→ Implement Lower Bound

↳ smallest index such that arr[index] ≥ target

0	1	2	3	4	5
3	5	8	15	19	19

target = 19

arr[0] ≥ 19 ✗
arr[1] ≥ 19 ✗
arr[2] ≥ 19 ✗
arr[3] ≥ 19 ✗
arr[4] ≥ 19 ✓
↳ return 4;

(5 > 1)
ans = 4

① s=0, e=9, ans = arr.length (10)
m = (0+9)/2 = 4
is arr[mid] ≥ target? ✓
↳ ans = mid
end = m-1

② s=0, e=3
m = (0+3)/2 = 1
(2 > 1) is arr[mid] ≥ target? ✓
↳ ans = mid
end = m-1
ans = 1

③ s=0, e=0
(1 ≥ 1) is arr[mid] ≥ target? ✓
↳ ans = mid
end = m-1
ans = 0 → lower bound

0	1	2	3	4	5	6	7	8	9
1	2	3	3	5	8	8	10	10	11

s ↑ target₁ = 1 , target₂ = 9 ↑ e

ans = 4 → we have to find smallest index, so we will keep on checking

Similarly we will find the answer for this target.

① s=0, e=9, ans = 10
m = (0+9)/2 = 4
arr[mid] ≥ target₂
(5 ≥ 9) ✗
start = mid+1

② s=5, e=9
m = (5+9)/2 = 7
arr[mid] ≥ target₂
ans = mid
end = mid-1

③ s=5, e=6
m = (5+6)/2 = 5
arr[mid] ≥ target₂
start = mid+1

④ s=6, e=6
m = (6+6)/2 = 6
arr[mid] ≥ target₂
(8 > 9) ✗
start = mid+1

ans = 7
↳ start > end
↳ lower bound

ans = 10
(10 > 9) ✓
ans = 7

(8 > 9) ✗
ans = 7

Part1.java

```java
public static int lowerBound(int[] arr,int target){
    int start = 0;
    int end = arr.length - 1;
    int ans = arr.length;

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

        if (arr[mid] >= target) {
            ans = mid;
            end = mid - 1;
        }
        else start = mid + 1;
    }

    return ans;
}
```

TC: O(log n)
SC: O(1)

→ Implement upper bound

↳ smallest index such that arr[index] > target

Part1.java

```java
public static int upperBound(int[] arr,int target){
    int start = 0;
    int end = arr.length - 1;
    int ans = arr.length;

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

        if (arr[mid] > target) {
            ans = mid;
            end = mid - 1;
        }
        else start = mid + 1;
    }

    return ans;
}
```

↳ the only difference between lower & upper bound is

if (arr[mid] > target)

TC : $\log_2(n)$

SC : O(1)

→ Search Insert position

↳ return the index if the target is found, else return the index where it would be inserted in that order.

① arr = | 1 | 3 | 5 | 6 | (0 1 2 3) target = 5

↳ return 2

② arr = | 1 | 3 | 5 | 6 | (0 1 2 3) target = 2

① s=0, e=3, m = (0+3)/2 = 1, ans = 4 (length)

is arr[mid] ≥ target

ans = mid

end = m-1

so, ans = 1

② s=0, e=0

is arr[mid] ≥ target ×

↳ Same question as lower bound

☆ Apply lower bound on this problem

→ Implement Floor & Ceil

largest no. in array ≤ target

smallest no. in array >= target

↓

same as lower bound

↳ return arr[lowerbound()]

```java
Part1.java
public static int floor(int[] arr,int target){
    int ans = -1;
    int start = 0;
    int end = arr.length - 1;

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

        if (arr[mid] <= target){
            ans = arr[mid];
            start = mid + 1;
        }
        else end = mid - 1;

    }
    return ans;
}
```

→ moving towards right since we are searching for largest number

TC: $O(\log_2(n))$

SC: $O(1)$

→ First & last occurrence of the target

	0	1	2	3	4	5	6
arr =	2	4	6	8	8	11	13

target = 8

① Brute force - O(n)

first = -1, last = -1

```
for(int i=0; i < arr.length; i++){
    if(arr[i] == target){
        if (first == -1) first = i;
        last = i;
```

→ keep on updating i for more last possible occurrence.

3

3

② Upper & lower bound

```java
Part1.java
public static int[] firstAndLast(int[] arr,int target){  no usages  new *
    int lb = lowerBound(arr, target);

    if (lb == arr.length || target != arr[lb]) return new int[]{-1,-1};

    int ub = upperBound(arr, target);

    return new int[]{lb , ub - 1};
}
```

TC: $2\log_2(n)$

SC: $O(1)$

2	4	6	8	8	8	11	13

$target_1 = 10$ $target_2 = 14$

lower bound (arr, 10) = 11

10 isn't in array, so we must check if target == arr[lb]

for target = 14, the lb = length of the array

☆ if any of the two cases occur we must return {-1, -1}

(3) Plain binary search

Part1.java

```java
public static int firstOccurrence(int[] arr, int target){
    int start = 0;
    int end = arr.length - 1;
    int first = -1;

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

        if (arr[mid] == target) {
            first = mid;
            end = mid - 1;
        }
        else if (arr[mid] < target) start = mid + 1;
        else end = mid - 1;
    }
    return first;
}
```

whenever we find first occurrence, we repeatedly check in left half of sorted array.

Part1.java

```java
public static int lastOccurrence(int[] arr, int target){
    int start = 0;
    int end = arr.length - 1;
    int last = -1;

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

        if (arr[mid] == target) {
            last = mid;
            start = mid + 1;
        }
        else if (arr[mid] < target) start = mid + 1;
        else end = mid - 1;
    }
    return last;
}
```

Similarly, whenever we find last occurrence, we search in right half of sorted array.

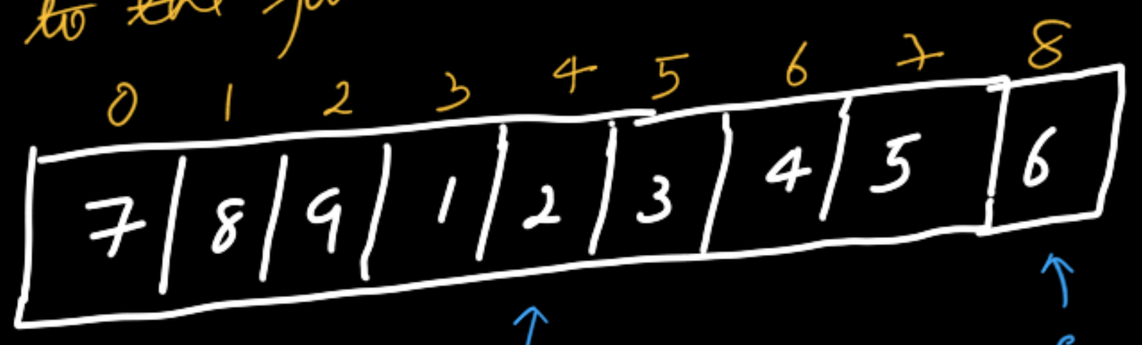
→ Number of occurrences

↳ first occurrence = 3
last occurrence = 5 → so (5-3)+1 = 3

★ No. of occurrences = last occurrence − first occurrence + 1

→ Search in Rotated Sorted Array 1

↳ Initially, array is rotated at some pivot K prior to being passed to the function.



target = 7

m = (0+8)/2 = 4

arr[mid] ≤ arr[end]

right sorted part

arr[mid] ≤ target ≤ arr[end]

target doesn't lie in this part of sorted array ↓

2 ≤ 1 ≤ 6 ✗

↳ e = m − 1

Observation

① Identify which part of the array is sorted

↓

left / right

② Since right part is sorted we can check if target lies in this part

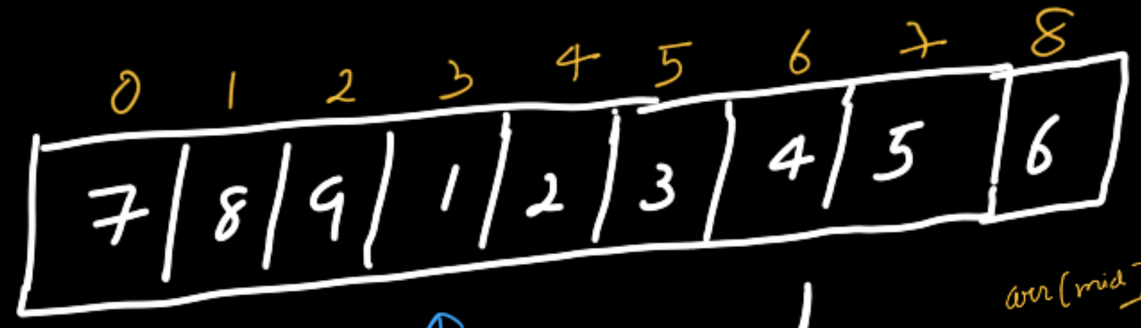
↓

arr[mid] ≤ target ≤ arr[end]

③ For left sorted part,

↓

arr[start] ≤ target ≤ arr[mid]



arr[mid] ≤ target ≤ arr[end]

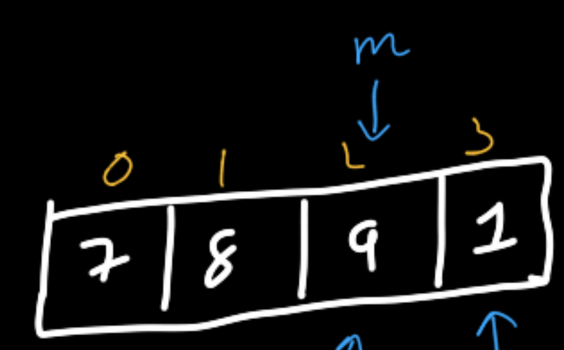
8 ≤ 1 ≤ 1 ✗

arr[start] ≤ target ≤ arr[mid]

7 ≤ 1 ≤ 8 ✗

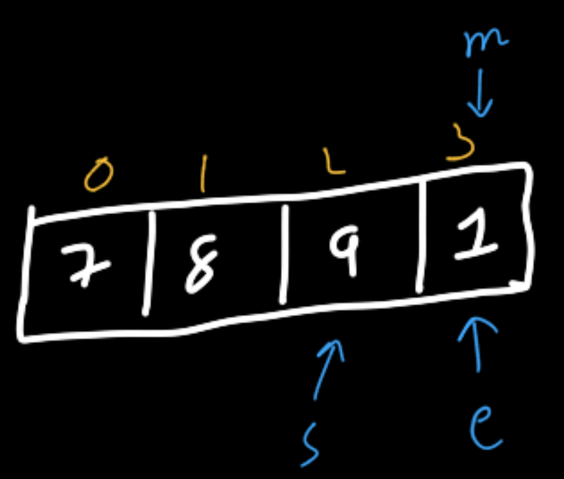
↳ does the target lie in between {arr[start], arr[mid]} ✗

