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.

ODD EVEN ANALYSIS

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums)
    {
        int n = nums.size();
        if(n==1) return nums[0];
        if(nums[0]!=nums[1]) return nums[0];
        if(nums[n-1]!=nums[n-2]) return nums[n-1];
        int ans = -1;
        int l = 2, h = n-3;
        while(l<=h)
        {
            int mid = (l+h)/2;
            if(nums[mid]==nums[mid-1]) {
                int len = mid-2-l+1;
                if(len%2) h = mid-2;
                else l = mid+1;
            }
            else if(nums[mid]==nums[mid+1]){
                int len = h - (mid+2)+1;
                if(len%2) l = mid+2;
                else h = mid-1;
            }
            else {
                ans = nums[mid];
                break;
            }
        }
        return ans;
    }
};
```

BETTER SOLUTION

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            int mid = (left + right) / 2;
            if (mid % 2 == 1) {
                mid--;
            }
            if (nums[mid] != nums[mid + 1]) {
                right = mid;
            } else {
                left = mid + 2;
            }
        }
        return nums[left];
    }
};
```

Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2

Input: nums = [3,3,7,7,10,11,11]
Output: 10

You are given 2 numbers **(n , m)**; the task is to find $\sqrt[n]{m}$ (n^{th} root of m).

If the root is not integer then returns -1.

Constraints:

$1 <= n <= 30$

$1 <= m <= 10^9$

Expected Time Complexity: O(n* log(m))

Expected Space Complexity: O(1)

```
int NthRoot(int n, int m)
{
    int l = 1, h = m;
    int ans = -1;
    while(l<=h)
    {
        long long int mid = (l+h)/2;
        long long int midn = 1;
        for(int i=1;i<=n;i++){
            midn *= mid;
            if(midn>m) break;
        }
        if(midn==m){
            ans = mid;
            break;
        }
        else if(midn<m){
            l = mid+1;
        }
        else
        {
            h = mid-1;
        }
    }
    return ans;
}
```

ROTATED SORTED ARRAY

1. Searching with distinct values

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.

Example 1:

> **Input:** nums = [4,5,6,7,0,1,2], target = 0
> **Output:** 4

Example 2:

> **Input:** nums = [4,5,6,7,0,1,2], target = 3
> **Output:** -1

Example 3:

> **Input:** nums = [1], target = 0
> **Output:** -1

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

2. Searching with repeated values

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.

nums =

`[1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1]`

target =

`2`

Output

`false`

Expected

`true`

- Previous code will fail for this input
- Just comparing with the last element wont work

```cpp
bool search(vector<int>& nums, int target) {
    int l = 0, h = nums.size()-1;
    while(l<=h){
        int mid = (l+h)/2;
        if(nums[mid]==target) return 1;
        if(nums[l]==nums[mid] && nums[h]==nums[mid]){
            l++; h--;
            continue;
        }
        if(nums[mid]<=nums[h]) //right half is sorted
        {
            if(target>=nums[mid] && target<=nums[h]) l = mid+1;
            else h = mid-1;
        }
        else //left half is sorted
        {
            if(target>=nums[l] && target<=nums[mid]) h = mid-1;
            else l = mid+1;
        }
    }
    return 0;
}
```

3. Finding Minimum

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.`

```
int findMin(vector<int>& nums) {
    int l = 0, h = nums.size()-1;
    int ans = nums[0];
    while(l<=h){
        int mid = (l+h)/2;
        ans = min(ans,nums[mid]);
        if(nums[mid]<=nums[h]) //right part sorted
        {
            h = mid - 1;
        }
        else //left part sorted
        {
            ans = min(ans,nums[l]);
            l = mid + 1;
        }
    }
    return ans;
}
```

No. of Rotations is given by the index of minimum element

```
int findKRotation(int arr[], int n) {
    int l = 0, h = n-1;
    int ind = 0;
    while(l<=h){
        int mid = (l+h)/2;
        if(arr[mid]<arr[ind]) ind = mid;

        if(arr[mid]<=arr[h]){ //right part is sorted
            h = mid - 1;
        }else{ //left part is sorted
            if(arr[l]<arr[ind]) ind = l;
            l = mid + 1;
        }
    }
    return ind;
}
```

PEAK ELEMENT - 1D

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.

eg: $-\infty$ [1,2,1,3,5,6,4] $-\infty$ lo = 0, hi = n-1

mid

a) Peak case

if (arr[mid] > arr[mid-1] && arr[mid] > arr[mid+1]) return mid

b) Sorted

arr[mid-1] < arr[mid] < arr[mid+1] } ⇒ we will definetly find a peak in l = mid+1 to h

n (-∞)

c) Reverse sorted ⇒ we will definetly find a peak in (l, mid-1)

d) Valley

} peak available on both (l, mid-1) & (mid+1, h)

```cpp
int findPeakElement(vector<int>& nums) {
    int n = nums.size();
    if(n==1) return 0;
    if(nums[0]>nums[1]) return 0;
    if(nums[n-1]>nums[n-2]) return n-1;

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

PEAK ELEMENT - 2D

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.

