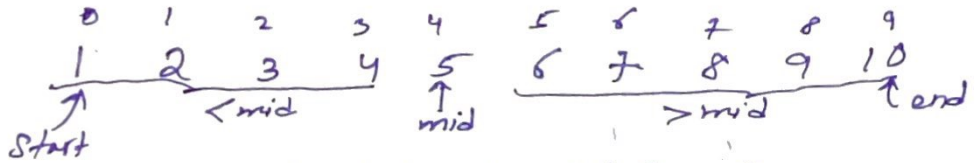
Binary Search

Note : "sorted" agar kahi bhi aaya → BS lag sakta hai (socho)
sorted hai toh TC aur reduce kr sakte ho.

→ Binary Search –

arr[] → {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} (sorted)
(0 1 2 3 4 5 6 7 8 9)

element = 2
↳ return index ie 1



$$mid = \frac{start + end}{2} = \frac{0+9}{2} = 4$$

```
int BS(int n, int arr[], int ele)
{
  int start = 0;
  int end = n-1;
  while (start <= end)
  {
    int mid = (start+end)/2;
    if (ele == arr[mid])
        return mid;
    else if (ele < arr[mid])
        end = mid-1;
    else
        start = mid+1;
  }
  return -1; //not present
}
```

$$T(n) = T\left(\frac{n}{2}\right) + 1$$
$$= O(\log n)$$
$$SC = O(1)$$

$\equiv start + \frac{(end - start)}{2}$ (for no OV problem)

→ BS on reverse sorted array –

↳ Descending

arr[] → {20, 17, 15, 14, 13, 12, 10, 9, 8, 4}
(0 1 2 3 4 5 6 7 8 9)

search for 4

for this, 20 17 15 14 13 12 10 9 8 4
(0 1 2 3 4 5 6 7 8 9)
s, mid, e

$$mid = \frac{s+e}{2} = \frac{0+9}{2} = 4$$ & a[4] ≠ 4
& 4 < 13
∴ go right (for desc order)

```
int start = 0, end = n-1;
while (start <= end)
{
    int mid = start + (end - start)/2;
    if (a[mid] == ele)
        return mid;
    else if (ele < a[mid])
        start = mid+1;
    else
        end = mid-1;
}
return -1;
```

Order Agnostic search

unknown: it is not known whether array is ascending/descending

Given an array & it is said it is sorted (but not known asc/desc) & a Key. Return the index of the key if present.

```
if (n==1) {
    if (a[0] == Key) // if size of array = 1
        return 0;
    return -1;
}
int firstele = a[0];
int lastele = a[n-1];
if (a[0] < a[n-1]) //ascending
{
    call BS for ascending order
}
else // descending
{
    call BS for descending
}
return -1; // not found
```

→ first & last occurrence of an element -

arr[] → {2, 4, 10, 10, 10, 18, 20}
(indices 0 1 2 3 4 5 6)

say, ele = 10

first occurrence = 2

last occurrence = 4

```
int BSfirst (int arr[], int n, int Key)
{
   int start = 0, end = n-1, res = -1;
   while (start <= end)
   {
      int mid = (start + end)/2;
      if (arr[mid] == Key)
      {
         res = mid;
         end = mid-1; //go left for first occurrence
      }
      else if (arr[mid] > Key)
         end = mid-1;
      else
         start = mid+1;
   }
   return res; //will have first occurrence
}

int BSlast (int arr[], int n, int Key)
{
   int start = 0, end = n-1, res = -1;
   while (start <= end)
   {
      int mid = (start + end)/2;
      if (arr[mid] == Key)
      {
         res = mid;
         start = mid+1; //go right for last occurrence
      }
      else if (arr[mid] > Key)
         end = mid-1;
      else
         start = mid+1;
   }
   return res; //will have last occurrence
}
```

$TC = O(\log n) + O(\log n)$

$SC = O(1)$

if asked : Count of an element in a sorted array-

arr[] → { 0:2, 1:4, 2:10, 3:10, 4:10, 5:18, 6:24 }

asked for 10 → first occur (2), last occur (4)

→ ans = 3

```
int count (int arr[], int n, int K)
{
    int start = BSfirst(arr, n, K);
    int end = BSlast(arr, n, K);
    return (end - start + 1);
}
```

if start = -1 then end = -1 also, element does not exist

check if start ≠ -1 then return end - start + 1 else return 0

→ Number of times a sorted array is rotated -

Consider an array of distinct nos. sorted in ascending order. The array has been rotated clockwise 'K' no. of times. Given such an array, find K.

i/p arr[] → { 0:2, 1:5, 2:6, 3:8, 4:11, 5:12, 6:15, 7:18 }

rotated 4 times (2)

∴ arr[] given is → { 11, 12, 15, 18, 2, 5, 6, 8 }

i/p arr[] → { 0:7, 1:9, 2:11, 3:12, 4:5 }

o/p = 4

i/p arr[] → { 0:7, 1:9, 2:11, 3:12, 4:15 }

o/p = 0

approach

When array is already sorted -

ie

0	1	2	3	4	5	6	7
2	5	6	8	11	12	15	18

Here, min element = 2
& it is already sorted (not rotated)

min element = 2 at index = 0
∴ it is rotated 0 times

When array is -

0	1	2	3	4	5	6	7
11	12	15	18	2	5	6	8

(rotated array)

min element at 4
∴ #times array rotated = 4

Conclusion

∴ #times array rotated = index of min element

** Question boils down to find index of min element using BS

now

0	1	2	3	4	5	6	7
11	12	15	18	2	5	6	8
↑ start							↑ end

Imp) if a[start] <= a[end] → true → array is sorted
∴ return 0.

2) To find min element -

0	1	2	3	4	5	6	7
11	12	15	18	2	5	6	8
				↑ say this is mid			

if you see, only mid element is < than prev element
∴ if (a[mid] < a[mid-1])
return mid.

- factor? to move left or right of mid.

(1)

0	1	2	3	4	5	6	7
11	12	15	18	2	5	6	8
↑ s			↑ mid				↑ e

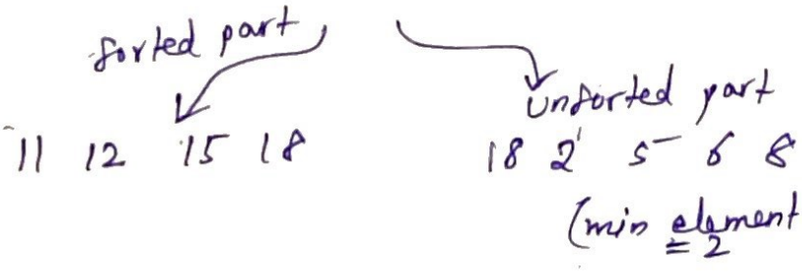
$mid = \frac{s+e}{2} = \frac{0+7}{2} = 3$

if we compare -

11	12	15	18	2	5	6	8
↑ s			↑ m				↑ e

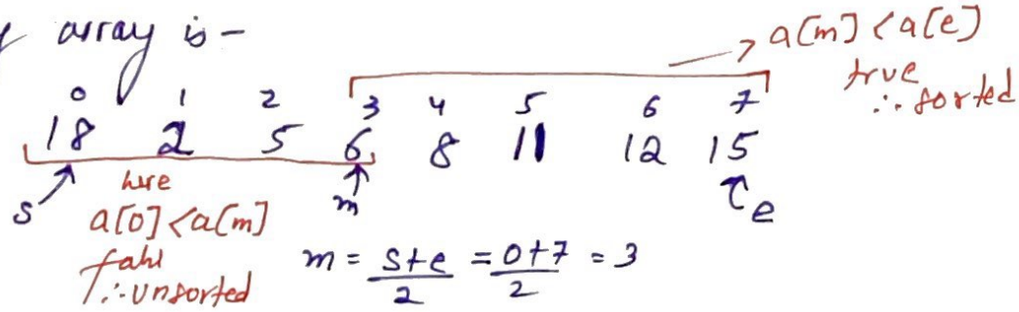
if a[s] < a[m] here, true ∴ array is sorted

→ a[m] < a[e] false ∴ unsorted part



(min element lies in unsorted part)

(2) say array is -



$m = \frac{s+e}{2} = \frac{0+7}{2} = 3$

here, also min element lies in unsorted part

∴ min element will lie in unsorted part

↳ ∴ if first part is sorted go to second part unsorted

↳ vice versa

```
int f(vector<int> &nums)
{
  int n=nums.size();
  int start=0, end=n-1;
  if(nums[start] <= nums[end]) //array already sorted
    return 0;
  while(start <= end)
  {
    int mid=(start+end)/2;
    if(nums[mid] < nums[(mid-1+n)%n]) //we get min element
      return mid;
    else if(nums[start] <= nums[mid])
      start=mid+1; //left part is sorted, ∴ go to right part
    else if(nums[mid] <= nums[end])
      end=mid-1;
  }
  return -1;
}
```

This works for unique elements

if you don't want to write this: if(m>0 && ----)

prev → (0-1) = -1 → will give array out of bound; (-1+n)%n

right part sorted → go to left part

with Duplicates

```
int l=0, h=n-1;
while (l<=h)
{
  while(l<h && a[l]==a[l+1])
    ++l;
  while(l<h && a[h]==a[h-1])
    --h;
  int m = (l+h)/2;
  if(m>0 && a[m]<a[m-1])
    return m;
  else if(a[0]<=a[m])
    l=m+1;
  else
    h=m-1;
}
return -1;
```

not l<=h

Dry run on

• 1 1 1 1 1 1 1 2 1 1 1 1

• 1 0 1 1 1

→ find an element in a rotated sorted Array –

↳ return it's index if present else -1

arr[] → 11, 12, 15, 18, 2, 5, 6, 8 (indices 0 1 2 3 4 5 6 7)

previously we find index of min element using BS

∴ index of min element = 4

11, 12, 15, 18 (sorted) | 2 | 5, 6, 8 (sorted)

Cin

```
int search (vector<int> &a, int target)
{
    int n = a.size();
    if (a[0] <= a[n-1]) // array is sorted.
        return BS(a, target, 0, n-1);
```

(BS: normal BS; 0: start; n-1: end)

if element is present return its index else -1

```
    int minInd = findMinInd(a); // gives index of min element
    if (target == a[minInd])
        return minInd;
    if (target >= a[0])
        return BS(a, target, 0, minInd-1);
    return BS(a, target, minInd+1, n-1);
}
```

check for -1 returned if returned then

go to left part

go to right

TC = O(log n) & SC = O(1)