↳ s = m + 1



9 ≤ 1 ≤ 9 ✓

↳ target may lie in this range



does the target lie in between {arr[start], arr[mid]} ✗

↳ s = m + 1

↳ 1 ≤ 1 ≤ 1 ✓

↳ Answer = 3

```java
PartIII.java
public static int rotatedSearch(int[] nums, int target) {  1 usage  new *
    int start = 0;
    int end = nums.length - 1;

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

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

        // Left half is sorted
        if (nums[start] <= nums[mid]) {
            if (nums[start] <= target && target < nums[mid]) end = mid - 1;
            else start = mid + 1;
        }
        // Right half is sorted
        else {
            if (nums[mid] < target && target <= nums[end]) start = mid + 1;
            else end = mid - 1;
        }
    }
    return -1;
}
```

→ TC : $O(\log_2 n)$

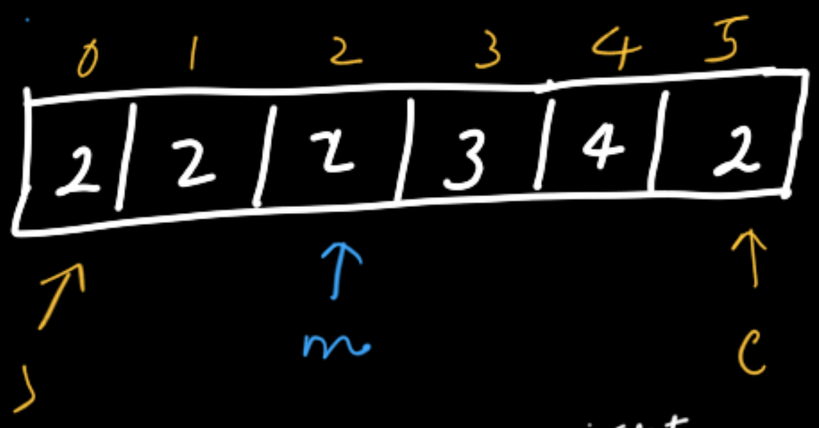
SC : $O(1)$

→ Search in rotated Sorted Array II

↳ Initially, array is rotated at some pivot k prior to being passed to the function.

- Also contains duplicate

↳ Why above solution doesn't work for array containing duplicates?



target = 3

s=0, e=5

m = (0+5)/2

= 2

① Checking if it is sorted right

↳ arr[m] ≤ arr[e] (2≤2)

↓

we move to left part of the array → we may think maybe left part is sorted, thus missing our target even though its there.

② Checking if it is sorted left

↳ arr[s] ≤ arr[m] (2≤2)

```java
PartIII.java
public static boolean rotatedSearchDuplicate(int[] nums, int target) {  no u
    int start = 0;
    int end = nums.length - 1;

    while (start <= end) {
        int mid = start + (end - start) / 2;
        if (nums[mid] == target) return true;

        if (nums[start] == nums[mid] && nums[mid] == nums[end]){
            start++;
            end--;
            continue;
        }
        // Left half is sorted
        if (nums[start] <= nums[mid]) {
            if (nums[start] <= target && target < nums[mid]) end = mid - 1;
            else start = mid + 1;
        }
        // Right half is sorted
        else {
            if (nums[mid] < target && target <= nums[end]) start = mid + 1;
            else end = mid - 1;
        }
    }
    return false;
}
```

So whenever we stumble upon condition where

arr[start] = arr[mid] = arr[end]

↓

we keep on incrementing start & decrementing end

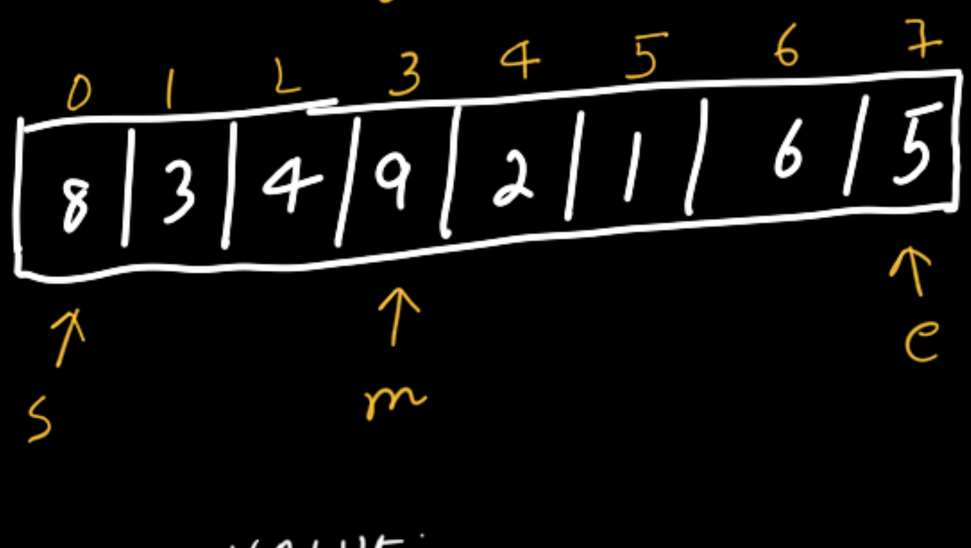
★ we just shrink the search space over the uncertain region where we can't trust sortedness due to equal values.

TC: $O(\log_2(n))$

SC: $O(1)$

→ Find Minimum in Rotated Sorted Array

↳ Sorted array of unique elements, we have to return minimum element of this array



$m = (0+7)/2 = 3$

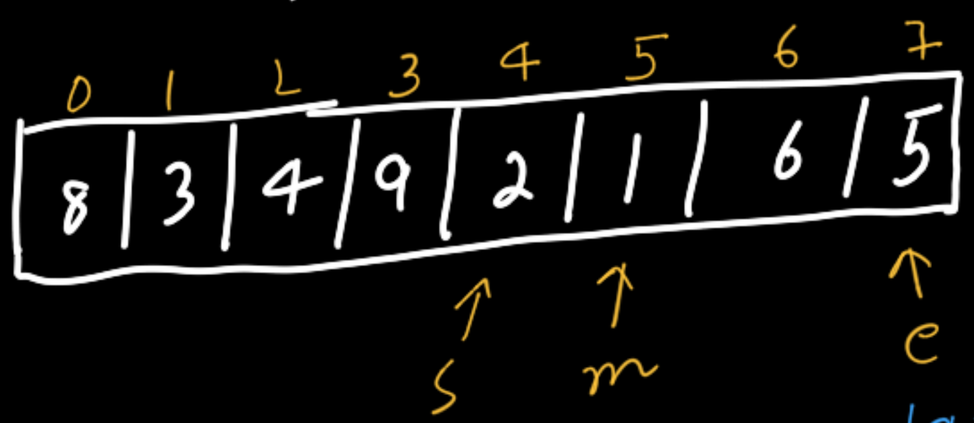
ans = Integer.MAX_VALUE;

① nums[start] ≤ nums[mid]

8 ≤ 9 ✓

ans = min(ans, nums[start]) ≈ 8

s = m + 1



② s = 4, e = 7, $m = (4+7)/2 = 5$

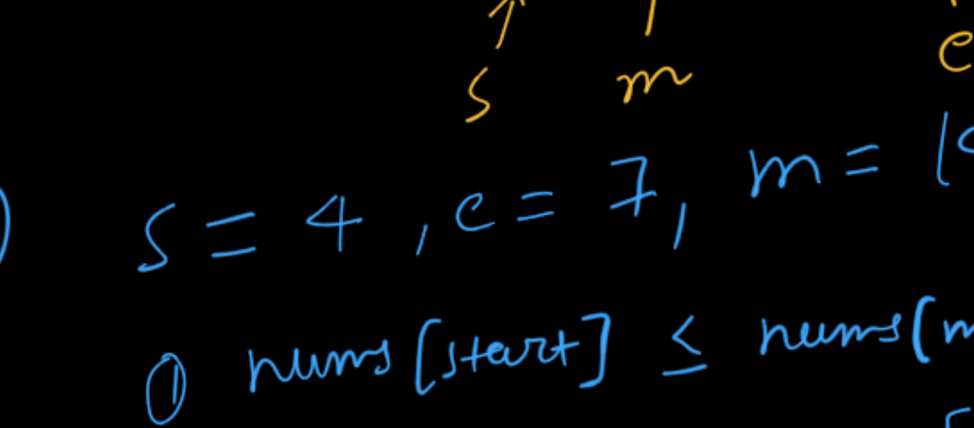
① nums[start] ≤ nums[mid] ✗

② nums[start] > nums[mid]

↳ ans = min(ans, arr[mid]) ≈ 1

c = m - 1

③ s = 4, e = 4, m = 4



① nums[start] ≤ nums[mid] ✓

ans = min(ans, nums[start]) ≈ 1

1 2

s = m + 1 End of loop as s > e

Observation

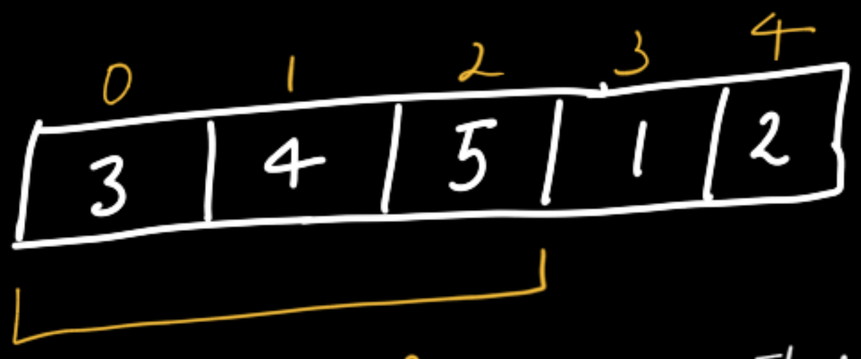
① Find which part of array is sorted either left / right

