[h]){
                    l=mid+1;
                }else{
                 //If target is not in right sorted half->Eliminate right search space
                    h=mid-1;
                }
            }
        }
        return -1;
    }
};
```

## 7.Search in Rotated Sorted Array II

There is an integer array `nums` sorted in non-decreasing order (not necessarily with **distinct** values).

Before being passed to your function, `nums` is **rotated** at an unknown pivot index `k` (`0 <= k < nums.length`) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,4,4,5,6,6,7]` might be rotated at pivot index `5` and become `[4,5,6,6,7,0,1,2,4,4]`.

Given the array `nums` **after** the rotation and an integer `target`, return `true` *if* `target` *is in* `nums`*, or* `false` *if it is not in* `nums`*.*

You must decrease the overall operation steps as much as possible.

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int n=nums.size();
        int l=0;
        int h=n-1;
        while(l<=h){
            int mid=l+(h-l)/2;

            if(nums[mid]==target)return true;
            //Eliminating problem of duplicacy
            else if(nums[l]==nums[mid]&&nums[mid]==nums[h]){
                l++;
                h--;
            }
            //Left half is sorted
            else if(nums[l]<=nums[mid]){
                 //Target is in left half
                 if(nums[l]<=target&&target<=nums[mid]){
                     h=mid-1;
                 }else{
                     l=mid+1;
                 }
            }
            //Right half is sorted
            else{
                if(nums[mid]<=target&&target<=nums[h]){
                    l=mid+1;
                }else{
                    h=mid-1;
                }
            }
        }
        return false;
    }
};
```

## 8.Find Minimum in Rotated Sorted Array

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

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

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

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

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

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int n=nums.size();
        int l=0;
        int h=n-1;
        while(l<h){
            int mid=l+(h-l)/2;
            //right half is sorted
            if(nums[mid]<=nums[h]){
                h=mid;
            }else{
                l=mid+1;
            }
        }
        return nums[l];
    }
};
```

## 9.Single Element in a Sorted Array

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return *the single element that appears only once*.

Your solution must run in `O(log n)` time and `O(1)` space.

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        int n=nums.size();
        int l=0;
        int h=n-1;

        while(l<h){
            int mid=l+(h-l)/2;
            //left part
            if((mid%2==0&&mid+1&&nums[mid]==nums[mid+1])||(mid%2!=0&&mid-1>=0&&nums[mid-1]==nums[mid]))l=mid+1;

            //right part
            else h=mid;
        }

        return nums[l];
    }
};

//right half
// 1st instance->odd index
// 2nd instance->even index

//left half
// 1st instance->even index
// 2nd instance->odd index

//Trick
// 1.If we do mid^1 and if mid is a even number then we will get a number that is mid+1
// mid-> 100   (4)
    //  ^001   (1)
// ans-> 101   (5)

//2.If we do mid^1 and if mid is a odd number then we will get a number that is mid-1
//mid->  101   (5)
//      ^001   (1)
//ans->  100   (4)
```

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        int n=nums.size();
        int l=0;
        int h=n-2;

        while(l<=h){
            int mid=l+(h-l)/2;

            //right half
            if(nums[mid]==nums[mid^1]){
                l=mid+1;
            }
            //left half
            else{
                h=mid-1;
            }
        }

        return nums[l];
    }
};

//right half
// 1st instance->odd index
// 2nd instance->even index

//left half
// 1st instance->even index
// 2nd instance->odd index

//  0 1 2 3 4 5 6 7 8
// [1,1,2,3,3,4,4,8,8]

//Trick
// 1.If we do mid^1 and if mid is a even number then we will get a number that is mid+1
// mid-> 100   (4)
    //  ^001   (1)
// ans-> 101   (5)

//2.If we do mid^1 and if mid is a odd number then we will get a number that is mid-1
//mid->  101   (5)
//      ^001   (1)
//ans->  100   (4)
```

## 10.K-th element of two sorted Arrays

Given two sorted arrays **arr1** and **arr2** of size **N** and **M** respectively and an element **K** . The task is to find the element that would be at the kth position of the final sorted array.

TC→O(log(min(m,n)))

SC→O(1)