→ Searching in a nearly sorted array –

Given an array which is sorted but after sorting, some elements are moved to either of the adjacent positions ie arr[i] may be at arr[i+1] or arr[i-1]. Search for an element & return it's index.

a[] → 5, 10, 30, 20, 40 (0, 1, 2, 3, 4)

nearly sorted [here, element at ith index can be at i-1, i, i+1]

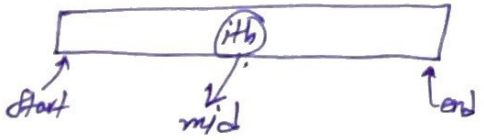
for a[], sorted order is -

sorted: 5, 10, 20, 30, 40

given a[]: 5, 10, 30, 20, 40

now,

normal BS for sorted array

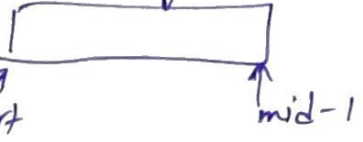


here, since purely sorted ith index wala ith pe hi rahega

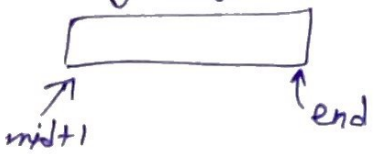
here, ele == arr[mid]

by comparing with a[mid]

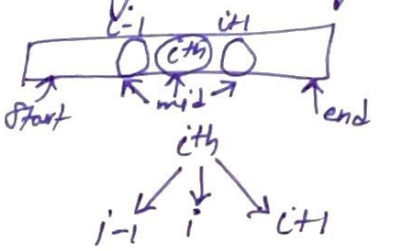
here, to go left



to go right



Modified BS for nearly sorted array



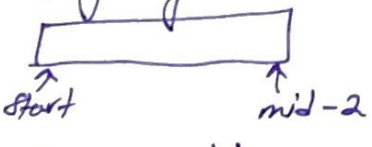
ele == arr[mid]

ele == arr[mid-1]

ele == arr[mid+1]

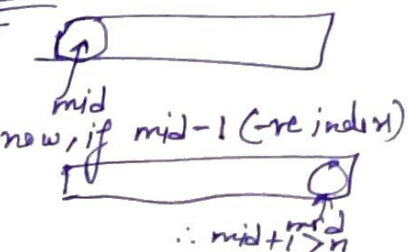
here, to go left

by comparing with a[mid]



to go right

Not



∴ mid+1 > n

∴ checks for this

ie mid-1 >= start & mid+1 <= end

code

```
int search (vector<int> &arr, int target)
{
  int n = arr.size();
  int start = 0, end = n-1;
  while (start <= end)
  {
    int mid = (start + end)/2;
    if (arr[mid] == target)
      return mid;
    else if (mid-1 >= start && arr[mid-1] == target)
      return mid-1;
    else if (mid+1 <= end && arr[mid+1] == target)
      return mid+1;
    else
    {
      if (target < arr[mid])
        end = mid-2;
      else
        start = mid+2;
    }
  }
  return -1;
}
```

TC = O(log n)

→ find floor of an element in a sorted array-

given a sorted array & a value 'x', the floor of x is the largest element in array smaller or equal to x.

i/p arr[] → {1, 2, 8, 10, 10, 12, 19} & x = 5

o/p = 2

approach

a[] → {1, 2, 3, 4, 8, 10, 10, 12, 29} & x = 5

(indices 0 1 2 3 4 5 6 7 8; s → 1, m → 8, e → 29)

$m = \frac{0+8}{2} = 4$

Note: if (a[m] == x) return m; → or a[m] // as x already present

- now, a[m] = 8 > x
  ∴ go to left
  e = m-1 = 3

(indices 0 1 2 3 4 5 6 7 8; s → 1, m → 2, e → 4)

$m = \frac{0+3}{2} = 1$ & a[m] = 2 < x

∴ res = a[m] (as of now)
& go to right i.e. s = m+1 = 2

0	1	2	3	4	5	6	7	8
1	2	3	4	8	10	10	12	29

s m e

$m = \frac{2+3}{2} = 2$

& a[m] = 3 < x

∴ res = ~~2~~ 3

& go to right ∴ s = m+1 = 3

0	1	2	3	4	5	6	7	8
1	2	3	4	8	10	10	12	29

s e

$m = \frac{3+3}{2} = 3$

& a[m] = 4 < x

∴ res = ~~3~~ 4

& go to right i.e s = m+1 = 5

now, s=5, e=4, ∴ terminate, ans = res = 4 (A1)

```
int floor(vector<int> a, int x)
{
    int start=0, end = a.size()-1;
    int res=-1;
    while(start<=end)
    {
        int mid = (start+end)/2;
        if(a[mid]==x)
            return x;
        if(a[mid]<x) // now we want element <=x but closer to x.
        {
            res=a[mid];
            start = mid+1;
        }
        else if(a[mid]>x){
            end = mid-1;
        }
    }
    return res;
}
```

TC = O(log n)

→ To find ceil of an element in a sorted array -

↳ ceil of x is smallest element in array greater than or equal to x.

arr[] → {1 2 3 4 8 10 10 12 19}
(0 1 2 3 4 5 6 7 8)

$x = 5$ ⟶ Ans = 8

```
int ceil (vector<int> a, int x)
{
    int start = 0, end = a.size()-1;
    int res = -1;
    while(start <= end)
    {
        int mid = (start + end)/2;
        if(a[mid] == x)
            return x;
        if(a[mid] > x)
        {
            res = a[mid];
            end = mid-1;
        }
        else if(a[mid] < x)
            start = mid+1;
    }
    return res;
}
```

→ Next alphabetical letter -

↳ given an array of letters in ascending order, find the smallest letter in the array which is greater than a given Key letter.

eg) a[] → {a, c, f, h} & Key = f

∴ o/p = h

It is same as ceil problem but here

if(a[mid] == Key)

we don't return as we want the next character, ∴ we move to right part

i.e. start = mid+1

Note: if target/key = 'z'
& array[] → {a, b}
∴ return a[0];
↳ first character

```
char nextletter(vector<char>& letters, char target)
{
    int res = -1;
    int n = letters.size();
    int start=0, end = n-1, res=-1;
    while( start<= end)
    {
        int mid = (start + end)/2;
        if( letters[mid] == target)
            start = mid+1;
        else if (letters[mid] > target)
        {
            res = mid;
            end = mid-1;
        }
        else
            start = mid+1;
    }
    if(res == -1)
        return letter[0];
    return letter[res];
}
```

TC = O(log n)

→ find position of an element in an infinite sorted array =

[1 2 3 10 12 17 18 ------]

∞ sorted array

start=0 but end = ??

to apply BS → we should have bounded region & start & end are fixed.

but here end = ??

```
int BSinfinite(int arr[], int Key)
{
  int start=0;
  int end= 1; //Keep it 1
  while (arr[end] < Key)
    {start= end;
     end = end*2;
    }
  //after this we get our bounded region
  //now simply apply normal BS
  return BS(arr, start, end);
}
```

TC = O(log n)

start ↑ end

TC = O(log n) + O(log n)

SC = O(1)

→ Index of first 1 in a Binary Sorted infinite Array-

Given an infinite sorted array consisting of 1's & 0's. find the index of the first 1 in that array.

arr[] → { 0 00000.....0 11111.....}

return this '1' index.

```
int BSmodify (int arr[])
{
  int start =0, end =1;
  int res =-1;
  while(arr[end] != 1)
    {
      start = end;
      end = end*2;
    }
  while (start <= end)
    {
    int mid =(start +end)/2;
      if(arr[mid] == 1)
        {res = mid;
         end = mid-1;
        }
      else if(arr[mid] ==0)
         start = mid+1;
      else if(arr[mid] == 1)
         end = mid-1;
    }
  return res;
}
```

for last occr:

```
while (l<=h)
  {
   int m = (l+h)/2;
   if(a[m] == 1)
     {ans = 1;
      l = m+1;
     }
   else if(a[m]==0)
      l=m+1;
   else
      h=m-1;
  }
return ans;
```

TC = O(log n) + O(log n)

while BS

→ Minimum difference element in a sorted array-

⇓ Given a sorted array, find the element in the array which has minimum (absolute) difference with given Key.

say arr[] → {4, 6, 10} & Key = 7

-7	-7	-7
-3	-1	3
	↓ (abs)	
3	1	3
	↓ min	

∴ ans = 6

Note: if Key is present in array-

1, 3, 8, 10, 12, 15 & Key = 12

-12 -12 -12 -12 -12 -12

abs = 11 9 4 2 0 15

↳ min

∴ Ans = 12 → Key only **

if Key is not present in array-

1, 3, 8, 10, 15 & Key = 12

then find floor & ceil of Key

& find the difference & which is min, that will be ans

here, floor of 12 = 10

ceil of 12 = 15

∴ abs(10-12) = 2) 2 is min

abs(15-12) = 3)

∴ Ans = 10

M1-

```
int MinDiff (int arr[], int n, int Key)
{
  int floor = BSfloor(arr, n, Key);   → return index
  if (arr[floor] == Key) // if Key present in array
      return Key;   → or arr[floor]
  int ceil = BSceil(arr, n, Key);
  if (floor == -1) // ie Key < arr[0]
      return arr[ceil];
  if (ceil == -1) // Key > arr[n-1]
      return arr[floor];
  int x = abs(arr[floor] - Key);
  int y = abs(arr[ceil] - Key);
  if (x < y)
      return arr[floor];
  else
      return arr[ceil];
}
```

here, TC = O(logn) + O(logn) = O(2.logn)
ceil ↓ floor ↓

M2-

** after complete BS is done

In normal BS, for given Key if Key is not present in array → start points to ceil & End points to floor

↳ if (key < arr[n-1]) if (key > arr[n-1]) ↳ ceil is -1

↳ if (key > arr[0]) if (key < arr[0]) ↳ floor is -1

if Key is present → it is ceil & floor (same)

We will do in 1 iteration of BS only-

we can use same concept to find floor & ceil

```
int minDiff(int arr[], int n, int key)
{
  int floor=0, ceil=0;
  if(key<arr[0]) //floor won't exist
    floor=-1
  if(key>arr[n-1]) // ceil won't exist
    ceil=-1;

  //Doing BS now
  int start=0, end=n-1;
  while(start<=end)
  {
    int mid=(start+end)/2;
    if(arr[mid]==key)
      return arr[mid];
    else if(key<arr[mid])
      end=mid-1;
    else
      start=mid+1;
  }
  //now compare → both floor & ceil can't be -1 simultaneously as both lie in opposite direction.
  if(floor==-1)
    return arr[start]; ↳ ceil val
  if(ceil==-1)
    return arr[end]; → floor val
  //if both floor & ceil not = -1
  int x=abs(arr[start]-key), y=abs(arr[end]-key);
  if(x<y)
    return arr[start];
  else return arr[end];
}
```

→ TC = O(logn) only

→ Binary Search on Answer –

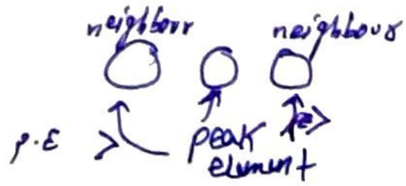
Aisa question mein array unsorted bhi hota hai, usme BS lag jata hai.

