# Striver’s SDE Sheet

## Day 11: Binary Search

### Find Nth Root Of M

You are given two positive integers 'N'and 'M'. You have to return the 'Nth'root of 'M', i.e., 'M(I/N)'. If the 'Nth' root is not an integer, return -1.

Ans:

The basic approach will we just find root by pow

Code:

Return pow(m,1/n);

Try using binary search

Approach is very simple we notice that our answer will always lies between 1 to m in sorted order so lets apply binary search

Code:

```
int low = 1;
  int high = m;

  while(low <= high)
  {
    int mid = low + (high-low)/2;

    if(pow(mid, n) == m)
      return mid;
    else if(pow(mid, n) > m)
      high = mid - 1;
    else
      low = mid + 1;  } return -1;
```

# Matrix Median

You have been given a matrix of 'N' rows and 'M' columns filled up with integers where every row is sorted in non-decreasing order. Your task is to find the overall median of the matrix i.e if all elements of the matrix are written in a single line, then you need to return the median of that linear array. The median of a finite list of numbers is the "middle" number when those numbers are listed in order from smallest to greatest. If there is an odd number of observations, the middle one is picked. For example, consider the list of numbers [1, 3, 3, 6, 7, 8, 9]. This

list contains seven numbers. The median is the fourth of them, which is 6.

Ans:

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

int getMedian(vector<vector<int>> &matrix)

{


    int start=INT_MAX;

    int end=INT_MIN;

    int n=matrix.size();

    int m=matrix[0].size();

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

        start=min(start,matrix[i][0]);

        end=max(end,matrix[i][m-1]);

    }

    int midpos=(n*m+1)/2;

    while(start<=end){

        int mid=start+(end-start)/2;

        int count=0;

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

            int index=upper_bound(matrix[i].begin(),matrix[i].end(),mi
d)-matrix[i].begin();

            count+=index;

        }

        if(count<midpos){
```

```
            start=mid+1;
        }else{
            end=mid-1;
        }
    }
    return start;
}
```

# 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.

Ans:

Can be done with help of xor

Code:

For()

Ans^=nums[i];

Return ans;

Trying to use binary search:

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        int left = 0;
        int n = nums.size()-1;
        int right = n;
        while(left<=right){
            int mid = left+(right-left)/2;
            //edge case, if mid is single element
            if((mid == 0 || nums[mid] != nums[mid-1]) &&( mid == n || nums[mid] != nums[mid+1] )) return nums[mid];
        //as size of nums is odd logic to find which side to move
        int leftpart;
        if(nums[mid] == nums[mid-1])
        leftpart = mid-1;
        else leftpart = mid;

        if(leftpart%2 == 1) right = mid-1;
        else left = mid+1;

        }
```

```
        return 0;
    }
};
```

# 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.

Ans:
clear cut binary search implementation

Code:

```
class Solution {

public:

    int search(vector<int>& arr, int target) {

          int n = arr.size();

        int low = 0, high = n-1;

        long int mid = -1;

        while(low <= high){

            mid = low + (high-low)/2;

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

            if(arr[mid] >= arr[low]){

                /* left half sorted */

                if(target >= arr[low] && target < arr[mid]) high = mid-1;

                else low = mid+1;

            }

            else{

                /* right half is sorted */

                if( target > arr[mid] && target <= arr[high]) low = mid+1;

                else high = mid-1;

            }

        }

        return -1;

    }

};
```

# 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)).

Ans:

The basic approach will we merge this array and find median

Code:

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        for(int i=0; i<nums2.size(); i++){
            nums1.push_back(nums2[i]);
        }

        int x = nums1.size();
        sort(nums1.begin(),nums1.end());
         if(x%2!=0){
            return nums1[x/2];
        }
        int n1 = nums1[x/2];
        int n2 = nums1[(x/2)-1];

        double x2;
        x2 = (n1+n2)/(double)2;
        return x2;
    }
};
```

# 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.

ANS:

Approach came  in my mind

1) Merger the array then give ans
2) Apply two pointer method

Code:

```
class Solution {

    public long kthElement( int arr1[], int arr2[], int n, int m, int
k) {

        int i = 0, j = 0;

        int ptr = 0;

        int ans = -1;

        while (i < n && j < m) {

            if (ptr == k) {

                return ans;

            }

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

                ans = arr2[j];

                j++;

            } else {

                ans = arr1[i];

                i++;

            }

            ptr++;

        }

        while (i < n) {

            if (ptr == k) {

                return ans;

            }

            ans = arr1[i];
```

```
            i++;
            ptr++;
        }
        while (j < m) {
            if (ptr == k) {
                return ans;
            }
            ans = arr2[j];
            j++;
            ptr++;
        }
        return ans;
    }
}
```

# Allocate Books

Ayush is studying for ninjatest which will be held after 'N' days, And to score good marks he has to study 'M' chapters and the ith chapter requires TIME[i] seconds to study. The day in Ayush's world has IOOAIOO seconds. There are some rules that are followed by Ayush while studying. 1. He reads the chapter in a sequential order, i.e. he studies i+lth chapter only after he studies ith chapter.

2. If he starts some chapter on a particular day he completes it that day itself.

3. He wants to distribute his workload over 'N' days, so he wants to minimize the maximum amount of time he studies in a day.

Your task is to find out the minimal value of the maximum amount of time for which Ayush studies in a day, in order to complete all the 'M' chapters in no more than 'N' days.

Ans:

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


using namespace std;


bool isPossible(int n, int m, vector<int>& arr, long long int mid) {
    int dayCount = 1;
    long long int timeCount = 0;

    for (int i = 0; i < m; i++) {
        if (timeCount + arr[i] <= mid) {
            timeCount += arr[i];
        } else {
            dayCount++;
            if (dayCount > n || arr[i] > mid) {
                return false;
            } else {
                timeCount = arr[i];
            }
        }
    }
```

```
    return true;
}


long long int ayushGivesNinjatest(int n, int m, vector<int>& time) {
    long long int s = 0;
    long long int sum = accumulate(time.begin(), time.end(), 0LL); //
Calculate sum of time values


    long long int e = sum;
    long long int ans = -1;


    while (s <= e) {
        long long int mid = s + (e-s) / 2;
        if (isPossible(n, m, time, mid)) {
            ans = mid;
            e = mid - 1;
        } else {
            s = mid + 1;
        }
    }
    return ans;
}
```

# Aggressive Cows

You are given an array 'arr'consisting of 'n'integers which denote the position of a stall. You are also given an integer 'k' which denotes the number of aggressive cows. You are given the task of assigning stalls to 'k' cows such that the minimum distance between any two of them is the maximum possible.

Ans:

```
bool ispossible(vector<int> &stalls, int k, int mid){
    int cowCount = 1;
    int lastpos = stalls[0];
    for (int i = 0; i<stalls.size();i++){
        if(stalls[i]-lastpos>=mid){

            cowCount++;
            if(cowCount==k){
                return true;
            }
            lastpos = stalls[i];
        }
    }
    return false;
}


int aggressiveCows(vector<int> &stalls, int k)
{
    //    Write your code here.
    sort(stalls.begin(),stalls.end());
    int s = 0;
    int maxi = -1;
    for(int i = 0; i<=stalls.size(); i++){

         maxi = max(maxi,stalls[i]);
    }
```

```
    int e = maxi;
    int ans = -1;
    int mid = s +(e-s)/2;
    while(s<=e){

        if(ispossible(stalls,k,mid)){
            ans = mid;
            s = mid+1;
        }

        else{
            e = mid-1;
        }
        mid = s +(e-s)/2;
    }
    return ans;
}
```