- nums[start] ≤ nums[mid]
  ↳ Left part is sorted
  start = mid + 1
  ans = min(ans, arr[start])
- nums[start] > nums[mid]
  ↳ end = mid - 1
  ans = min(ans, arr[mid])

PartIV.java

```java
public static int findMin(int[] nums) {  1 usa
    int start = 0;
    int end = nums.length - 1;
    int ans = Integer.MAX_VALUE;

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

        if (nums[start] <= nums[mid]){
            ans = Math.min(ans,nums[start]);
            start = mid + 1;
        }else {
            end = mid - 1;
            ans = Math.min(ans,nums[mid]);
        }
    }
    return ans;

}
```

→ Number of times array has been rotated

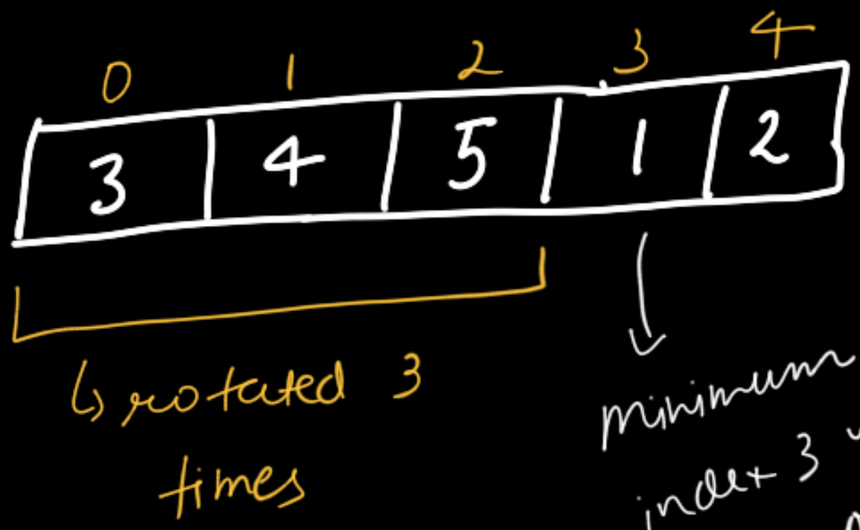


↳ rotated 3 times

→ So if we find out where minimum of the array is we can find out Number of times array is rotated.

↓

By looking at the index of the minimum number.



↳ rotated 3 times

minimum is at index 3 which is number of times array is rotated.

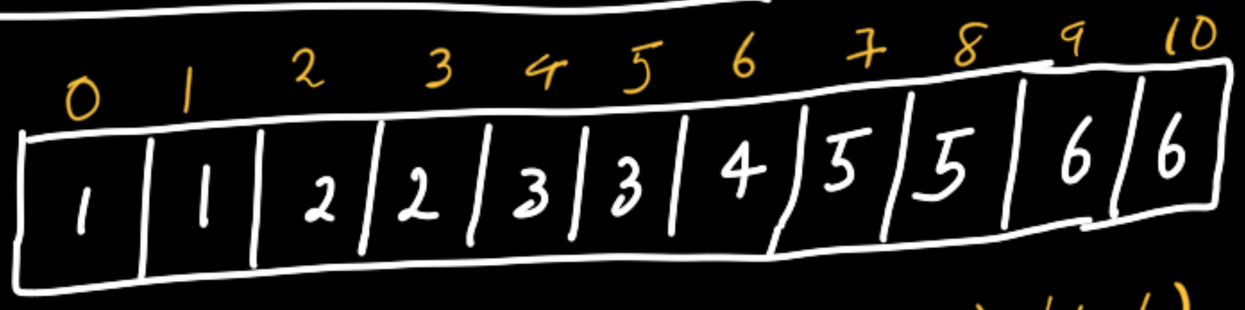
```java
public static int findKRotation(int[] nums) {
    int start = 0;
    int end = nums.length - 1;
    int ans = Integer.MAX_VALUE;
    int index = -1;

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

        if (nums[start] <= nums[mid]){
            if (nums[start] < ans){
                ans = Math.min(ans,nums[start]);
                index = start;
            }
            start = mid + 1;
        }else {
            if (nums[mid] < ans){
                ans = Math.min(ans,nums[mid]);
                index = mid;
            }
            end = mid - 1;
        }
    }
    return index;

}
```

we only want to update ans & index if the next set of ans & index is less than previous set

if we don't put up these condition, we might overwrite a smaller minimum with a bigger value because we are blindly Assigning index = start & index = mid every time we compare, even if the value at start or mid is not actually smaller than our current minimum.

These conditions ensure we only update when we find a smaller value.

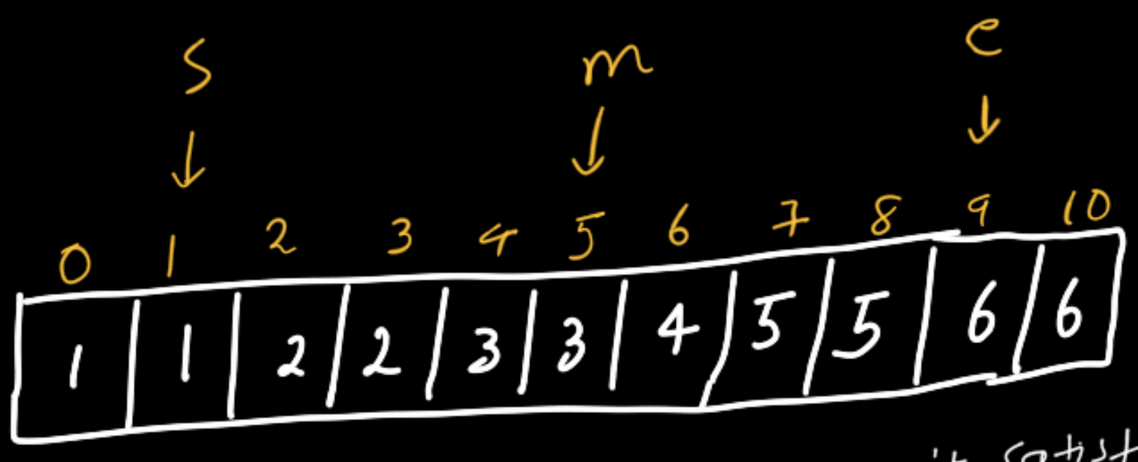
→ Single Element in Sorted Array



0 → odd
e → even

(1,1) (2,2) (3,3) (5,5) (6,6)
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
e o e o e o o e o e

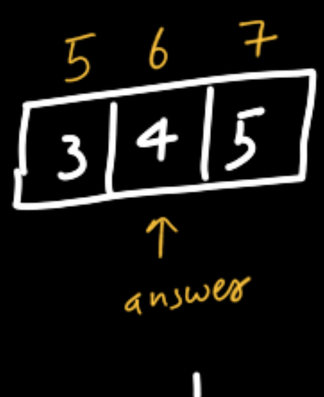
Intuit →
→ we are eliminating 1st & last element by using simple if checks.



① is mid our answer ✗ → doesn't satisfy the condition
arr[index-1] != arr[index] != arr[index+1]

② now we will eliminate the part where our element is not there.
↓
(even, odd) → mid is at odd index
↳ we are standing at odd index, so if that (odd-1) index is even then we surely know where our answer lie
↓
surely nums[mid] must equal to nums[mid-1]
↓
this tell us that we are on left side of the array
↓
i.e it lies on the right half of our array so, we will move s = m+1

else → we are on right side of our array



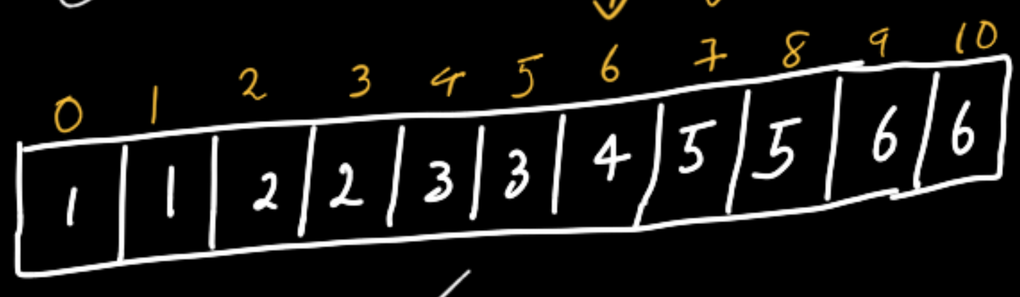
↑ answer

Observation

① By observing this, we can infer that, when
(e,o) → standing on left side of the array
(o,e) → standing on right side of the array

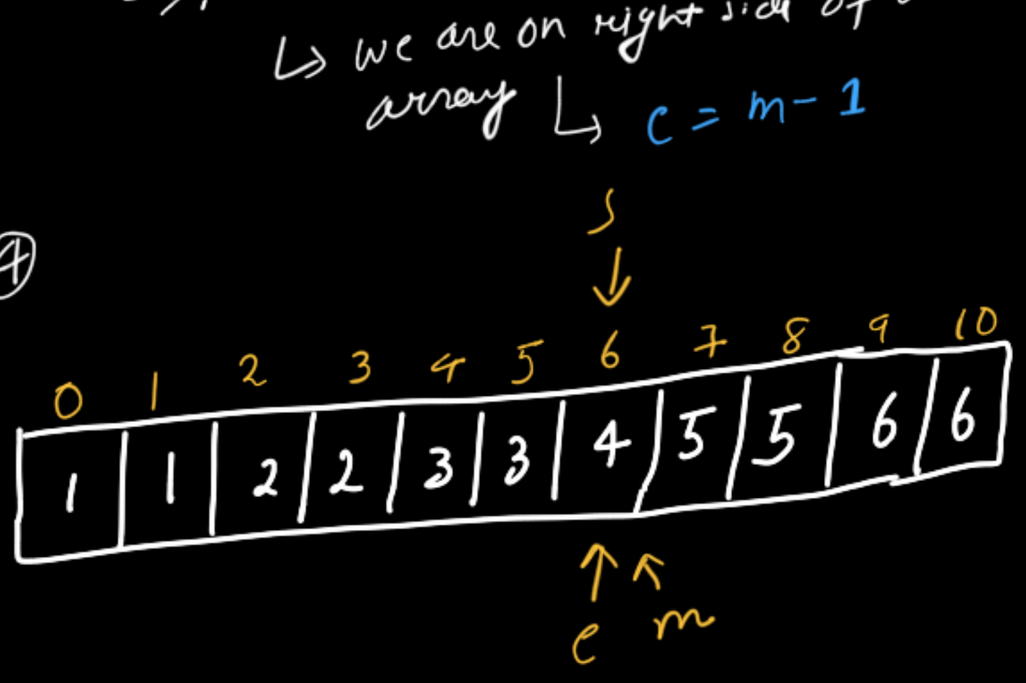
② if we are at our answer, both left & right of the answer will be equal i.e
arr[index-1] != arr[index] != arr[index+1]