```
public:
    int kthElement(int arr1[], int arr2[], int size1, int size2, int k)
    {
        //Applying BS on smaller array
        if(size1>size2){
            return kthElement(arr2,arr1,size2,size1,k);
        }


        //Applying BS on arr1
        int l=max(0,k-size2);
        int h=min(k,size1);

        while(l<=h){
            int cut1=l+(h-l)/2;

            int cut2=k-cut1;

            int l1=cut1-1>=0?arr1[cut1-1]:INT_MIN;
            int l2=cut2-1>=0?arr2[cut2-1]:INT_MIN;
            int r1=cut1<size1?arr1[cut1]:INT_MAX;
            int r2=cut2<size2?arr2[cut2]:INT_MAX;

            if(l1<=r2&&l2<=r1){
                return max(l1,l2);
            }
            else if(l1>r2){
                h=cut1-1;
            }else l=cut1+1;
        }
        return -1;
    }
};
```

## 11.Find out how many times has an array been rotated

Given an ascending sorted rotated array **Arr** of distinct integers of size **N**. The array is right rotated **K**times. Find the value of **K**.

Example1:-

```
Input:
N = 5
Arr[] = {5, 1, 2, 3, 4}
Output: 1
Explanation: The given array is 5 1 2 3 4.
The original sorted array is 1 2 3 4 5.
We can see that the array was rotated
1 times to the right.
```

```
class Solution{
public:
  int findKRotation(int arr[], int n) {
      // code here
      int l=0;
      int h=n-1;
      while(l<h){
          int mid=l+(h-l)/2;
          if(arr[mid]<=arr[h]){
              h=mid;
          }else l=mid+1;
      }
      return l;
  }

};
```

## 12.Search a 2D matrix

Write an efficient algorithm that searches for a value `target` in an `m x n` integer matrix `matrix`. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m=matrix.size();
        int n=matrix[0].size();
        int i=0;
        int j=n-1;
        while(i<m&&j>=0){
            if(matrix[i][j]==target)return true;
            else if(matrix[i][j]>target)j--;
            else i++;
        }
        return false;
    }
};
```

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int ROW=matrix.size();
        int COL=matrix[0].size();
        int l=0;
        int h=ROW*COL-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(matrix[mid/COL][mid%COL]==target)return true;
            else if(matrix[mid/COL][mid%COL]>target)h=mid-1;
            else l=mid+1;
        }
        return false;
    }
};
```

## 13.Find a Peak Element II

A **peak** element in a 2D grid is an element that is **strictly greater** than all of its **adjacent** neighbors to the left, right, top, and bottom.

Given a **0-indexed** `m x n` matrix `mat` where **no two adjacent cells are equal**, find **any** peak element `mat[i][j]` and return *the length 2 array* `[i,j]`.

You may assume that the entire matrix is surrounded by an **outer perimeter** with the value `-1` in each cell.

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

Example1:-

```
Input: mat = [[1,4],[3,2]]
Output: [0,1]
Explanation: Both 3 and 4 are peak elements so [1,0] and [0,1] are both acceptable answers.
```

```
class Solution {
public:
    vector<int> findPeakGrid(vector<vector<int>>& mat) {
        int m=mat.size();
        int n=mat[0].size();

        //BS for column
        int l=0;
        int h=n-1;
        while(l<=h){
            int mid=l+(h-l)/2;

            int max_elem=INT_MIN;
            int max_elem_row;

            for(int i=0;i<m;i++){
                if(max_elem<mat[i][mid]){
                    max_elem=mat[i][mid];
                    max_elem_row=i;
                }
            }
            if((mid-1<0||mat[max_elem_row][mid-1]<mat[max_elem_row][mid])&&(mid+1>=n||mat[max_elem_row][mid]>mat[max_elem_row][mid+1])){
                return {max_elem_row,mid};
            }
            else if(mid+1<n&&mat[max_elem_row][mid]<mat[max_elem_row][mid+1]){
                l=mid+1;
            }else h=mid-1;
        }

        return {-1,-1};
    }
};
```

## 14.Median in a row-wise sorted Matrix

Given a row wise sorted matrix of size **R*C** where R and C are always **odd**
, find the median of the matrix.

Example1:-