→ Peak element –

find peak element in an array. A peak element is an element that is greater than it's neighbours.
Given an input array nums, where nums[i] != nums[i+1], find a peak element & return it's index.

arr[] → {5, 10, 20, 15}
↳ o/p = 2

unsorted



peak element can be more than 1.

eg arr[] → {10, 20, 15, 2, 23, 90, 67} (indices 0 1 2 3 4 5 6)

peak element (at 20), peak element (at 90)

can return 1 or 5

** eg → arr[] → 10, 20, 30, 40, 50 → nothing
50 > 40
∴ 50 is PE

Similarly → X 50, 40, 30, 20, 10
50 > 40
∴ 50 is peak element

How we know BS on Answer (BSA) logic?

0	1	2	3
5	10	20	15

{unsorted}

s = 0, m = 1, e = 3

$m = \frac{0+3}{2} = 1$

now, check if(a[m] > a[m-1] && a[m] > a[m+1])
return m;

but if it is not the case, then where to move? left/right

here, a[m+1] > a[m] (20 > 10)

a[m-1] < a[m] (5 < 10)

↓

better move in this direction (more promising)

- Also see edge case where m-1 >= 0 & m+1 < n


```
int start = 0, end = n-1;

if (n == 1) //edge cases
    return 0;          → index 0;

if (nums[0] > nums[1])
    return nums[0];
if (nums[n-1] > nums[n-2]) //edge cases
    return nums[n-1];

TC = O(logn)

while (start <= end)
{
    int mid = (start+end)/2;
    if (mid > 0 && mid < n)
        if (arr[mid] > arr[mid+1] && arr[mid] > arr[mid-1])
            return mid;
        else if (arr[mid] < arr[mid+1])
            start = mid+1;
        else end = mid-1;
    else //to check for boundary condition
    {
        if (mid == 0) or 0 → 1
            if (arr[mid] > arr[mid+1])
                return mid
            else return mid+1;
        else if (mid == n-1)
        {
            if (arr[mid] > arr[mid-1])
                return mid;
            else return mid-1;
        }
    }
}
return -1; //not exist
```

→ finding maximum element in bitonic Array-

Given bitonic array find the maximum value of array.
An array is said to be bitonic if it has an increasing sequence of integers followed immediately by a decreasing sequence of integers.

a[] → 1 3 8 12 4 2 (unsorted)

bitonic is montonically ↑ then monotonically ↓. (∵ only 1 peak element)

i.e. 

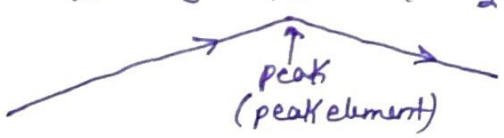
or 

or 

(3 cases)

Note- 5 5 7 15

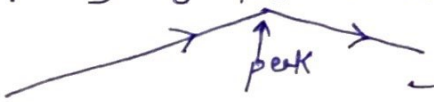
for this, 1 3 8 12 4 2



It is same as finding peak element (same code)

→ Search an element in Bitonic Array-
↳ return the index of element

1 3 8 12 4 2 & Key = 4



∴ [1 3 8] & [4 2]
sorted ascending, sorted descending

```
int search(int arr[], int n, int Key)
{
    int peak = peakElement(arr, n);
    if (arr[peak] == Key)
        return peak;
    if (peak > 0)
    int leftpart = BSascend(arr, n, 0, peak-1);   // start, end
    if (arr[leftpart] == Key) return leftpart;
    leftpart != -1
```

```
int rightpart = BSdec(arr, n, peak+1, n-1);
if (rightpart != -1 && arr[rightpart] == Key)
        return rightpart;
    return -1; //not found;
}
```

$TC = 3 * O(\log n)$

Search in row wise & column wise sorted array-

Given an nxm matrix & a number x, find the position of x in the matrix if it is present. Otherwise print "Not found". In given matrix, every row & every column is sorted in increasing order.

every row sorted (ascend)

arr[][] →				
	10	20	30	40
	15	25	35	45
	27	29	37	48
	32	33	39	50

every col'm sorted in ascend order

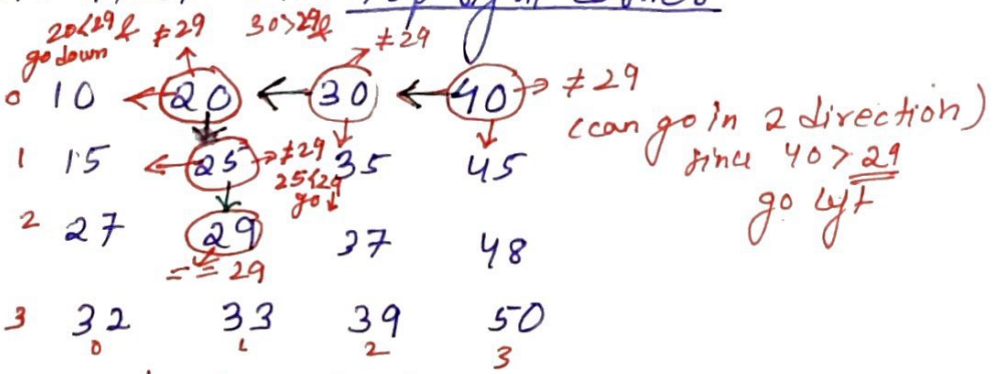
& Key = 29

Brute force-

compare every element using 2 loops → $TC = O(n^2)$ with Key

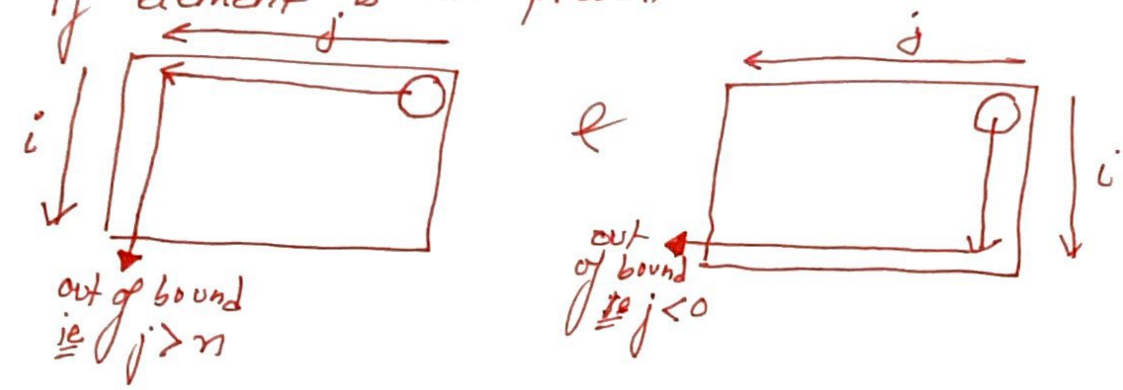
optimal

We will ~~start~~ with top right corner

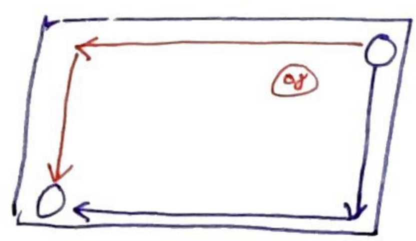


∴ return (2,1)

Note : if element is not present



here WC occurs if element is present in left bottom corner.



TC = O(n+m)

```
pair<int,int> search (vector<vector<int>> matrix, int n, int m, int x)
{
    int i=0, j=m-1; // index of top corner element.
    pair<int,int> ans;
    ans.first=-1;
    ans.second=-1;
    while(i<n && j>=0)
    {
        if(matrix[i][j]==x)
        {
            ans.first=i;
            ans.second=j;
            return ans;
        }
        else if (matrix[i][j]>x)
            --j;
        else if (matrix[i][j]<x)
            ++i;
    }
    return ans; // {-1,-1} //not present
}
```

TC = O(n+m)

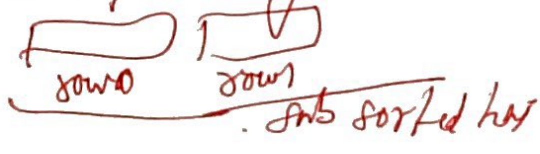
this method works for row is sorted & col'm is sorted

only condition

eg 10 12 15
11 13 16

11<15

& also for: if any row is kept side by side



Leetcode → In leetcode → only mentioned
- each row sorted in ↓ order
- first integer of each row is greater than last integer of previous row.