③



is mid our answer ✗ → doesn't satisfy the condition
arr[index-1] != arr[index] != arr[index+1]
↓
(even, odd)
↳ now is nums[mid] == nums[mid-1] ✗
↳ we are on right side of our array ↳ e = m-1

④



is mid our answer ✓ → does satisfy the condition
arr[index-1] != arr[index] != arr[index+1]

```java
public static int singleNonDuplicate(int[] nums) {  1 usage  new *
    if (nums.length == 1) return nums[0];

    if (nums[0] != nums[1]) return nums[0];
    if (nums[nums.length - 1] != nums[nums.length - 2]) return nums[nums.length - 1];

    int start = 1;
    int end = nums.length - 2;

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

        if (nums[mid] != nums[mid - 1] && nums[mid] != nums[mid + 1]) return nums[mid];

        // Check if mid is in the left (well-formed) part where pairs are starting at even indices.
        // For even mid: nums[mid] == nums[mid + 1]
        // For odd mid:  nums[mid] == nums[mid - 1]
        // If this pattern holds, the single element is on the right.


        if (mid % 2 == 1 && nums[mid - 1] == nums[mid] ||
                mid % 2 == 0 && nums[mid] == nums[mid + 1]) {
            start = mid + 1; // eliminate left half
        } else {
            end = mid - 1; // eliminate right half
        }
    }

    return -1;

}
```

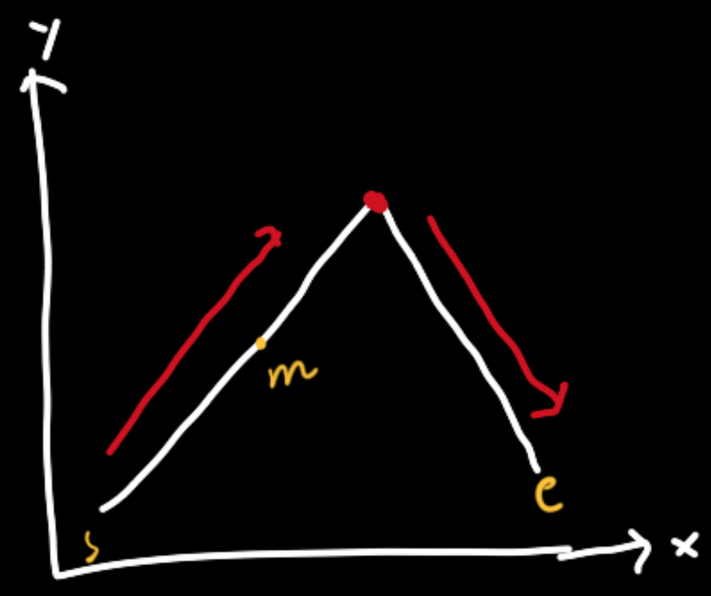
→ Find Peak element

↳ • $arr[i-1] < arr[i] > arr[i+1]$

• $nums[-1] = nums[n] = Integer.MIN\_VALUE$

0	1	2	3	4	5	6
1	2	1	3	5	6	4

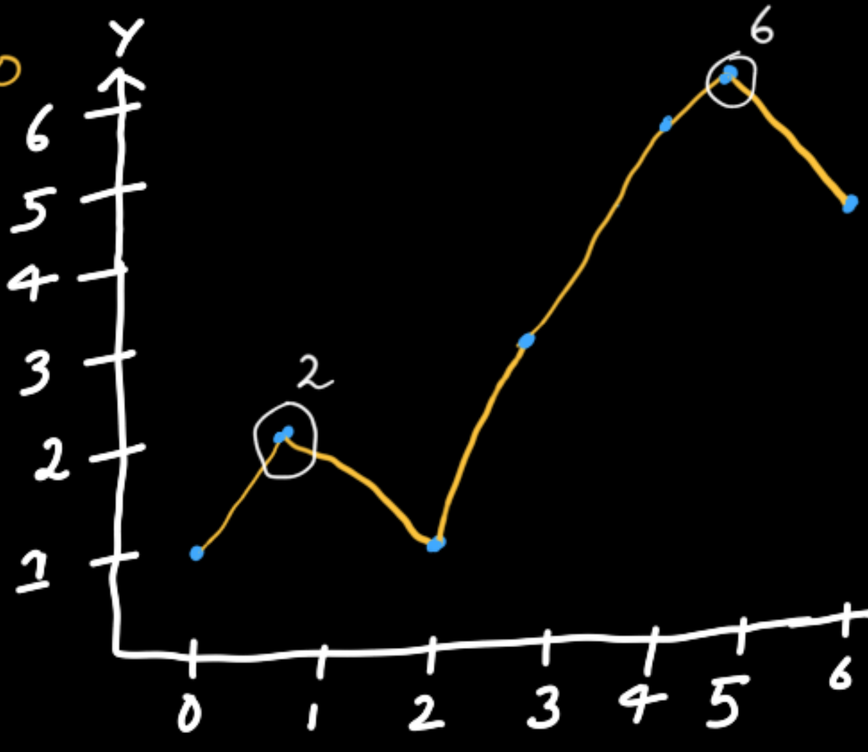
$-\infty$ ... $-\infty$



Observations:

① Strictly increasing side ↓ peak is at right

② Strictly decreasing side ↓ peak is at left



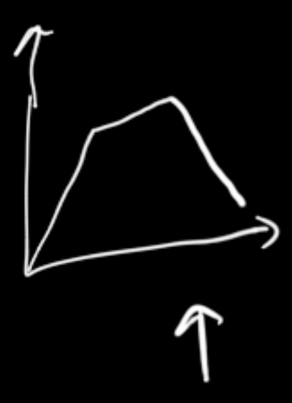
Since arr[0] & arr[n] will only have to compare with one element

i.e $-\infty < arr[0] > arr[1]$

$arr[n-2] < arr[n-1] > -\infty$

↓

We can perform simple if check for these two condition



what if arr[mid] = arr[mid+1] & arr[mid-1] i.e we have a flat lines ↓ recursively do bs on left & right take max(left,right)

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

s m e

is $arr[mid-1] < arr[mid] > arr[mid+1]$ ✗

is $arr[mid] < arr[mid+1]$ ✓

↳ we are in strictly increasing side so 1 answer will lie on our right side

↳ $s = m+1$

else → we are in strictly decreasing side, so answer will lie on our left side

↳ $e = m-1$

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

s m e

is $arr[mid-1] < arr[mid] > arr[mid+1]$ ✗

is $arr[mid] < arr[mid+1]$ ✓

↳ we are in strictly increasing side so 1 answer will lie on our right side

↳ $s = m+1$

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

s m e

is $arr[mid-1] < arr[mid] > arr[mid+1]$ ✓

↓

answer

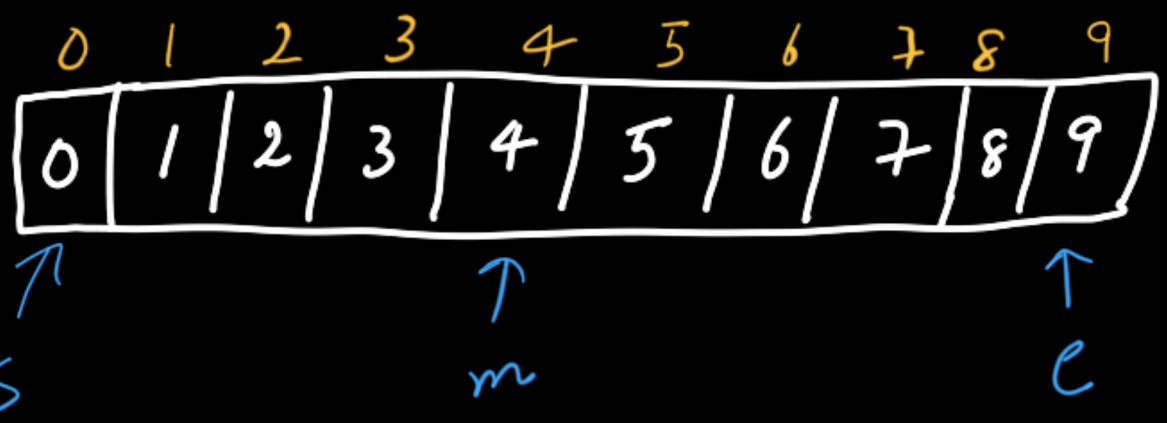
```java
PartVII.java
public static int findPeak(int[] arr) {  1 usage  new *
    int len = arr.length;
    if (len == 1) return 0;

    if (arr[0] > arr[1]) return 0;
    if (arr[len - 1] > arr[len - 2]) return len - 1;

    int start = 1;
    int end = len - 2;

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

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

        if (arr[mid] < arr[mid + 1]) start = mid + 1;
        else end = mid - 1;
    }
    return -1;
}
```