```
Input:
R = 3, C = 3
M = [[1, 3, 5],
     [2, 6, 9],
     [3, 6, 9]]
Output: 5
Explanation: Sorting matrix elements gives
us {1,2,3,3,5,6,6,9,9}. Hence, 5 is median.
```

```
class Solution{
public:
    int f(vector<int>&row,int x){
        int cnt_small=0;
        int l=0;
        int h=row.size()-1;

        while(l<=h){
            int mid=l+(h-l)/2;
            if(row[mid]<=x){
                l=mid+1;
            }else h=mid-1;
        }
        return l;
    }

    int median(vector<vector<int>> &matrix, int R, int C){
        // code here
        int l=0;
        int h=1e9;

        while(l<=h){
            int mid=l+(h-l)/2;

            //counting element less than equal to mid;
            int cnt_small=0;
            for(int i=0;i<R;i++){
                cnt_small+=f(matrix[i],mid);
            }

            if(cnt_small<=((R*C)/2)){
                l=mid+1;
            }else h=mid-1;
        }

        return l;
    }
};
```

## 15.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.
```

```
class Solution{
  public:
    long long int floorSqrt(long long int x)
    {
        // Your code goes here
        long long int l=0;
        long long int h=x;
        long long ans=-1;
        while(l<=h){
            long long int mid=l+(h-l)/2;
            if(mid*mid==x){
                ans=mid;
                break;
            }else if(mid*mid<x){
                ans=mid;
                l=mid+1;
            }else h=mid-1;
        }
        return ans;
    }
};
```

## 16.Find Nth root of M

You are given 2 numbers **(n , m)**; the task is to find **n√m**(nth root of m).

```
class Solution{
  public:
  int NthRoot(int n, int m)
  {
      // Code here.
      int l=0;
      int h=m;

      while(l<=h){
          int mid=l+(h-l)/2;

          double mul=1;
          for(int i=1;i<=n;i++){
              mul=mul*mid;
          }

          if(mul==m)return mid;
          else if(mul<m)l=mid+1;
          else h=mid-1;
      }
      return -1;
  }
};
```

## 17.Koko Eating Bananas

Koko loves to eat bananas. There are `n` piles of bananas, the `i th` pile has `piles[i]` bananas. The guards have gone and will come back in `h` hours.

Koko can decide her bananas-per-hour eating speed of `k`. Each hour, she chooses some pile of bananas and eats `k` bananas from that pile. If the pile has less than `k` bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return *the minimum integer* `k` *such that she can eat all the bananas within* `h` *hours*.

Example▬

```
Input: piles = [3,6,7,11], h = 8
Output: 4
```

```
class Solution {
public:
    int minEatingSpeed(vector<int>& piles, int h) {
        sort(piles.begin(),piles.end());
        int low=1;
        int high=*max_element(piles.begin(),piles.end());
        while(low<high){
            int mid=low+(high-alow)/2;
            int time_taken=0;
            for(int i=0;i<piles.size();i++){
                time_taken+=ceil((double)piles[i]/(double)mid);
            }

            if(time_taken<=h){
                high=mid;
            }else{
                low=mid+1;
            }
        }

        return low;
    }
};
```

## 18.Capacity To Ship Packages Within D Days

A conveyor belt has packages that must be shipped from one port to another within `days` days.

The `i th` package on the conveyor belt has a weight of `weights[i]`. Each day, we load the ship with packages on the conveyor belt (in the order given by `weights`). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within `days` days.