	0	1	2	3
0	1 (0)	3 (1)	5 (2)	7 (3)
1	10 (4)	11 (5)	16 (6)	20 (7)
2	23 (8)	30 (9)	34 (10)	50 (11)

n = 3
m = 4

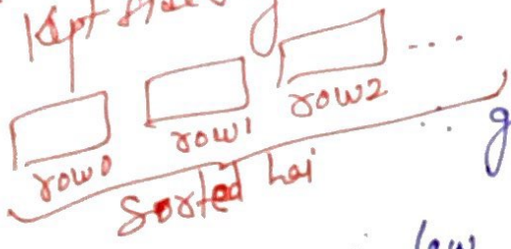
& target = 30

1	3	5	7
10	11	16	20
23	30	34	50

7 < 10
20 <

we will discuss for this

this method works only for when every row if kept side by side -



low = 0
high = 11 (3×4 − 1)
mid

rowind = 5/4 = 1 (m)
colind = 5 % 4 (m) = 1

→ mid = (0+11)/2 = 5 & 5 corresponds to (1,1)

& a[1][1] = 11 < 30

∴ go to right half

low = 5+1 = 6 (mid)
high = 11
mid = (6+11)/2 = 8

≡ 8/4 = 2, 8/4 = 0 ∴ 8 ≡ (2, 0)

& a[2][0] = 23 < 30

go to right half

low = 9 (8+1)
high = 11
mid = (9+11)/2 = 10 → ≡ 10/4 = 2, 10 % 4 = 2 (2, 2)

∴ a[2][2] = 34 > 30

go to lft part

low = 9
high = 9 (10 − 1)
mid = (9+9)/2 = 9 → ≡ 9/4 = 2, 9 % 4 = 1 (2, 1)

& a[2][1] = 29
return

- $TC = \log_2(nm)$ & $SC = O(1)$

(Total # elements)

optimal for leetcode

```
bool search(vector<vector<int>> &matrix, int target)
{ → if(!matrix.size()) return false;
  int n = matrix.size();
  int m = matrix[0].size();
  int start = 0, end = (n*m) - 1;
  while (start <= end)
  {
    int mid = (start + end)/2;
    if (matrix[mid/m][mid%m] == target)
      return true;
    if (matrix[mid/m][mid%m] < target)
      start = mid+1;
    else end = mid-1;
  }
  return false;
}
```

it works for leetcode only & not gfg

Better

↳ previous code $TC = O(m+n)$ (for gfg)

Brute

↳ previous $TC = O(nm)$ (for gfg)

→ **Allocated books -**

Given an array of integers A of size N & an integer B. The college library has N books. The i^{th} book has A[i] number of pages.

You have to allocate books to B no. of students so that the maximum no. of pages allocated to a student is minimum.

conditions

1) A book will be allocated to exactly one student.
2) Each student has to be allocated atleast one book.
3) Allotment should be in contiguous order
   eg) A student can't be allocated book1 & book3 skipping book2.

Calculate & return the minimum possible no.

Return -1 if a valid assignment is not possible

Soln

arr[] → [12, 34, 67, 90] (0→Book0, 1→Book1, ...), size n = 4 & K = 2 (#students)

Your task is to allocate books to students s.t. max. no. of pages allocated to a student is minimum.

① says, if 2 students then Book0 can't be distributed to 2 students (Student 1 & Student 2) ✗

③ says 12, 67 (Student 1) 34, 90 (Student 2) ✗ (not contiguous)

now for, 12, 34, 67, 90 & K=2 allocation possible

[given array need not be sorted, our search space for BS will be #page in total] ↳ you will see

3 possible allocations:

- 12 | 34 67 90 → max no. of pages allocated = 191 (34+67+90)
- 12 34 | 67 90 → 157
- 12 34 67 | 90 → 113

∴ 113 is ans

approach

[12, 34, 67, 90] & n = 4 & K = 2

here, for BS will be done on #pages ↳ search space for BS

① [12 ..... 107 ..... 203] (total no of pages)

low = 12, high = 203

take min in arr

** [Better: if you take max in array]

mid = (12+203)/2 = 107

here, K = 2

*now, try allocating books to students such that no student gets more than 107 pages (↳ should be <=)

S1 = 12+34 = 46

S2 = 67 → can't add 90 as 67+90 > 107

S3 = 90

∴ #students = 3 > K (not possible)

go to right

∴ low = mid+1 = 108

Array
0 1 2 3
12, 34, 67, 90

② [108 155 203]
low = 0, mid = 1, high = 2

$mid = \frac{108+203}{2} = 155$

S1 = 12 + 34 + 67 = 113 (<155)
S2 = 90
∴ #students = 2 not > K
∴ possible

store res = mid = 155
here, we want to minimize it further
∴ go left
high = mid − 1 = 154

③ [108 131 154]
low = 0, mid = 1, high = 2

$mid = \frac{108+154}{2} = 131$

S1 = 12 + 34 + 67 = 113 (<131)
S2 = 90
↓
2 == K (not > K)
∴ possible, res = ~~155~~ 131
& go further left to minimize it
high = mid − 1 = 130

④ [108 119 130]
low = 0, mid = 1, high = 2

mid = 119

S1 = 12 + 34 + 67 = 113 (<119)
S2 = 90
↓
2 == K (not > K)
∴ possible, res = ~~131~~ 119
& go further left ie high = 119 − 1 = 118

⑤ [108 113 118]
low = 0, mid = 1, high = 2

mid = 113

S1 = 12 + 34 + 67 = 113 (<= 113) mid
S2 = 90
↳ 2 == K ~~i.e.~~
∴ possible res = ~~119~~ 113
& go further left ie high = 113 − 1 = 112

⑥ [108 110 112]
low = 0, mid = 1, high = 2

mid = 110

S1 = 12 + 34
S2 = ~~67~~
S3 = 90
→ 3 students > K
∴ go right ie low = (mid + 1) = 111

⑦ [111 111 112]
low ↑ mid ↑ high

mid = 111

S1 ⇒ 12+34
S2 ⇒ 67
S3 ⇒ 90
↳ 3 > K (=2) ∴ not possible & go right ∴ low = 111+1 = 112

⑧ [112 112 112]
low ↓ mid ↳ high

S1 = 12,34
S2 = 67
S3 = 90
↳ 3 > K ∴ not possible, ∴ go right low = 113

since low = 113, high = 112 → out of while

Note –

When it is possible to allocate books → we go left to further
ie ↓ high = mid − 1

When not possible, we try to ↑ no. of pages
∴ we go right → ie low = mid + 1

Code

```
if (K > n)
    return -1; //② condition
low = min in array
high = sum of array;
res = -1;
while (low <= high)
{
    mid = (low+high)/2;
    if (allocationIsPossible(mid) == true)
    {
        res = mid;
        high = mid-1; //go left
    }
    else //allocation not possible
    {
        low = mid+1;
    }
}
return res;
```

$TC = O(n \times \log n)$ (n: isPossible, log n: BS)

$SC = O(1)$

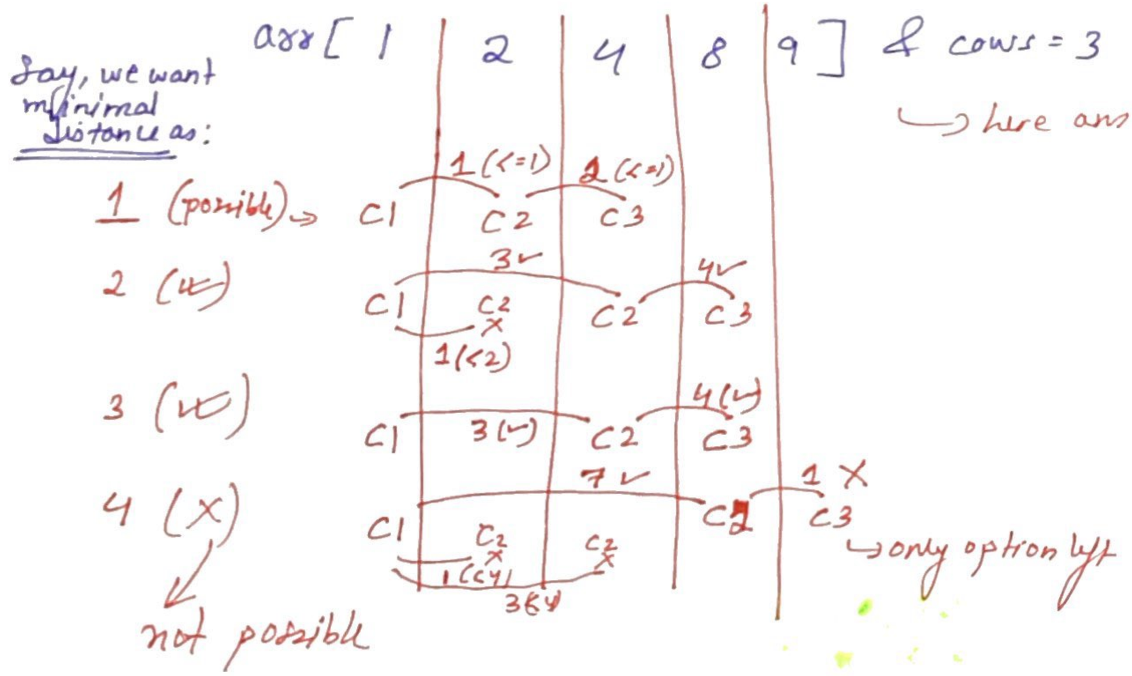
```
bool allocationIsPossible(int mid)
{
    int student = 1;   // initially
    int pages = 0;     // (no pages allocated till now)
    for (i=0; i<n; i++)
    {
        if (a[i] > mid) // #pages in a book > mid ∴ can't allocate
            return false;
        if (pages + arr[i] > mid)
        {
            ++student;
            pages = a[i];
        }
        else pages += a[i];
    }
    if (student > K)
        return false;
    return true;
}
```

→ Aggressive Cows –

n = 5 , arr[ ] → {1, 2, 4, 8, 9} & cows = 3

- n = 5 ↓ 5 stalls
- 1 → location of 1st stall
- array contains location of every stalls
- ↳ array can be unsorted

here, Your task is to place these cows = 3 in these 5 stalls such that we get the largest minimal distance b/w any two of cows.