→ Find square root of a given number

↳ $\sqrt{25} = 5$, $\sqrt{36} = 6$, $\sqrt{38} = 6.2\ldots \simeq 6$

↳ return floor value

$\sqrt{9}$ ↳



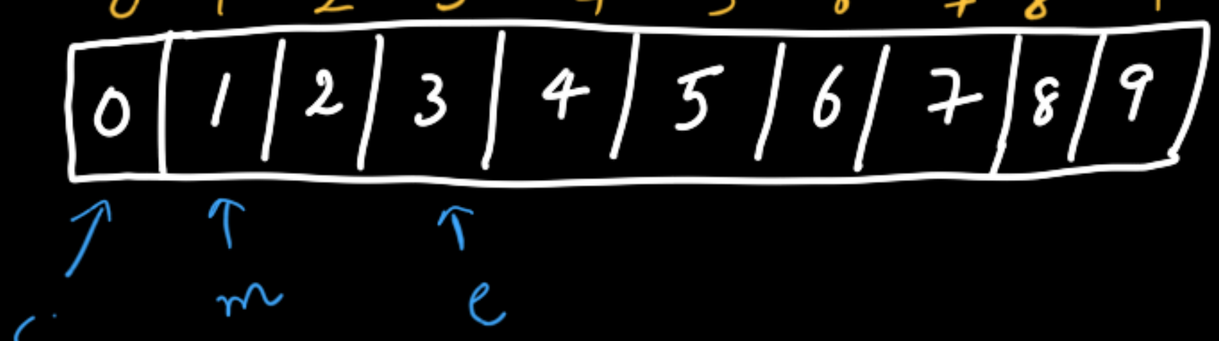
is mid * mid == number ✗

is mid * mid > number ✓

↳ surely our answer will be on the left side of the mid

↓

e = m − 1



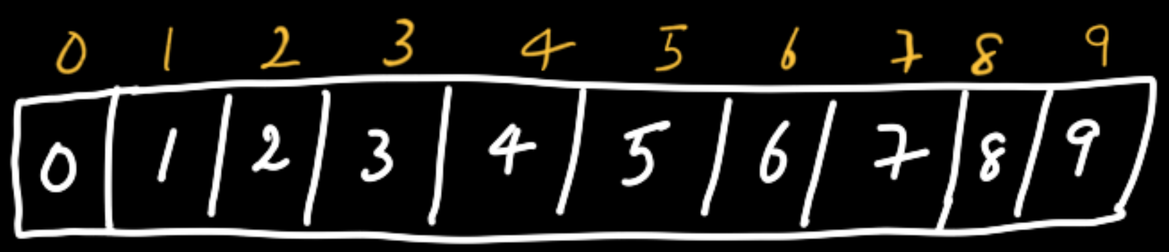
is mid * mid == number ✗

is mid * mid < number ✓

↳ our answer will lie on the right part of the array

↓

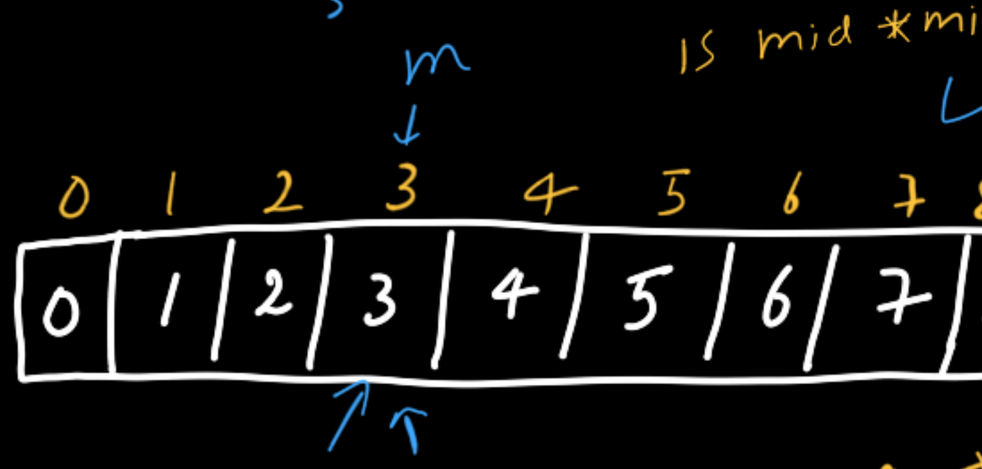
s = m + 1



is mid * mid == number ✗

is mid * mid < number ✓

↳ s = m + 1



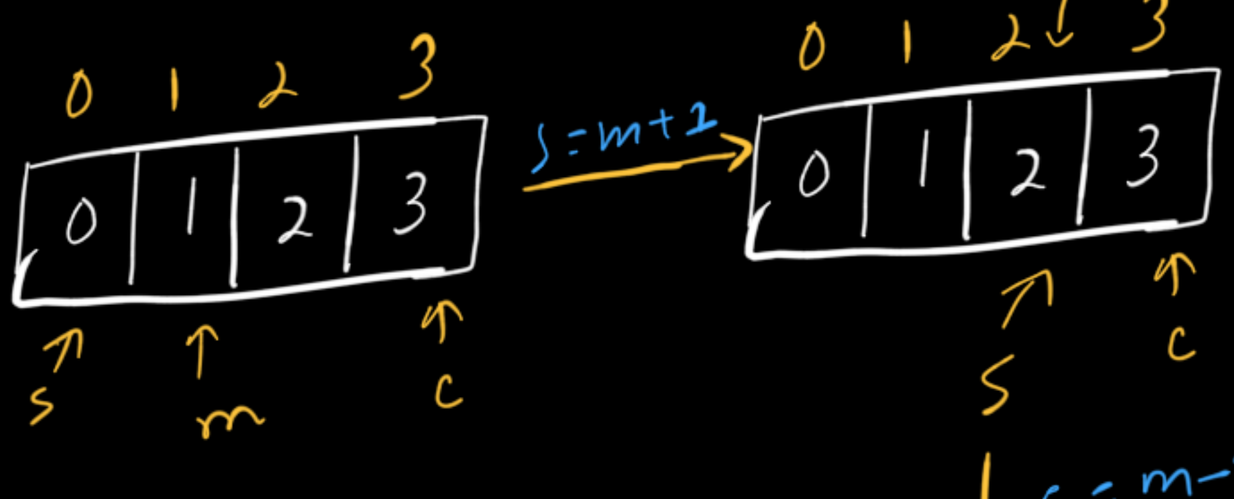
is mid * mid == number ✓

↳ mid is our answer.

add base case, where number == 1, → return number itself → optimization

For non-perfect square

```
PartVIII.java
public static int root(int number){  1 usage  ne
    int start = 0;
    int end = number;                          → n/2

    while (start <= end){
        int mid = (start + end )/ 2;

        if (mid * mid == number) return mid;
        if (mid * mid > number) end = mid - 1;
        else start = mid + 1;
    }
    return end;
}
```



So whenever end < start that will be our floor of the non-perfect square number.