```cpp
int ij(vector<vector<int>>& mat, int i,int j){
    int n = mat.size(), m = mat[0].size();
    if(i<0 || j<0 || i>=n || j>=m) return -1;
    return mat[i][j];
}
vector<int> findPeakGrid(vector<vector<int>>& mat)
{
    int n = mat.size(), m = mat[0].size();
    int l = 0,h = m-1;
    int peak = -1;
    vector<int> ans(2);
    while(l<=h)
    {
        int mid = (l+h)/2;
        int ma = 0;
        for(int i=0;i<n;i++) {
            if(mat[i][mid]>mat[ma][mid]) ma = i;
        }
        if(ij(mat,ma,mid)>ij(mat,ma,mid+1) && ij(mat,ma,mid)>ij(mat,ma,mid-1)){
            ans[0] = ma; ans[1] = mid;
            return ans;
        }
        else if(ij(mat,ma,mid)<ij(mat,ma,mid+1)) l = mid+1;
        else h = mid-1;
    }
    return ans;
}
```

See Striver's video for explanation

BINARY SEARCH ON ANSWER

Farmer John has built a new long barn, with N (2 <= N <= 100,000) stalls. The stalls are located along a straight line at positions x1 ... xN (0 <= xi <= 1,000,000,000).

His C (2 <= C <= N) cows don't like this barn layout and become aggressive towards each other once put into a stall. To prevent the cows from hurting each other, FJ wants to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

```cpp
#include<bits/stdc++.h>
using namespace std;
using lli = long long int;

lli n,c;
vector<lli> x;
int check(int mid)
{
    int cows = 1, cur = x[0]; //FIRST COW IS PLACED AT X[0]
    for(int i=1;i<n;i++){
        if(x[i]-cur>=mid){
            cur = x[i];
            cows++;
        }
    }
    return (cows>=c);
}
```

```cpp
int main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);

    int t;cin>>t;
    while(t--)
    {
        cin>>n>>c;
        x.assign(n,0);
        for(int i=0;i<n;i++) cin>>x[i];
        sort(x.begin(),x.end());

        lli l = 0, h = 1e9;
        lli ans = 0;
        while(l<=h){
            lli mid = (l+h)/2;
            if(check(mid)){
                ans = mid;
                l = mid+1;
            }else h = mid-1;
        }
        cout<<ans<<'\n';
    }
}
```

Minimize Max Distance to Gas Station

We have a horizontal number line. On that number line, we have gas **stations** at positions stations[0], stations[1], ..., stations[N-1], where **n** = size of the stations array. Now, we add **k** more gas stations so that **d**, the maximum distance between adjacent gas stations, is minimized. We have to find the smallest possible value of d. Find the answer **exactly** to 2 decimal places.

Example 1:

Input:
n = 10
stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
k = 9
Output: 0.50
Explanation: Each of the 9 stations can be added mid way between all the existing adjacent stations.

USING PRIORITY QUEUE

```cpp
#include<bits/stdc++.h>
double minimiseMaxDistance(vector<int> &arr, int k) {
    int n = arr.size();
    vector<int> divi;
    priority_queue<pair<long double, int>> pq;
    for (int i = 0; i < n - 1; i++) {
        divi.push_back(1);
        pq.push({1.0 * (arr[i + 1] - arr[i]), i});
    }
    for (int i = 0; i < k; i++) {
        auto temp = pq.top();
        pq.pop();
        int ind = temp.second;
        double cur_dis = temp.first;
        double new_dis = (1.0 * cur_dis * divi[ind]) / (divi[ind] + 1.0);
        divi[ind]++;
        pq.push({new_dis, ind});
    }
    return pq.top().first;
}
```

Binary Search : Problem will always tell you to which decimal place they want the answer

How to apply ??

```
while ( h - l > 10^-6 )
{
    double mid = (l+h)/2 ;
    (check condition)
        ↳ low = mid;
        ↳ high = mid;
}
```

Dont do -1 or +1

```cpp
#include<bits/stdc++.h>
bool check(long double mid, vector<int> &arr, int k) {
    int n = arr.size();
    int req = 0;
    for(int i=0;i<n-1;i++)
    {
        long double dis = arr[i+1]-arr[i];
        req += (dis)/mid;
        if(mid*req == dis) req--;
        if(req>k) return false;
    }
    return true;
}
double minimiseMaxDistance(vector<int> &arr, int k) {
    long double l = 0, h = 1e9, diff = 1e-6, ans = h;
    while(h-l>diff){
        long double mid = (l+h)/2.0;
        if(check(mid,arr,k)){
            ans = mid;
            h = mid;
        }
        else l = mid;
    }
    return ans;
}
```

K-th element of two 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.

arr1[] (n): { 2, 2, 3, 6, 7, 9 } ↑ arr2[] (m): { 1, 2, 4, 8, 10 } ↑ K = 5

merging process: {1, 2, 2, 2, 3, 4, 6, 7, 8, 9, 10}

if arr1[i] == arr2[j] → arr1[i] is pushed

Now,

no. of element in merged array when arr1[i] is pushed

cnt1 = i + 1 } from arr1

cnt2 = lower_bound (arr2, arr2 + m, arr1[i]) - arr2

So, total = cnt1 + cnt2 {1, 2, 2, 2, 3, 4, 6, 7, 8, 9, 10}

for eg: when arr1[1] is pushed, total = 3

when arr1[2] is pushed, total = 5

no. of element in merged array when arr2[j] is pushed

cnt2 = j + 1

cnt1 = upper_bound (arr1, arr1 + n, arr2[j]) - arr1

{1, 2, 2, 2, 3, 4, 6, 7, 8, 9, 10}

for eg: when arr2[1] is pushed, total = 4

when arr2[2] is pushed, total = 6

Note: The difference b/w lower_bound & upper_bound