∴ we can keep cows at distance of

1 ✓ 2 ✓ 3 ✓ | 4 X 5 X - - - -

largest min = 3 (Ans)

as 4 not possible, >4 will not be possible

Now What will be our search space for BS
↳ will be distances b/w cows

① [1 4 8]

- 1 → low (minimal ans possible)
- 4 → mid
- 8 → high ↳ 9 − 1 when C1 at 1 & C2 at 9 ∴ dist b/w them = 9 − 1 = 8 (WC)

$middle = \frac{1+8}{2} = 4$

now, can we place 3 cows in stalls with minimal dist = 4
not possible (we saw above)

∴ go left as we want to ↓ distance

∴ high = mid − 1 = 3

② [1 2 3]

low → 1, mid → 2, high → 3

mid = 2

↑ we can place cows with minimal distance of 2? (Yes) (see prev page)

res = 2

∴ ↑ dist as we want largest min

∴ go right

low = mid + 1

= 3

③ [3 3 3]

low → 3, mid → 3, high → 3

mid = 3

↑ we can place cows w/ minimal distance of 3? Yes (see prev page)

∴ res = 3

∴ ↑ distance i.e. go right

low = mid + 1

= 4

Now, low = 4, high = 3 } low > high → out of while

Ans = res = 3

V. Imp Note: If Questions says array is not sorted → sort it as we want to fig out distance b/w 2 cows

Code

sort the array → always ∴ it works

```
low = 1;
high = a[n-1] - a[0];   (max - min)
res = -1;
while (low <= high)
{
    mid = (low + high)/2;
    if (canplaceCows(mid))
    {
        res = mid;
        low = mid+1; //go right
    }
    else // can't place cows, ∴ ↓ dist
        high = mid+1;
}
return res;
```

```
bool canplaceCows(arr, n, cows, mid)   // n → size of array
{
    coord = a[0]; //first cow at 0th index
    cnt = 1; // #cows placed till now = 1
    for(i = 1; i<n; i++)   // check from next stall as a[0] occupied
    {
        if(a[i] - coord >= mid)   // ↳ want atleast mid
        {
            ++cnt; //count of cows ↑
            coord = a[i]; //for next cow
        }
        if(cnt == cows)
            return true;
    }
    return false;
}
```

we do when dist is satisfied if not, ignore

TC = O(N × log N) + O(N log N) if need to sort

loop BS

→ find the Nth root of a Number Using BS –

↳ tell nth root in terms how many decimal places you want the answer

eg $N=3$, $M=27$ → find $\sqrt[3]{27}$ & find upto d decimal places
$d=1$ $=3$ $n.\underbrace{\ldots\ldots}_{d}$

eg) $d=5$
$N=4, M=15$
$\therefore \sqrt[4]{15} = 1.\underbrace{96798}_{d}$

$\sqrt[N]{M}$

We will use **BS**
↳ we use BS when search space is montonically ↑ or ↓

for $N=3, M=27$

$1*1*1=1$
$2*2*2=8$
$3*3*3=27$
$4*4*4=64$ ↓ montonically ↑ as bigger the no.

Search space
↳ where can your answer lie?

$\sqrt[1]{M} = M$ ← if $N=1$

$[1 \quad 27]$
low high

$m = \frac{1+27}{2} = 14.0$ (of type double)

now, $14*14*14 > 27$ (M)
$\therefore$ go lyt ↳ $h = m$ (not $m-1$)

② $[1 \quad 14]$
$l \quad h$

$m = 7.5$
↳ $7.5*7.5*7.5 > 27$ (M)
$\therefore$ go lyt → $h = m$

③ $[1 \quad 7.5]$
$l \quad h$

$m = 4.25$
↳ $4.25*4.25*4.25 > 27$ (M)
$\therefore$ go to lyt → $h = m$

④ $[1 \quad 4.25]$
$l \quad h$

$m = 2.625$
↳ $2.625*2.625*2.625 < 27$ (M)
$\therefore$ go to right
$l = m$

⑤ [2.625 (l) 4.25 (h)]

m = 3.4375

↳ 3.4375 * 3.4375 * 3.4375 > 27

∴ go to lft ↳ h = m

⑥ [2.625 (l) 3.4375 (h)]

↙

how long will the search space will be shrinked.

till $(h-l) > 1e^{-(d+1)}$

↙ if asked d = 5 ↳ 5 decimal place

& at end of day $(h-l) > 1e^{-6}$ ∴ $1e^{-6}$

ans = ...

Code

```
double multiply(double num, int n) // for nth root
{
    double ans = 1.0;
    for(int i=1; i<=n; i++)
    {
        ans = ans * num;
    }
    return ans;
}
double getNthroot(int n, int m)   → d not given (Lets take d = 5)
{
    double low = 1.0, high = m, eps = 1e-6;
    while ((high - low) > eps)
    {
        double mid = (low + high)/2.0;
        if(multiply(mid, n) < m)
        {
            low = mid;
        }
        else
            high = mid;
    } return low; (or high)
```

TC = say d = 1, then search space

↳ finding the ans for 1 decimal place

[1.0, 1.1, ..., 1.9, 2.0, 2.1, ..., 2.9, ..., 27]

(1.0, 1.1, ..., 1.9 : 10; 2.0, 2.1, ..., 2.9 : 10)

∴ for this TC = N * $\log_2(M \times 10^d)$ (d → ↳ BS, N → loop)

∴ for d decimal places

= * $(M \times 10^d)$ = $O(N \log_2 M)$

→ Median of Row wise Sorted Matrix (Nested BS)

given $\begin{bmatrix} 1 & 3 & 6 \\ 2 & 6 & 9 \\ 3 & 6 & 9 \end{bmatrix}$ & $N=3$, $M=3$ → has all row sorted

Note here, N×M should be odd

↳ here, if we write all integers in sorted order -

1 2 3 3 **6** 6 6 9 9

middle element ↳ median = 6

here, constraint is all array elements b/w the range $[1-10^9]$

Brute

put all elements of matrix in vector → sort vector → find middle index = median

TC = O(N×M) ; SC = O(NM) ; O(NM log(NM)) ; O(1)

$TC = O(N \times M) + O(NM \log NM)$

$SC = O(NM)$ ↳ vector

Optimal

↳ using BS

here, constraint is $[1$ to $10^9]$ for A[i]

∴ median will lie in $[1 \underbrace{\text{----}}_{\text{our search space}} 10^9]$

here, $n=3$, $m=3$ → Total nos = 3×3 = 9

1, 2, 3, 3, 6, 6, 6, 9, 9 Now, Nos <=1 → 1
<=2 → 2
<=3 → 4
<=4 → 4
<=6 → 7

↑ fashion (∴ BS can be used)

$$\begin{bmatrix} 1 & 3 & 6 \\ 2 & 6 & 9 \\ 3 & 6 & 9 \end{bmatrix}_{3\times3}$$

$n = 3,\ m = 3$

$\therefore \frac{3\times3}{2} = 4$

① Say, [1 ... 15] (low = 1, high = 15; should be 18; taken to give eg)

$m = 8$

↳ find how many are $<= 8$

↙ go to matrix row wise & use BS call(8) for each row

gives = 7 (>4) ↙

means there are 7 nos. including 8, $<= 8$

[1 ... 8] → 7

$\therefore$ go left

$\therefore L = m - 1 = 7$

② [1 ... 7] (L, h)

$m = 4$

↳ find how many are $<= 4$

we get → 4 nos $<= 4$ (including 4)

[1 ... 4] → 4

↳ $\therefore$ this can't be median as for median

<= | median | >=

4 elements, 4 elements

$\frac{3\times3}{2}$

$\therefore$ go to right (as $4 < 4$ ✗)

$h = m + 1 = 5$

③ [5 ... 7] (L, h)

$m = 6$

↳ find how many nos. $<= 6$

↳ we get = 7 (>4) ↙

[1 ... 6] → 7

$\therefore$ go to ~~right~~ left.

$\therefore h = m - 1 = 5$

④ [5 ... 5] (L, h)

$m = 5$

↳ find how many nos. $<= 5$ ↳ we get 4 (<4) ✗

[1 ... 4] → 3

$\therefore$ go to right

$L = m + 1 = 6$

now, $L = 6$, $h = 5$ ↳ out of while

$\therefore$ Ans = $L = 6$

- Now to find how many nos <= mid
  ↓
  we can do is find ceil(mid) using BS

Conclusion

search space [1 to 10^9)

In bs → find mid → find how many <= mid (say x)

- x <= (n*m)/2 → go to right, l = mid+1
- x > → go to left, h = mid-1

⇓

done using BS ceil(mid)

Code

```
int BSceil(vector<int>& row, int mid)
{
    int l=0, h=row.size()-1;
    while(l<=h)
    {
        int mid=(l+h)/2;
        if(row[mid]<=mid)
            l=mid+1;
        else
            h=mid-1;
    }
    return l;
}

int findMedian(vector<vector<int>>& A)
{
    int low=1;
    int high=1e9;
    int n=A.size(), m=A[0].size();
    while(low<=high)
    {
        int mid=(low+high)/2;
        int cnt=0;
        for(int i=0;i<n;i++) // for each row, find #elements <= mid
            cnt+=BSceil(A[i], mid);
        if(cnt<=(n*m)/2)
            low=mid+1;
        else high=mid-1;
    }
    return low;
}
```

(l → contain ceil, h → " floor)

how this ceil gives ceil(x) & not ceil(x)<x

return index (which stores how many nos <= x)

TC - here, $10^9 \approx 2^{32}$ → highest integer we can have

$\therefore \log_2(2^{32}) \times N \times \log_2 M$

(for every row we do BS)

$TC = O(32 \times N \times \log_2 M)$

& $SC = O(1)$

→ Median of 2 sorted Arrays of different sizes -

arr1[]: 1, 3, 4, 7, 10, 12 , $n_1 = 6$

arr2[]: 2, 3, 6, 15 , $n_2 = 4$

Better

Merge sort wala technique using 2 pointers

& if n_1+n_2 = even → median is at $\frac{total}{2}$ & $\frac{total}{2}+1$ (here, $\frac{10}{2}=5$ → 6 & median is avg of this)

n_1+n_2 = odd → " " at $\frac{n_1+n_2}{2}$

& use a counter.

$O(n_1+n_2)$

→ we get

1, 2, 3, 3, 4, 6, 7, 10, 12, 15

↓ while using 2 pointers. ↑ count & whenever count = 5 (we get 4) & 6 (we get 6) we get our median at this count

& no need to go until 7

$\therefore$ median = $\frac{4+6}{2} = 5$

$\therefore TC = O(n_1+n_2)$

$SC = O(1)$

Optimal

arr1[] → {1, 3, 4, 7, 10, 12} , $n_1 = 6$

arr2[] → {2, 3, 6, 5} , $n_2 = 4$

Here, $n_1+n_2 = 6+4 = 10$

$\therefore$ for median, left part should have 5 elements & right " " " 5 "

sorted: 1, 2, 3, 3, 4 | 6, 7, 10, 12, 15

→ need partition of 5 & 5

Ways

5 elements for left part | 5 elements for right part

① from arr1: 1 3 4 7 | 10 12

from arr2: 2 | 3 6 15

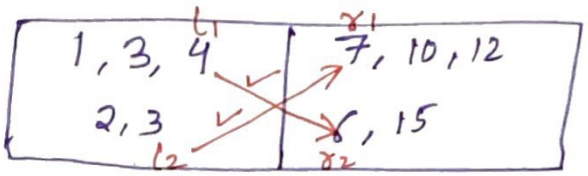
L_1 = 7, R_2 = 3 (not valid)

$L_1 <= R_2$

$7 <= 3$ X

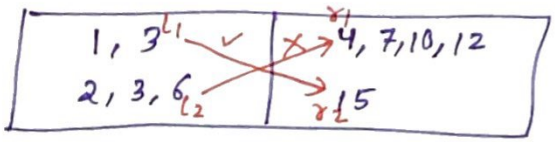
PTO

(2)



(valid)

(3)



(not valid)

Now, to check ①, ②, ③ are valid or not?