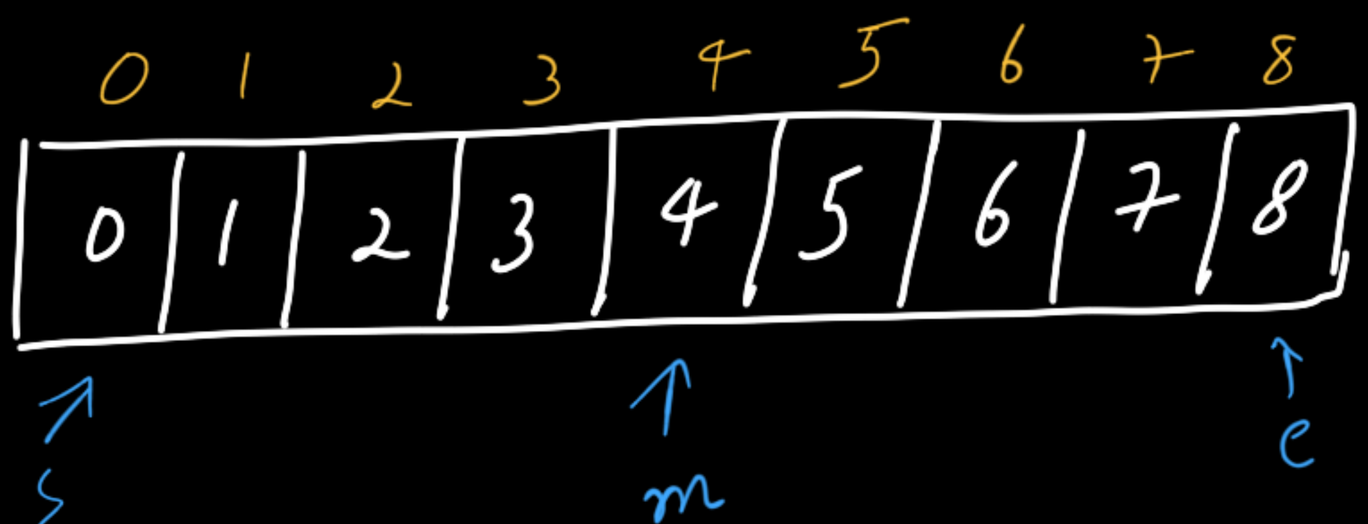
↳ loop ends

→ Find n^{th} root of a number

↳ Find $\sqrt[n]{m}$, Ex: $\sqrt[3]{8} = 2$, $\sqrt[4]{16} = 2$

$\sqrt[3]{8}$

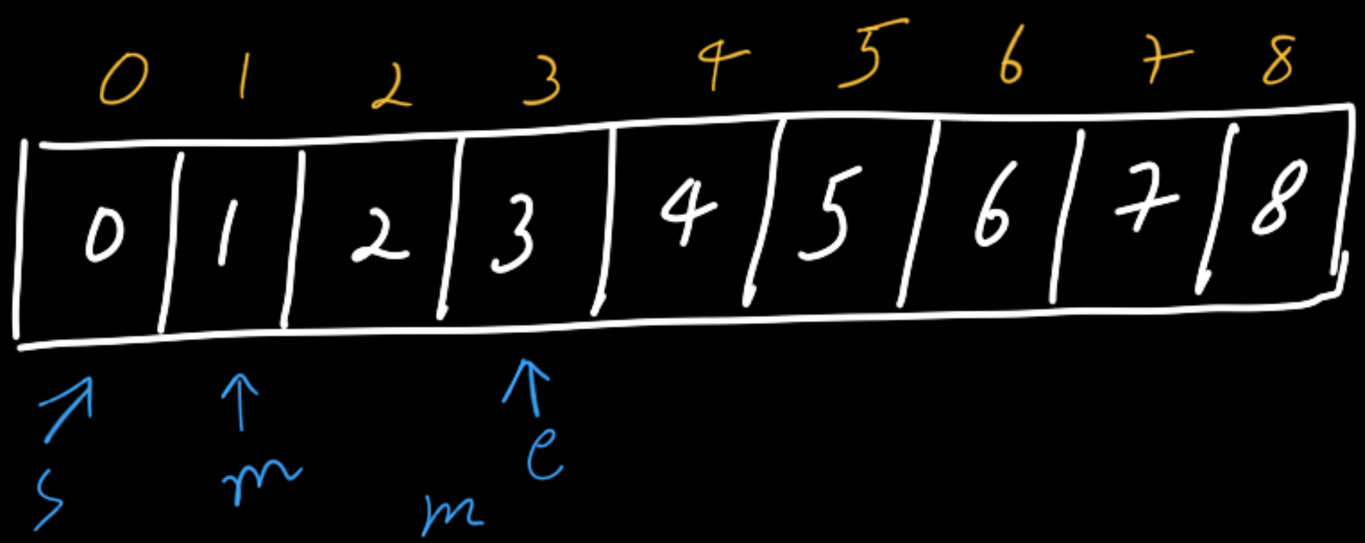
↳



is mid^n == number ✗

is mid^n > number ✓

↳ e = m - 1

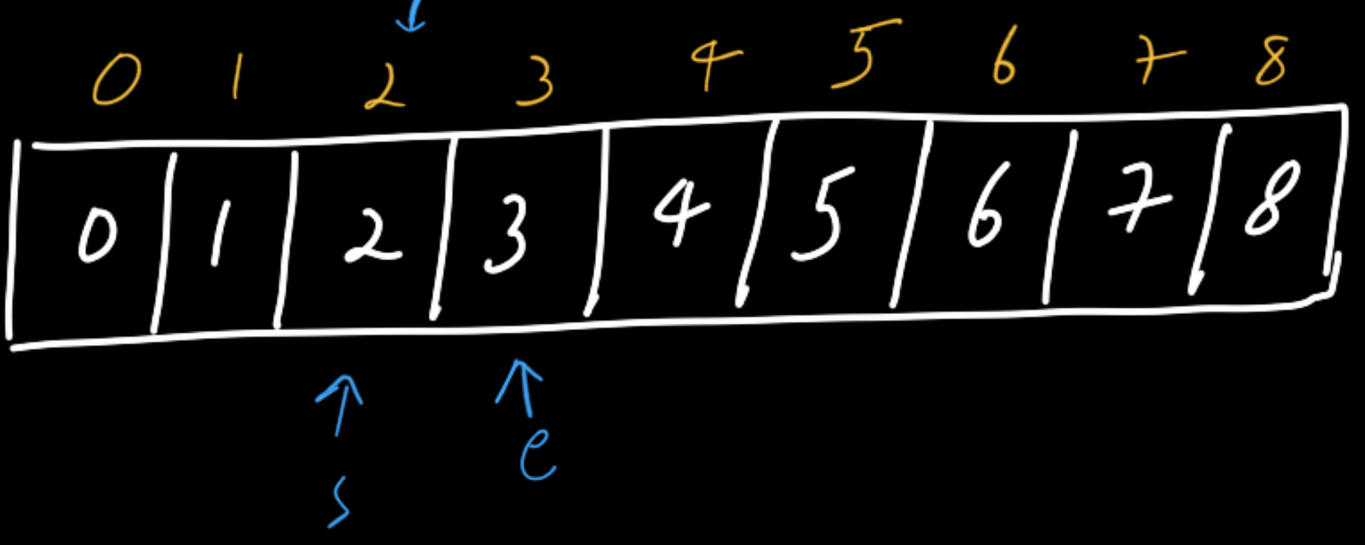


is mid^n == number ✗

is mid^n > number ✗

is mid^n < number ✓

↳ s = m + 1



is mid^n == number ✓

↳ mid is answer

→ This way we can reduce the search space by n

↓

edge case

↓

if number == 1 dividing by n will result in 0.

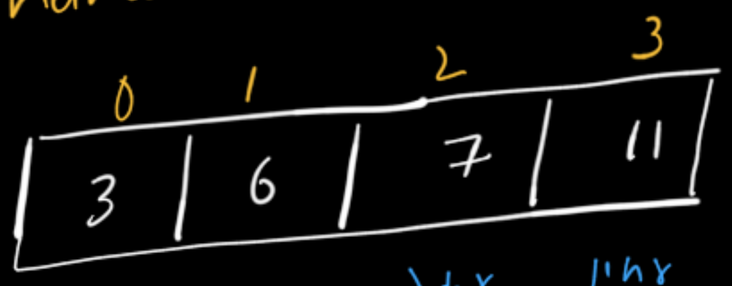
↓

So, we just return the number itself by doing a simple if check.

```java
PartVIII.java
public static int nroot(int number , int n){ 1u
    int start = 0;
    int end = number / n;

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

        long square = (long) Math.pow(mid,n);

        if (square == number) return mid;
        else if (square > number) end = mid - 1;
        else start = mid + 1;
    }

    return -1;
}
```

→ Koko eating banana

↳ Return the minimum integer k such that Koko can eat all banana within h hours.



h = 8

3hr 6 hr 7hr 11hr = 27hr > 8

2hr 3 hr 4hr 6hr = 15hr > 8

1hr 2 hr 3hr 4hr = 10hr > 8

1hr 2hr 2hr 3hr = 8 hr = 8hr

↳ So Koico can eat 4 banana/hr within 8 hour.

Assumption

- Koico can eat
  1 banana/hour ✗
- 2 banana/hour ✗
- 3 banana/hour ✗
- 4 banana/hour ✓

So, Required total hour to eat all piles of banana is

↓

(ceil (arr[i] / hourly)

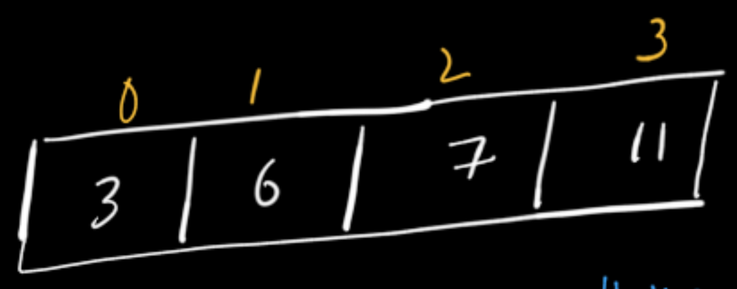
↓

we need smallest number such that the number ≥ target

↳ We need to calculate this for every mid.

What is ~~the~~ maximum number of bananan Can Koico eat hourly?

↳ to set upper bound (end) pointer of the array.



1hr 1hr 1hr 1hr = 4hr < h 13 banana/hour

1hr 1hr 1hr 1hr = 4hr < h 12 banana/hour

1hr 1hr 1hr 1hr = 4hr < h 11 banana/hour

1hr 1hr 1hr 2hr = 5hr < h 10 banana/hour

Find more minimum

possible answer

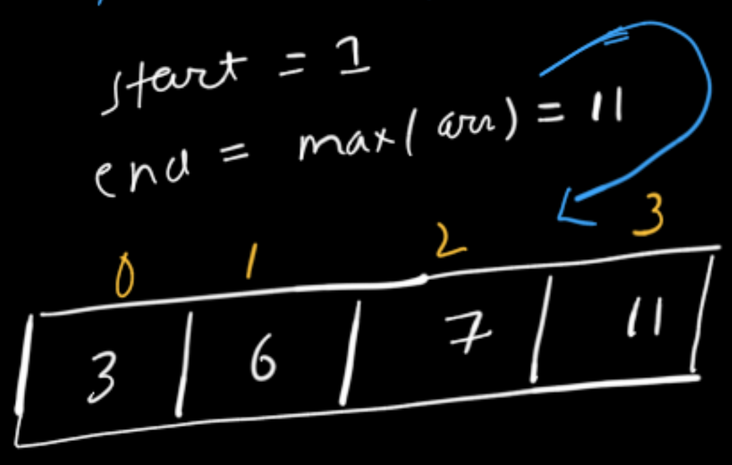
So, the maximum number of banana Koico eat hourly is

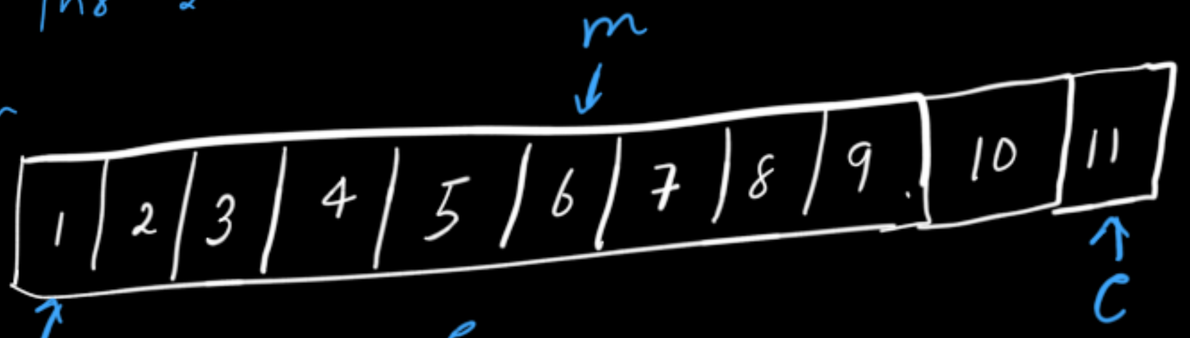
↓

The maximum element in the array.

start = 1

end = max(arr) = 11





For mid = 6

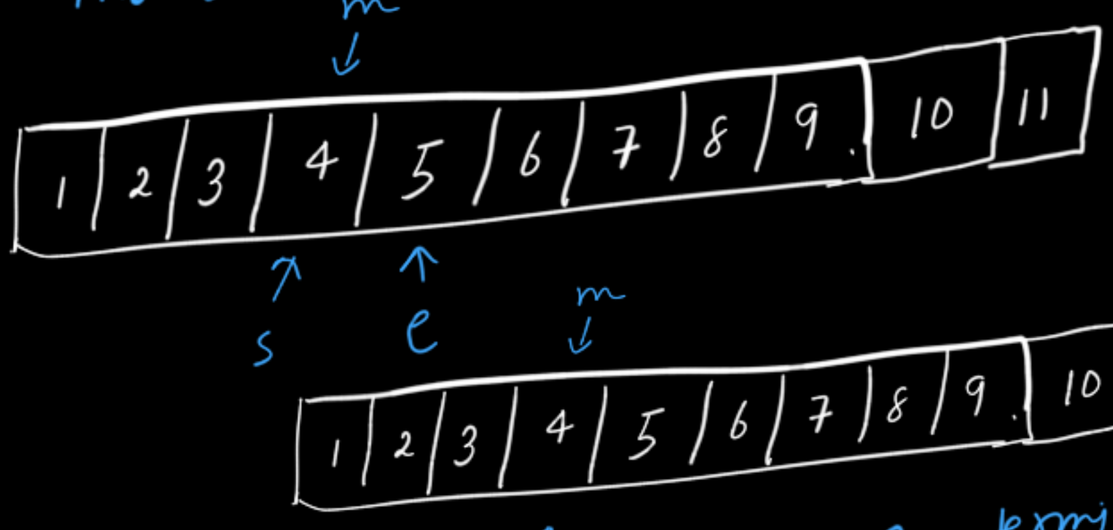
1hr 1hr 2hr 2hr ≤ h = 8 ✓

For mid = 3

1hr 2hr 3hr 4hr > h = 8 ✗

For mid = 4

1hr 2hr 2hr 3hr ≤ h = 8 ✓



e s s > e terminate

start will always be at answer

↳ return start

```java
public static int koko(int[] arr, int hourly) {  no usages  bravo1goingdark

    int start = 1;
    int end = getMax(arr);

    while (start <= end){
        int mid = start + (end - start) / 2;
        int req = totalHour(arr,mid);

        if (req <= hourly) end = mid - 1;
        else start = mid + 1;
    }
    return start;
}

private static int totalHour(int[] arr, int hourly) {  1 usage  bravo1going
    double total = 0;

    for (int e : arr) total += Math.ceil(((double) e / (double) hourly));
    return (int) total;
}


private static int getMax(int[] arr) {  1 usage  bravo1goingdark
    int max = Integer.MIN_VALUE;

    for (int e : arr) max = Math.max(max, e);
    return max;
}
```

TC: O(N * log(max(arr)))

SC: O(1)

→ Minimum no. of days to make M bouquets

↳ Minimum number of days, we need to wait to be able to be able to make m bouquets from the garden.