Example

```
Input: weights = [1,2,3,4,5,6,7,8,9,10], days = 5
Output: 15
Explanation: A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:
1st day: 1, 2, 3, 4, 5
2nd day: 6, 7
3rd day: 8
4th day: 9
5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2,
3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.
```

```
class Solution {
public:
    int possible(vector<int>&weights,int cap,int days){
        int curr_cap=0;
        int curr_days=1;
        for(int i=0;i<weights.size();i++){
            if(curr_cap+weights[i]<=cap){
                curr_cap+=weights[i];
            }else{
                curr_days++;
                if(curr_days>days)return false;
                if(weights[i]>cap)return false;
                curr_cap=weights[i];
            }
        }

        return true;
    }

    int shipWithinDays(vector<int>& weights, int days) {
        int capacity=0;

        int wsum=0;

        for(auto& w:weights)wsum+=w;

        int l=1;
        int h=1e9;// or h=wsum

        while(l<=h){
            int mid=l+(h-l)/2;

            if(possible(weights,mid,days)){
                capacity=mid;
                h=mid-1;
            }else l=mid+1;
        }

        return capacity;
    }
};
```

## 19.Median of two sorted arrays

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

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

**Example 1:**

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

**Example 2:**

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

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int size1=nums1.size();
        int size2=nums2.size();

        // applying BS on smaller array so to get min TC
        if(size1>size2){
            return findMedianSortedArrays(nums2,nums1);
        }

        int l=0;
        int h=nums1.size();

        while(l<=h){
            int cut1=l+(h-l)/2;
            int cut2=(size1+size2+1)/2-cut1;

            int l1=cut1-1>=0?nums1[cut1-1]:INT_MIN;
            int l2=cut2-1>=0?nums2[cut2-1]:INT_MIN;
            int r1=cut1<size1?nums1[cut1]:INT_MAX;
            int r2=cut2<size2?nums2[cut2]:INT_MAX;

            if(l1<=r2&&l2<=r1){
                if((size1+size2)%2==0){
                    return (max(l1,l2)+min(r1,r2))/2.0;
                }else{
                    return max(l1,l2);
                }
            }else if(l1>r2){
                h=cut1-1;
            }else l=cut1+1;
        }

        return 0.0;
    }
};
```

## 20.Aggressive Cows

Given an array of length 'N', where each element denotes the position of a stall. Now you have 'N' stalls and an integer 'K which denotes the number of cows that are aggressive. To prevent the cows from hurting each other, you need to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. Return the largest minimum distance.

```
#include<bits/stdc++.h>

//TC-> O(stall.size()*log(max(stalls)-min(stalls)))

bool possible(vector<int>&stalls,int min_dist,int cows){
    int curr_cows=0;
    int last_placed_cow=-1;
    int n=stalls.size();
    for(int i=0;i<n;i++){
        if(last_placed_cow==-1||stalls[i]-last_placed_cow>=min_dist){
            curr_cows++;
            last_placed_cow=stalls[i];
        }

        if(curr_cows==cows)return true;
    }
    return false;
}

int aggressiveCows(vector<int> &stalls, int k)
{
    //    Write your code here.
    sort(stalls.begin(),stalls.end());
    int n=stalls.size();
    int l=1;
    int h=stalls[n-1]-stalls[0];
    int ans=-1;
    while(l<=h){
        int mid=l+(h-l)/2;
        if(possible(stalls,mid,k)){
            ans=mid;
            l=mid+1;
        }else h=mid-1;
    }
    return ans;
}
```

## 21.Book Allocation

You are given **N** number of books. Every **ith** book has **Ai** number of pages. You have to allocate contiguous books to **M** number of students. There can be many ways or permutations to do so. In each permutation, one of the M students will be allocated the maximum number of pages. Out of all these permutations, the task is to find that particular permutation in which the maximum number of pages allocated to a student is the minimum of those in all the other permutations and print this minimum value.

Each book will be allocated to exactly one student. Each student has to be allocated at least one book.

Note: Return **-1** if a valid assignment is not possible, and allotment should be in contiguous order (see the explanation for better understanding).

Example1:-

```
Input:
N = 4
A[] = {12,34,67,90}
M = 2
Output:113
Explanation:Allocation can be done in
following ways:{12} and {34, 67, 90}
Maximum Pages = 191{12, 34} and {67, 90}
Maximum Pages = 157{12, 34, 67} and {90}
Maximum Pages =113. Therefore, the minimum
of these cases is 113, which is selected
as the output.
```

Example2:-