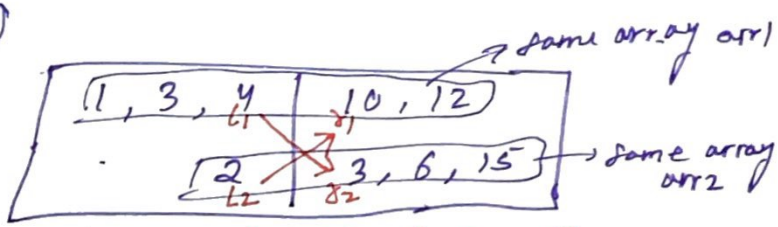
ie left part should have & right part should have

sorted 1, 2, 3, 3, 4, | 6, 7, 10, 12, 15

all elements over here <= than this

∴ to check this we can use l_1, l_2, r_1, r_2

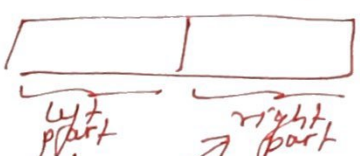
like in ①



check $l_2 <= r_1$ & $l_1 <= r_2$

why?

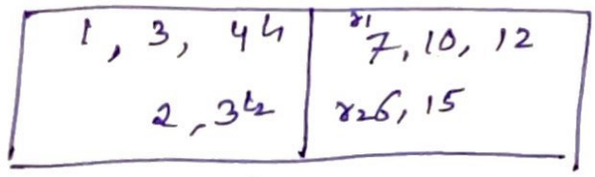
from here



all elements here are <= than right part

∴ this ensure all the elements in left half <= all elements in right half

now, in (2)



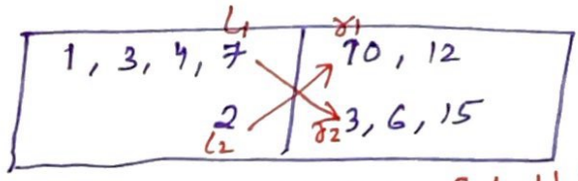
if length is even, (→ n_1, n_2)

$$median = \frac{\max(L_1, L_2) + \min(R_1, R_2)}{2}$$

How to use BS?

↳ first see → How we know which is valid one?

(1)



here, $7 <= 3$ (should be)

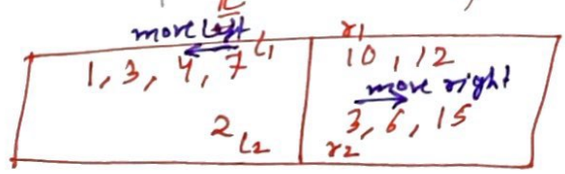
↘ X

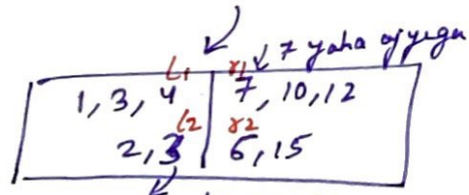
in order to make 7 lesser than 3

↓↓ 7 <= 3 ↑↑

reduce this, increase this

to form correct partition



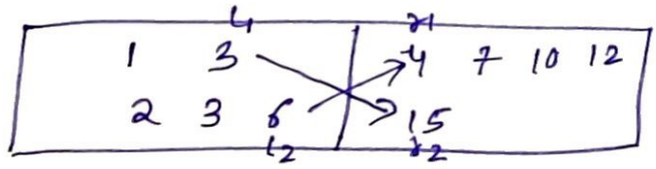


3 yaha ayega

we reduce 7 to 4 (now L_1 → now R_1)

& increase 3 to 6

(3)

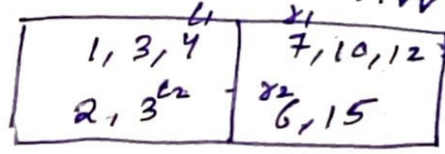


$L_1 <= R_2$ ✓

$L_2 <= R_1$

i.e $6 <= 4$ X

$\therefore$ ↓↓ $6 <= 4$ ↑↑



Dry run

arr1 → {7, 12, 14, 15} & $n_1 = 4$ → smaller = 4

arr2 → {1, 2, 3, 4, 9, 11} & $n_2 = 6$

(0 1 2 3 for arr1; 0 1 2 3 4 5 for arr2)

we will work wrt smaller array.

① ∴ [0 ... 4]

- 0 → low → we can pick min of 0 element from arr2
- 4 → high → can pick max of 4 elements from arr1

$m = 2$

say, we call $m = cut1 = 2$ → # elements we pick from arr1

$cut2 = 5 - 2 = 3$ → 3 elements from arr2

for this part

left part 5 elements | right part 5 elements → $\frac{4+6}{2} = 5$

cut1

0	1	2	3
7	12	14	15

0	1	2	3	4	5
1	2	3	4	9	11

cut2

∴ $l_1 = 12$ $r_1 = 14$

$l_2 = 3$ $r_2 = 4$

[what matters are l_1, l_2, r_1 & r_2]

now, $l_1 <= r_2$

$12 <= 4$ ✗

↳ not a valid partition

∴ ~~if~~ $12 <= 4$ ↑↑

move left to reduce 12

high = mid − 1

= 2 − 1

= 1

② [0 ... 1]

- 0 → low
- 1 → high

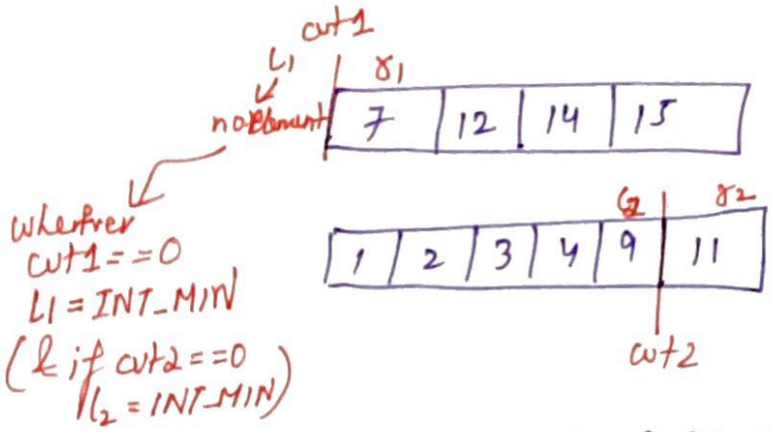
$m = 0$

cut1 = m = 0 → taking no elements from arr1

cut2 = 5 − 0 = 5 → taking 5 elements from arr2

for this

5 | 5



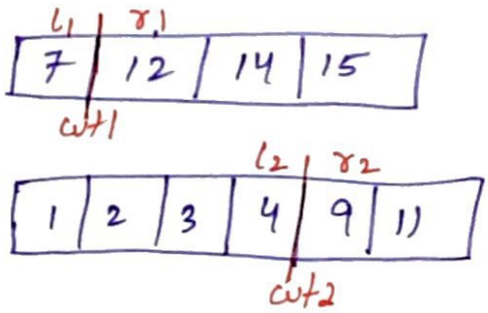
whether
cut1 == 0
l_1 = INT_MIN
(& if cut2 == 0
l_2 = INT_MIN)

- $l_1 <= r_2$
  INT_MIN <= 11 ✓
- $l_2 <= r_1$
  9 <= 7 ✗
  ↳ go to right
  $l = mid + 1$
  $= 0 + 1$
  $= 1$

(3) [1 1]
low, high

$m = \frac{1+1}{2} = 1$

cut1 = 1 → pick 1 element from arr1
cut2 = 5 − 1 = 4 → pick 4 elements from arr2



- $l_1 <= r_2$
  7 <= 9 ✓
- $l_2 <= r_1$
  4 <= 12
  ∴ this is a valid partition
  Stop & return

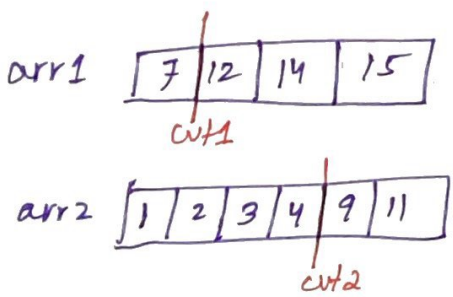
Since $n_1 + n_2 = 10$ (even)

$median = \frac{\max(l_1, l_2) + \min(r_1, r_2)}{2}$

$= \frac{7+9}{2} = \frac{8}{2}$

Ans

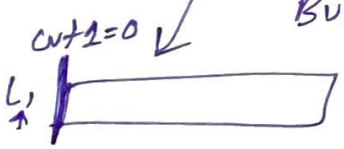
Note — Since we require l_1, l_2, r_1 & r_2



$l_1 = arr1[cut1-1]$ $\quad r_1 = arr1[cut1]$

$l_2 = arr2[cut2-1]$ $\quad r_2 = arr2[cut2]$

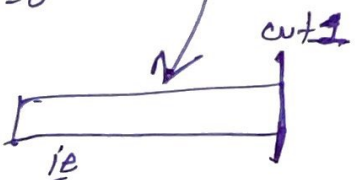
But make sure



↳ then make

$l_1 = INT\_MIN$ if $cut1 = 0$

Similarly, $l_2 = INT\_MIN$ if $cut2 = 0$



if $cut1 = n1$, $r_1 = INT\_MAX$

Similarly, if $cut2 = n_2$, $r_2 = INT\_MAX$

If length = odd

Say, arr1 | 7 | 12 | 14 | 15 | 16 | $n_1 = 5$

arr2 | 1 | 2 | 3 | 4 | 9 | 11 | $n_2 = 6$

$\therefore n_1 + n_2 = 5 + 6 = 11$ (odd)

if sorted: 1, 2, 3, 4, 7, 9 | 11, 12, 14, 15, 16

(6) (5)

once we know how many elements on lyt half, we will figure out the partition.

$\frac{n_1+n_2+1}{2}$ ← works both for odd & even

Since, it is odd length.