↳ if it is impossible, return 1

1	10	3	10	2

m = 3, k = 1

↳ we need to make m bouquets using k adjacent flower.

day-1 → [✓, ×, ×, ×, ×,]

↳ 1 Flowers can bloom on day-1 < m ×

day-2 → [✓, ×, ×, ×, ✓]

day-3 [✓, ×, ✓, ×, –]

2 Flowers can bloom on day-2 < m ×

3 Flowers can bloom on day-3 = m ✓

So, 3 days would be the answer.

① What is the earliest possible day when any flower blooms? → min(array)

② What is the Latest possible day when any flower blooms? → max(array)

↳ so our search range would be {min, max}

7	7	7	7	13	11	12	7

m = 2, k = 3 min = 7 max = 13

day-7

7	7	7	7	13	11	12	7

count = 0 1 2 3 4

= 0 1

↳ so there is only 1 set of bouquets is possible.

day-12

7	7	7	7	13	11	12	7

count = 0 1 2 3 4

= 0 1 2 3

↳ so, there is 2 set of bouquets with 3 adjacent flowers

↳ Answer.

① For each number check if that no. <= day

↓

count++

↓

else

② do, count/k & then set count = 0

nog Bouquets += count/k

③ whatever count we have at the end of the iteration we will update

nog Bouquets += count/k

① Linear - Search Solution

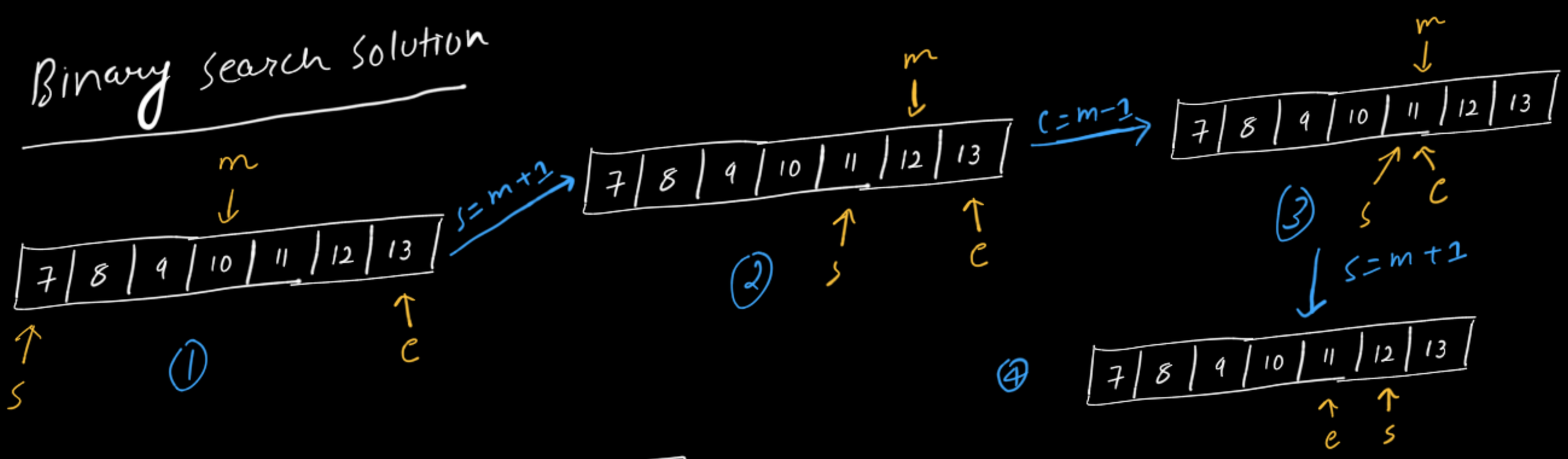
```
for(int i= min; i<= max; i++){
    if| isPossible(arr, i, m, k)){
        return i;
    }
}
return -1;
```

TC: $O((max - min + 1) \times N)$

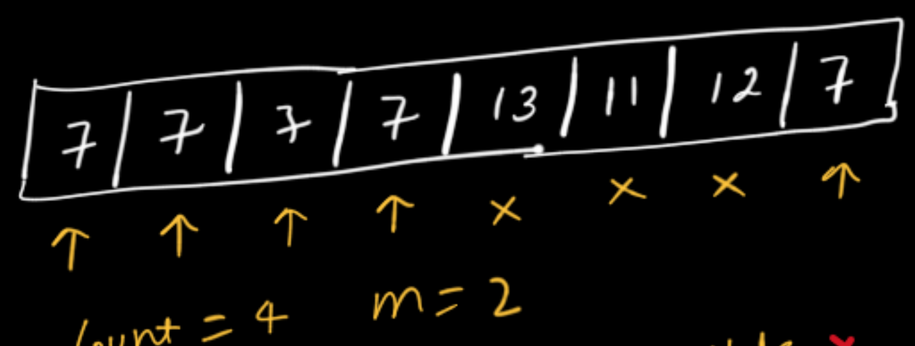
SC: $O(1)$

```
boolean isPossible(int[] arr, int day, int m, int k){
    int count = 0;
    int noofB = 0;
    for(int c : arr ){
        if| c <= day) count++;
        else {
            noofB += count/ k;
            count = 0;
        }
    }
    noofB += count/k;
    return noofB >= m;
}
```

② Binary search solution



Day-10



count = 4 m = 2

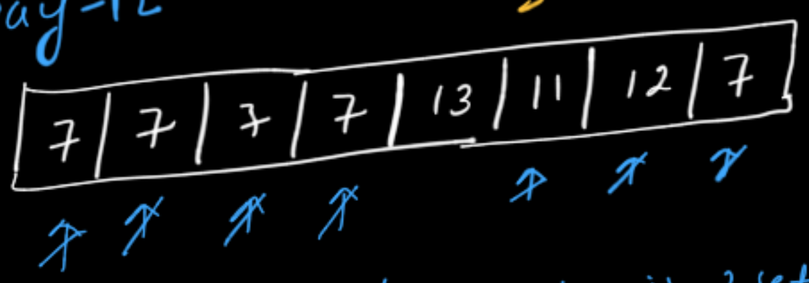
↳ only 1 bouquet possible ✗

So, if day 10 cannot suffice flower for 2 bouquets; then surely day before that cannot so,

↓

$s = m + 1$

Day-12



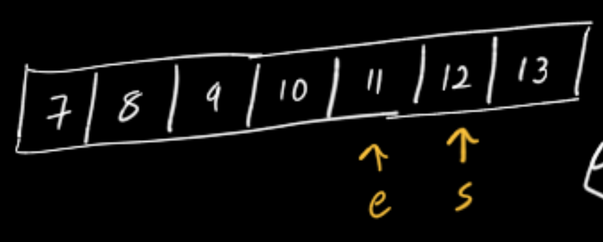
count = ~~0~~ 1 2 3 4

= ~~0~~ 1 2 3

↳ so, there is 2 set of bouquets with 3 adjacent flower

↳ Answer.

- it will keep trying to find minimum even after finding our answer, so logically for finding minimum we should go left i.e $c = m - 1$
- After, $s > e$ → start will be our answer.




```java
public static int minDays(int[] bloomDay, int m, int k) {  1 usage  new
    long v = (long) m * k;

    if (v > bloomDay.length) return -1;
    int start = getMaxMin(bloomDay)[0];
    int end = getMaxMin(bloomDay)[1];

    while (start <= end) {

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

        if (isPossible(bloomDay, mid, m, k)) end = mid - 1;
        else start = mid + 1;
    }
    return start;
}

private static boolean isPossible(int[] arr, int day, int m, int k) {
    int count = 0;
    int noOfBouquet = 0;

    for (int e : arr) {
        if (e <= day) count++;
        else {
            noOfBouquet += count / k;
            count = 0;
        }
    }
    noOfBouquet += count / k;
    return noOfBouquet >= m;
}

private static int[] getMaxMin(int[] arr) {  2 usages  new *
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;

    for (int e : arr) {
        max = Math.max(max, e);
        min = Math.min(min, e);
    };

    return new int[]{min, max};
}
```

→ Find the Smallest Divisor given a threshold

↳ We will choose a positive integer divisor, such that it divide all array by it & sum the division result. Find the smallest divisor such that the result mentioned above is less than or equal to threshold.