```
Input:
N = 3
A[] = {15,17,20}
M = 2
Output:32
Explanation:Allocation is done as
{15,17} and {20}
```

```
class Solution
{
    public:

    int possible(int A[],int N,int students,int max_pages){
        int curr_students=1;
        int curr_pages=0;
        for(int i=0;i<N;i++){
            if(curr_pages+A[i]<=max_pages){
                curr_pages+=A[i];
            }else{
                curr_students++;
                if(curr_students>students)return false;

                if(A[i]>max_pages)return false;

                curr_pages=A[i];
            }
        }

        return true;
    }

    //Function to find minimum number of pages.
    int findPages(int A[], int N, int M)
    {
        //code here
        if(M>N)return -1; //Don't forget this edge case -> It's very important

        int l=1;
        int sum=0;
        for(int i=0;i<N;i++){
            sum+=A[i];
        }
        int h=sum;
        int ans=-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(possible(A,N,M,mid)){
                ans=mid;
                h=mid-1;
            }else l=mid+1;
        }
        return ans;
    }
};
```

## 21.Split Array Largest Sum

Given an integer array `nums` and an integer `k`, split `nums` into `k` non-empty subarrays such that the largest sum of any subarray is **minimized**.

Return *the minimized largest sum of the split*.

A **subarray** is a contiguous part of the array.

Example1:-

```
Input: nums = [7,2,5,10,8], k = 2
Output: 18
Explanation: There are four ways to split nums into two subarrays.
The best way is to split it into [7,2,5] and [10,8], where the largest sum among the two subarrays is only 18.
```

```
class Solution {
public:
    bool possible(vector<int>&nums,int k,int sum){
        int c_sum=0;
        int c_k=0;
        int n=nums.size();
        for(int i=0;i<n;i++){
            if(c_sum+nums[i]<=sum){
                c_sum+=nums[i];
            }else{
                c_k++;
                if(c_k==k)return false;
                if(nums[i]>sum)return false;

                c_sum=nums[i];
            }
        }
        return true;
    }

    int splitArray(vector<int>& nums, int k) {
        int n=nums.size();
        int l=0;
        int sum=0;
        for(int i=0;i<n;i++){
            sum+=nums[i];
        }
        int h=sum;
        int ans=-1;
        while(l<=h){
            int mid=l+(h-l)/2;
            if(possible(nums,k,mid)){
               ans=mid;
               h=mid-1;
            }else l=mid+1;
        }

        return ans;
    }
};
```

## 22.Kth Missing Positive Number

Given an array `arr` of positive integers sorted in a **strictly increasing order**, and an integer `k`.

Return *the* `k` `th` ***positive*** *integer that is* ***missing*** *from this array.*

Example1:-

```
Input: arr = [2,3,4,7,11], k = 5
Output: 9
Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.
```

```
class Solution {
public:
    // case 1:arr->[3,4,5] k=2 result=2
    // case 2:arr->[2,3,4,7,11] k=5 result=9
    // case 3:arr->[1,2,3] k=2 result=5
    int findKthPositive(vector<int>& arr, int k) {
        int n=arr.size();
        int missed=0;
        for(int i=0;i<n;i++){
            if(i==0){
                missed+=arr[0]-1;
                //case 1
                if(missed>=k){
                    return k;
                }
            }else{
                //case 2
                missed+=arr[i]-arr[i-1]-1;
                if(missed>=k){
                    missed-=arr[i]-arr[i-1]-1;
                    int result=arr[i-1];
                    while(missed<k){
                        result++;
                        missed++;
                    }
                    return result;
                }
            }
        }

        //case 3
        int result=arr[n-1];
        while(missed<k){
            result++;
            missed++;
        }
        return result;
    }
};
```

```
class Solution {
public:
    int findKthPositive(vector<int>& arr, int k) {
        int n=arr.size();
        int l=0;
        int h=n;
        while(l<h){
            int mid=l+(h-l)/2;

            if(arr[mid]-(mid+1)<k)l=mid+1;
            else h=mid;
        }

        return l+k;
    }
};
```