↳ median is the last element of lyt half

9 | 11

5 5

(PTO)

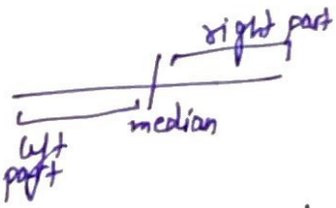
∴ median will be = max(l_1, l_2)

Code

```
double median(vector<int>& nums1, vector<int>& nums2)
{
    if(nums2.size() < nums1.size())
        return median(nums2, nums1); // if nums2 is smaller array, do this

    int n1 = nums1.size(); //∵ nums1 is smaller array
    int n2 = nums2.size();
    int low = 0, high = n1;
    while (low <= high)
    {
        int cut1 = (low + high)/2;
        int cut2 = ((n1+n2+1)/2) - cut1;
                                  // works both for odd & even
        int l1 = (cut1 == 0) ? INT_MIN : nums1[cut1-1];
        int l2 = (cut2 == 0) ? INT_MIN : nums2[cut2-1];
        int r1 = (cut1 == n1) ? INT_MAX : nums1[cut1];
        int r2 = (cut2 == n2) ? INT_MAX : nums2[cut2];

        if (l1 <= r2 && l2 <= r1) // partition is valid ∴ return median
        {
            if((n1+n2)%2 == 0) //even length
                return (max(l1,l2) + min(r1,r2))/2.0;
            else //odd length
                return max(l1,l2);
        }
        else if (l1 > r2) //move left to reduce l1
            high = cut1 - 1;
        else
            low = cut1 + 1;
    }
    return 0.0; //if nums1 & nums2 not sorted
}
```



Everything is done wrt left part ie cut1

if left part sahi hai, automatically right part sahi hoga [as sorted]

TC = we are doing BS on minimal array
ie $O(\log_2(\min(n_1, n_2))) = TC$

SC = O(1)

→ Kth element of two sorted Arrays

Given 2 sorted arrays arr1 & arr2 of size M & N respectively & an element K. find the element that would be at the Kth position of the final sorted array.

Soln

arr1 → {2, 3, 6, 7, 9} & K = 5
arr2 → {1, 4, 8, 10}

↳ o/p = 6

↙

How when final array is sorted -

1, 2, 3, 4, 6, 7, 8, 9, 10
K=1 K=2 ... K=5
∴ Ans = 5

Better

Merge sort wala concept lgao using 2 pointers on 2 sorted array & use count & when count == K, we get the Kth element

∴ TC = O(n1+n2)
SC = O(1)

Optimal

↳ same as previous Question

↙

we will work for left part only as when we get left part of final sorted array we automatically get the right part of final sorted array!

we will use same concept as l_1, l_2, r_1, r_2

& when $l_1 <= r_2$ & $l_2 <= r_1$ } we get the ans

& $\max(l_1, l_2)$ is the ans

edge cases

arr1 | 7 | 12 | 14 | 15 |

arr2 | 1 | 2 | 3 | 4 | 9 | 11 |

when K = 3

low = 0

high = 3 → max element we can pick from arr1 when K=3

when K = 7

low = ~~0~~ 1 → not zero as in 2nd array we have 6 element ∴ if we take 0 element from arr1, we can't have 7 element in left part of sorted array (final)

high = 4 → max element we can pick from arr1 even K=7

∴ low = 1

1 from arr1, 6 from arr2 ∴ total 7

low = 1 how?

if in min you have to take 1 guy from arr1

here, n=4 & m=6 & K=7

final, K<7, so we can't take 0 elements as the lowest no. of elements picked from arr1. It should be $\max(0, K-m) = \max(0, 7-6) = 1$ (=low)

& high is obvious $\min(K, n)$ i.e. $\min(7,4) = 4$ elements can be taken at most from arr1

left arr1

$low = \max(0, K-m)$ → size of 2nd array

$= \max(0, 7-6)$

$= 1$

& $high = \min(K, n)$

$= \min(7, 4) = 4$

Code

```
int KthElement(int arr1[], int arr2[], int n, int m, int K)
{
    if(n > m) // so we will always do BS on smaller array
        return KthElement(arr2, arr1, m, n, K);

    int low = max(0, K-m);
    int high = min(K, n);
    while(low <= high)
    {
        int cut1 = (low+high)/2;
        int cut2 = K-cut1;
        int l1 = (cut1 == 0) ? INT_MIN : arr1[cut1-1];
        int l2 = (cut2 == 0) ? INT_MIN : arr2[cut2-1];
        int r1 = (cut1 == n) ? INT_MAX : arr1[cut1];
        int r2 = (cut2 == m) ? INT_MAX : arr2[cut2];

        if(l1 <= r2 && l2 <= r1)
            return max(l1, l2);
        else if(l1 > r2)
            high = cut1-1;
        else
            low = cut1+1;
    }
    return 1; // when array is not sorted
}
```

edge cases

TC = O(log(min(n, m)))

SC = O(1)

→ find the smallest divisor given a threshold -

Given an array of integers nums & an integer threshold, we will choose a +ve integer divisor, divide all the array by it & sum the division's result. find the smallest divisor such that the result mentioned above is less than or equal to threshold.

Each result of the division is rounded to the nearest integer greater than or equal to that element (eg $7/3 = 3$ → ceil value, $10/2 = 5$)

It is guaranteed that there will be an answer.

soln

i/p: [1, 2, 5, 9] & threshold = 6

if divisor = 1 ∴ $\frac{1}{1} + \frac{2}{1} + \frac{5}{1} + \frac{9}{1} = 17 (<=6) \times$

" = 2 ∴ $\frac{1}{2} + \frac{2}{2} + \frac{5}{2} + \frac{9}{2}$

$= 1 + 1 + 3 + 5 = 10 (<=6) \times$

(ceil value)

" = 3 ∴ $\frac{1}{3} + \frac{2}{3} + \frac{5}{3} + \frac{9}{3}$

$= 1 + 1 + 2 + 3 = 7 (<=6) \times$

" = 4 ∴ $\frac{1}{4} + \frac{2}{4} + \frac{5}{4} + \frac{9}{4}$

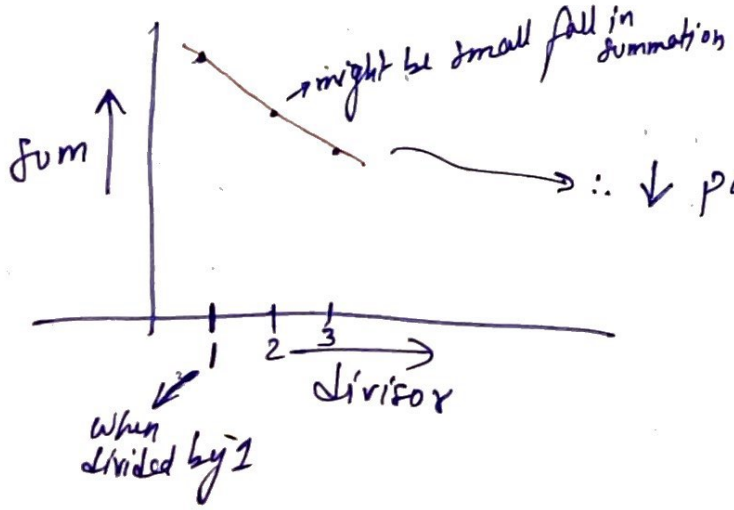
$= 1 + 1 + 2 + 3 = 7 (<=6) \times$

" = 5 ∴ $\frac{1}{5} + \frac{2}{5} + \frac{5}{5} + \frac{9}{5}$

$= 1 + 1 + 1 + 2 = 5 (<=6)$ ✓

∴ Ans = 5

find smallest divisor



∴ ↓ pattern when we ↑ divisor, sum ↓

∴ ↑↑ divisor, sum ↓↓

for <u>divisor</u>

[1 ← minimal divisor

if div by 1, we get $sum = \sum_{i=1}^{n} arr[i]$ always

max(arr[i])] ← we get lowest sum when we divide by max element in array.

eg {1, 2, 5, 9} & max is 9

$\therefore \frac{1}{9} + \frac{2}{9} + \frac{5}{9} + \frac{9}{9}$

$= 1 + 1 + 1 + 1$

$= 4$ ↪ ≡ size of array (n)

& beyond 9 (>9) whatever we divide by, sum = 4 only.

∴ when divisor = 1 ⟶ max sum = $\sum arr[i]$

" = max(arr[i]) ⟶ min sum = n

here, search space for BS -

given - [1 2 5 9] & threshold = 6

① [1 (low) 9] (high)

m = 5

↪ now $\frac{1}{5} + \frac{2}{5} + \frac{5}{5} + \frac{9}{5}$

$= 1 + 1 + 1 + 2 = 5 \ (<=6)$ ✓

∴ ans = 6 (abhi ki liye)

now, since we want min divisor

go <u>left</u>

high = m - 1 = 4

② [1 (low) 4 (high)]

m = 2

↪ $\frac{1}{2} + \frac{2}{2} + \frac{5}{2} + \frac{9}{2} \Rightarrow 1 + 1 + 3 + 5 = 10 \ (<=6)$ ✗

∴ go right

low = m + 1 = 3

⋮

go like this

and at end, return <u>ans</u>

```
int findSumAfterDiv(int arr[], int n, int div)
{
    int sum=0;
    for(int i=0; i<n; i++)
    {
        sum += (arr[i]/div);
        if(arr[i]%div != 0) // for ceil
            sum += 1;
    }
    return sum;
}

int findMinDiv(int arr[], int n, int thres)
{
    int low=1, high = *max_element(arr, arr+n);
                      ↳ give max element in array
    int ans = high; //  → they have said ans has a
                        ∴ if div by max element, result always <= threshold
    while(low <= high)
    {
        int mid = (low+high)/2;
        if(findSumAfterDiv(arr, n, mid) <= thres)
        {
            ans = mid;
            high = mid-1; //look for smaller ans
        }
        else
            low = mid+1;
    }
    return ans;
}
```

(or 0)

$TC = O(N * \log(\text{max element in array}))$

$SC = O(1)$

1→ Split Array Largest Sum-

Given an array nums which consists of non-negative integers & an integer m, you can split the array into m non-empty contiguous subarrays.

Minimize the largest sum among these m subarrays.