```cpp
class Solution{
    public:
    int kthElement(int arr1[], int arr2[], int n, int m, int k)
    {
        int l = 0, h = n-1;
        while(l<=h)
        {
            int mid = (l+h)/2;
            int cnt = mid+1;
            int taken = lower_bound(arr2,arr2+m,arr1[mid]) - arr2;
            cnt += taken;
            if(cnt==k) return arr1[mid];
            else if(cnt>k) h = mid-1;
            else l = mid+1;
        }
        l = 0, h = m-1;
        while(l<=h)
        {
            int mid = (l+h)/2;
            int cnt = mid+1;
            int taken = upper_bound(arr1,arr1+n,arr2[mid]) - arr1;
            cnt += taken;
            if(cnt==k) return arr2[mid];
            else if(cnt>k) h = mid-1;
            else l = mid+1;
        }
    }
};
```

Same could be used to find median of two sorted array

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

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.

Expected Time Complexity: O(32 * R * log(C))
Expected Auxiliary Space: O(1)

Constraints:
1 <= R, C <= 400
1 <= matrix[i][j] <= 2000

Binary Search on Ans (Median Range)

```cpp
int median(vector<vector<int>> &mat, int n, int m){
    int l = 2000, h = 1;
    for(int i=0;i<n;i++) l = min(l,mat[i][0]);
    for(int i=0;i<n;i++) h = max(h,mat[i][m-1]);

    int ans = h;
    int half = n*m/2;
    while(l<=h)
    {
        int mid = (l+h)/2;
        int cnt = 0, temp = h;
        for(int i=0;i<n;i++){
            cnt += upper_bound(mat[i].begin(),mat[i].end(),mid) - mat[i].begin();

            if(lower_bound(mat[i].begin(),mat[i].end(),mid)!=mat[i].end()){
                temp = min(*lower_bound(mat[i].begin(),mat[i].end(),mid),temp);
            }
        }
        if(cnt>half){
            ans = temp;
            h = mid-1;
        }else{
            l = mid+1;
        }
    }
    return ans;
}
```

You are given an `m x n` integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer `target`, return `true` *if* `target` *is in* `matrix` *or* `false` *otherwise.*

You must write a solution in `O(log(m * n))` time complexity.

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        int s = 0;
        int e = n*m-1;
        while(s<=e){
            int mid = s+(e-s)/2;

            if(matrix[mid/m][mid%m] == target)return true;

            else if(matrix[mid/m][mid%m] > target)e = mid-1;

            else s = mid+1;
        }
        return false;
    }
};
```

```
bool searchMatrix(vector<vector<int>>& matrix, int target)
{
    int n = matrix.size(), m = matrix[0].size();
    if(target>matrix[n-1][m-1]) return false;
    int l = 0, h = n-1;
    int r = h;
    while(l<=h){
        int mid = (l+h)/2;
        if(target<=matrix[mid][m-1]){
            r = mid;
            h = mid-1;
        }else{
            l = mid+1;
        }
    }
    l = 0, h = m-1;
    while(l<=h){
        int mid = (l+h)/2;
        if(matrix[r][mid]==target) return true;
        else if(matrix[r][mid]<target) l = mid + 1;
        else h = mid - 1;
    }
    return false;
}
```

TC - log(m) + log(n) = log(mn)

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 in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

```cpp
bool search_row(int r, vector<vector<int>>& matrix, int target) {
    int m = matrix[0].size();
    int l = 0, h = m-1;
    while(l<=h){
        int mid = (l+h)/2;
        if(matrix[r][mid]==target) return true;
        else if(matrix[r][mid]<target) l = mid+1;
        else h = mid - 1;
    }
    return false;
}
bool searchMatrix(vector<vector<int>>& matrix, int target)
{
    int n = matrix.size(), m = matrix[0].size();
    if(target>matrix[n-1][m-1]) return false;
    for(int i=0;i<n;i++)
    {
        if(target>=matrix[i][0] && target<=matrix[i][m-1])
        {
            if(search_row(i,matrix,target)){
                return true;
                break;
            }
        }
    }
    return false;
}
```

TC - min(n, m)*log(max(n, m))