Soln

nums → {7 2 5 10 8} → not sorted & m = 2

possible ways - (want 2 non-empty subarrays)

	sum1, sum2	largest
[7][2 5 10 8]	7, 25	25
[7 2][5 10 8]	9, 23	23
[7 2 5][10 8]	14, 18	18
[7 2 5 10][8]	24, 8	24

min (max) we need to do

is 18 (Ans)

split is [7 2 5][10 8]

∴ for BS, search space is -

[10 ... 32]

low → will be max element in array

high → sum of array

mid = 21

↳ ∴ see if (m) 2 subarrays should have sum <= 21

∴ [7, 2, 5] [10, 18] ← ans = 21

& go left ↳ as min(

∴ high = mid - 1 = 20

as we are asking min (max sum)

as even if we asked to find m = n subarrays

[7] [2] [5] [10] [8]

∴ 10 is the most minimal we can get

now

[10 (low) ... 20 (high)]

mid = 15 ⟶ see 2 subarrays should have sum <= 15

∴ [7, 2, 5]
[10]
[8]

get 3 subarrays
but we want 2 subarrays
∴ go to right as we want 2 subarrays
∴ low = mid + 1 = 16

[16 (low) ... 20 (high)]

mid = 18 ⟶ see 2 subarrays should have sum <= 18

∴ [7, 2, 5] } it possible
[10, 8] }
∴ ans = 18
& go left to min(max, mid)
high = 18 - 1 = 17

⋮

aise krte raho
& when low > high
out of while & return ans

Code

```
int minMaxSumSubarray(int arr[], int n, int m)
{
    int low = *max_element(arr, arr+n);
    int high = 0;
    for(int i=0; i<n; i++)
        high = high + arr[i];
    int ans = high; // in WC
    while(low <= high)
    {
        int mid = (low+high)/2;
        if(numOfSubarrays(arr, n, mid) == false)
            low = mid+1;
        else
        {
            ans = mid;
            high = mid-1;
        }
    }
    return ans;
}
```

```
bool numOfSubarrays(int arr[], int n, int mid)
{
    int cnt = 1; // first subarray starting at 0
    int sum = 0;
    for(int i=0; i<n; i++)
    {
        if(arr[i] > mid) // arr[i] can't be part of any subarray
            return false;
        if(sum + arr[i] > mid)
        {
            cnt++; // create 1 more subarray
            sum = arr[i];
        }
        else sum += arr[i];
    }
    return cnt <= m;
}
```

$$TC = O(N * \log(\underbrace{\text{sum of array} - \max \text{element}}_{\text{BS space}})) \text{ \& } SC = O(1)$$

loop

BS

Binary Search (Striver)

1) find Single element in Sorted array -

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

answer

M1) XOR concept → O(n)

M2) for (i = 0 to n-1)

```
{ if (i == 0)
    { if (a[i] != a[i+1])
        return a[0];
    }
  else if (i == n-1)
    { if (a[i] != a[i-1])
        return a[i];
    }
  else
    { if (a[i] != a[i+1] && a[i] != a[i-1])
        return a[i];
    }
}
```

→ if (n == 1) return a[0]

TC = O(n)

Optimized → BS

0	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	3	4	5	5	6	6

(1,1) (2,2) (3,3) (5,5) (6,6)

even index, odd index

e o e o e o e o

∴ (even, odd) → we are on left half → our element is in right half

(odd, even) → we are on right half → our element is in left part

```
int singleele(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 l=0, h=n-1;
    while(l<=h)
    {
        int m = (l+h)/2;
        if(nums[m]!=nums[m-1] && nums[m]!=nums[m+1])
            return nums[m];
        else if((m%2==0 && nums[m]==nums[m+1])
                || (m%2==1 && nums[m]==nums[m-1]))
            l = m+1;
        else
            h = m-1;
    }
    return -1; // dummy
}
```

//edge cases.

(e, o)
m ↑ check

(e, o)
check ↑ m

2) Square root of a no.- (return int value = sqrt(n))

```
int l=1, h=n,
while (l<=h)
{
  long long mid = l+(h-l)/2;
  if(mid*mid == n)
    return mid;
  else if(mid*mid < n)
    l=mid+1;
  else
    h=mid-1;
}
return h;
```

3) Kth missing positive number-

Given an array of +ve integers sorted in a strictly ↑ order & an integer K.

Return the Kth +ve integer missing from this array.

eg) arr[] = {2,3,4,7,11}, & K=5

o/p = 9 → missing +ve integers are [1,5,6,8,9,10,12,....].

5th +ve integer = 9

M1 : O(n) TC

```
int f(vector<int>& arr, int K)
{
  int ans = K;
  for(auto it : arr)
  {
    if(it <= ans)
      ++ans;
    else
      break;
  }
  return ans;
}
```

M2 : TC = O(log n)

```
int l=0, h=arr.size()-1;
while (l<=h)
{
  int m = (l+h)/2;
  if(arr[m]-(m+1) < K)
    l=m+1;
  else
    h=m-1;
}
return l+K;
```

arr[m]-(m+1) → calculating the # of missing values before mid index value

so, if # missing values < K then check on right side

or

l+K+1

4) finding peak element in 2D array -

A peak element in a 2D grid is an element that is strictly greater than all of it's adjacent neighbours to the left, right, top & bottom.

Given a 0-based indexing m×n matrix, where no 2 adjacent elements/cells are equal, find any peak element mat[i][j] & return the length 2 array [i, j].

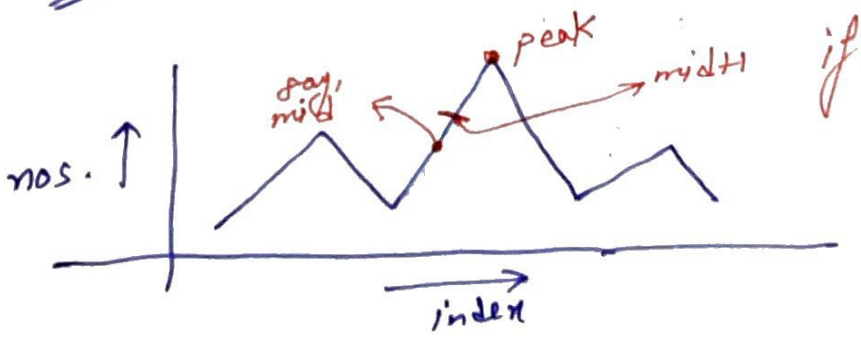
Assume that the entire matrix is surrounded by an outer perimeter with value -1 in each cell.

Soln

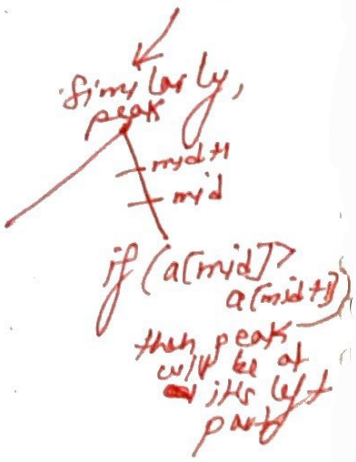
M1: go through every element & check ←, ↑, →, ↓ TC = O(m*n*4)

M2: find largest element in grid TC = O(m*n)
↳ will be peak element

M3: BS space



for 1D arrays,
if (a[mid] < a[mid+1])
↳ chances peak is at right side



now,

PTO

we have 6 colms

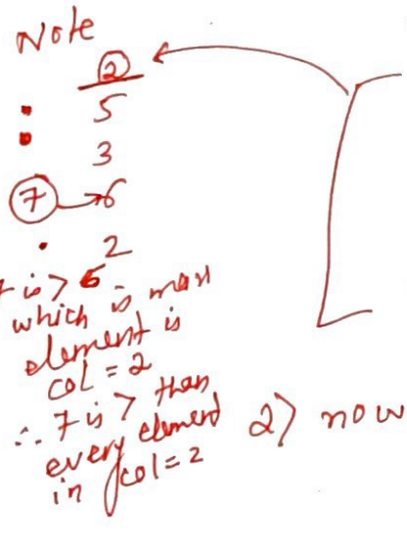
low ↑ (0) high → (5)

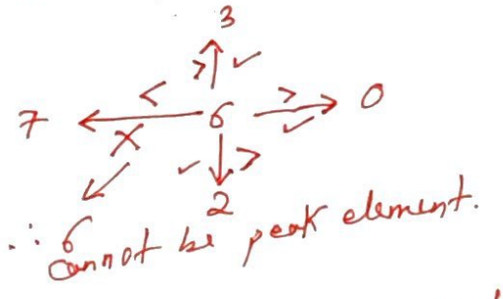
	0	1	2	3	4	5
0	4	2	5	1	4	5
1	2	9	3	2	3	2
2	1	7	6	0	1	3
3	3	6	2	3	7	2

1) low = 0 high = 5

$mid = \frac{0+5}{2} = 2$

max element in col=2 is 6 → It has high probab to be peak element.





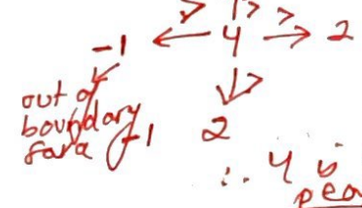
2) now, where to move → left or right ?

since 6 < 7 ⇒ left of 6 ∴ go to left side

∴ low = 0 high = 1

$mid = \frac{0+1}{2} = 0$

↳ in colm = 0, max = 4



∴ 4 is the peak element (Ans)

```
int findmaxIndex(vector<vector<int>> &mat, int n, int m, int col)   // return index of row
{
    int maxi = -1;
    int ind = -1;
    for (int i=0; i<n; i++)
    {
        if(mat[i][col] > maxi)
        {
            maxi = mat[i][col];
            ind = i;
        }
    }
    return ind;
}
vector<int> findpeakgrid(vector<vector<int>> &mat)
{
    int n = mat.size();
    int m = mat[0].size();
    int low = 0, high = m-1;
    while(low <= high)
    {
        int mid = (low+high)/2;
        int maxRowindex = findmaxIndex(mat, n, m, mid);
        int left = mid-1 >= 0 ? mat[maxRowindex][mid-1] : -1;
        int right = mid+1 < m ? mat[maxRowindex][mid+1] : -1;
        if(mat[maxRowindex][mid] > left && mat[maxRowindex][mid] > right)
            return {maxRowindex, mid};
        else if(mat[maxRowindex][mid] < left)
            high = mid-1;
        else
            low = mid+1;
    }
    return {-1,-1}; // dummy;
}
```

we are not checking for up & down since we found the max element in the col'mn, it will be greater than up & down.

$TC = O(n \log m)$ & $SC = O(1)$

$\log m$ → BS on col'm

n → to find max