1	2	5	9

threshold = 6

Let that,

divisor = 1	1 + 2 + 5 + 9 = 17 > 6	×
divisor = 2	1 + 1 + 3 + 5 = 10 > 6	×
divisor = 3	1 + 1 + 2 + 3 = 7 > 6	×
divisor = 4	1 + 1 + 2 + 3 = 7 > 6	×
divisor = 5	1 + 1 + 1 + 2 = 5 ≤ 6	✓
divisor = 12	1 + 1 + 1 + 1 = 4 ≤ 6	✓
divisor = 11	1 + 1 + 1 + 1 = 4 ≤ 6	✓
divisor = 10	1 + 1 + 1 + 1 = 4 ≤ 6	✓
divisor = 9	1 + 1 + 1 + 1 = 4 ≤ 6	✓

(low) So the start of the range would be from 1, as it will divide any number

(high) So the end of the range would be from ↓ max(arr)

```
boolean isFeasible(int[] arr, int threshold, int divisor){
    int sum = 0;
    for(int e : arr){
        sum += Math.ceil(e/divisor);
    }
    return sum ≤ threshold;
}
```

PartXI.java

```java
public static int smallestDivisor(int[] nums, int threshold) {  no usages  ne
    int start = 1;
    int end = getMax(nums);

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

        if (isFeasible(nums,threshold,mid)){
            end = mid - 1;
        }
        else start = mid + 1;
    }
    return start;

}

private static boolean isFeasible(int[] arr, int threshold, int divisor) {
    double sum = 0;

    for (int e : arr) {
        double ce = ((double) e) / ((double) divisor);
        sum += Math.ceil(ce);
    }
    return sum <= threshold;
}

private static int getMax(int[] arr) {  1 usage  new *
    int max = Integer.MIN_VALUE;

    for (int e : arr) {
        max = Math.max(max, e);
    };
    return max;
}
```

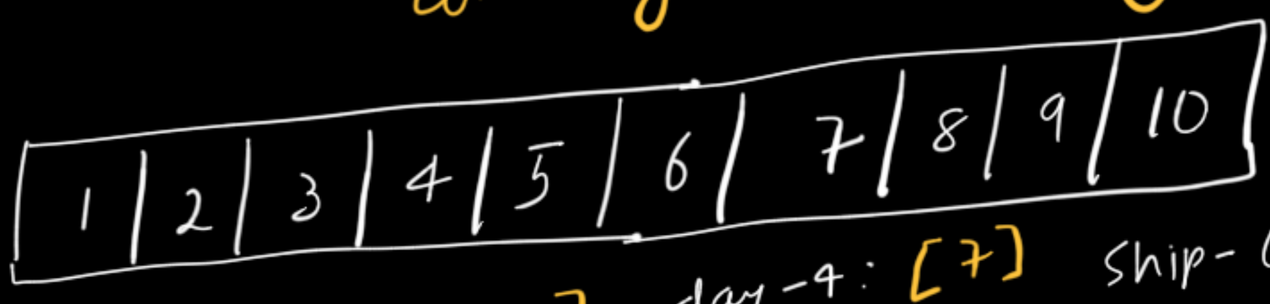
TC: $N \times \log(max)$

SC: $O(1)$

- Same as previous question
  - ↳ At the end start will always have our answer.

→ Capacity to ship packages within D days

↳ return least weight capacity of ~~the~~ ship that will result in all packages on the conveyor belt being shipped within d days.



days = 5

day-1 : [1, 2, 3, 4]
day-2 : [5]
day-3 : [6]
day-4 : [7]
day-5 : [8]
day-6 : [9]
day-7 : [10]

Ship-capacity of 10 takes 7 days to ship all the packages which is > 5.

day-1 : [1, 2, 3, 4, 5] ≤ C
day-2 : [5, 6] ≤ C
day-3 : [7, 8] ≤ C
day-4 : [9] ≤ C
day-5 : [10] ≤ C

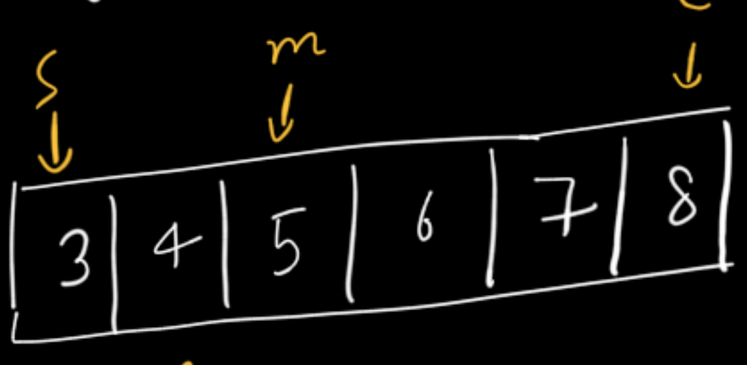
Ship-capacity of 15 takes 5 days to ship all the packages which is = 5. ↳ Answer

Searching Range:

① At minimum what capacity of the ship packages within D days → max(arr)

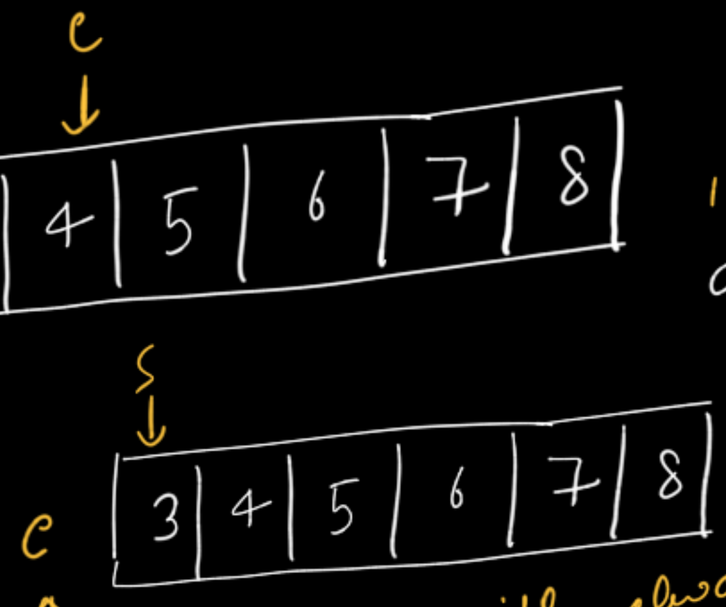
② At maximum what capacity of the ship packages within D days → sum(arr) → will take 1 day to ship all packages.

weights[] = [1 | 2 | 3 | 1 | 1] days = 4 start = max(arr) = 3 end = sum(arr) = 8



is req days till 5 feasible?
day-1 : [1, 2] day-2 : [3, 1, 1]
2 ≤ days ✓
keep finding minimum,
↳ c = m − 1

is req days till 3 feasible?
day-1 : [1, 2] day-2 : [3]
day-3 : [1, 1] 3 ≤ days ✓



↳ start will always be at our answer.

```java
PartXII.java
public int shipWithinDays(int[] weights, int days) {  no
    int start = getMax(weights);
    int end = Arrays.stream(weights).sum();

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

        if (reqDays(weights,mid) <= days) end = mid - 1;
        else start = mid + 1;
    }

    return start;
}

private static int reqDays(int[] arr , int cap){  1 usage
    int days = 0;
    int load = 0;

    for (int e : arr){
        if (load + e > cap) {
            days += 1;
            load = e;
        }
        else load += e;
    }
    return days + 1;
}

private static int getMax(int[] arr) {  1 usage  new *
    int max = Integer.MIN_VALUE;

    for (int e : arr) {
        max = Math.max(max, e);
    };
    return max;
}
```

TC : O(N × log(sum))
SC : O(1)

→ Kth missing positive number

↳ Return Kth positive integer i.e missing from this array.

0	1	2	3	4
2	3	4	7	11

K = 5

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

missing

Linear - solution

0	1	2	3	4
2	3	4	7	11

K = ~~5~~ ~~6~~ ~~7~~ ~~8~~ 9

I ✗ ✗ ✗ ↑

```
for(int e: arr){
  if(e <= k) k++;
  else break;
}
return k;
```

↳ TC : O(N)

0	1	2	3	4
2	3	4	7	11

k <= 5, ans = 9

↳ we can clearly say that answer must lie between 7 & 11

→ (1) Find two nearby indexes, our answer lie in this range.

0	1	2	3	4
2	3	4	7	11

— (minus)

1	2	3	4	5

Missing 1 1 1 3 6

missing = arr[i] - (i+1)

Ideally, if there wasn't any missing number then array should have been

↳

1	2	3	4	5

0	1	2	3	4
2	3	4	7	11

1 1 1 3 6

s e

m

mid

high

low

m < K → s = m+1

s

0	1	2	3	4
2	3	4	7	11

1 1 1 3 6

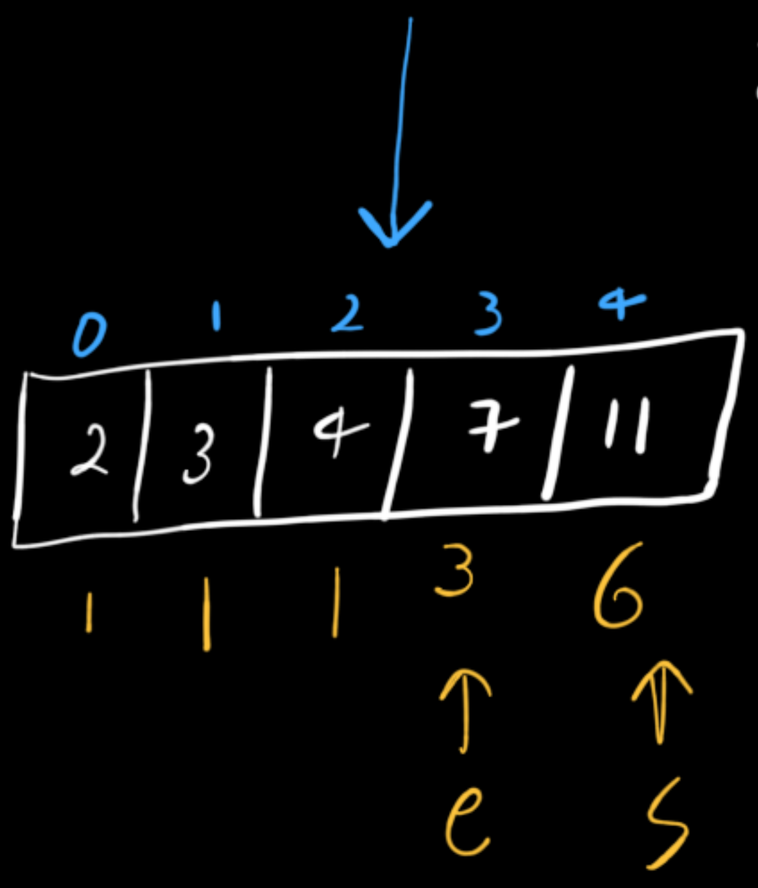
s e

K = 5

missing < 5

↳ s = m+1

missing > k → e = m - 1



e > s

{arr[e], arr[s]} → range where our answer will be there.

Answer = arr[end] + more (needed)

arr[e] = 7, missing = 3, more = 2

missing = arr[e] - [e+1]

more = (k - missing)

= arr[e] + more

= arr[e] + (k - missing)

= arr[e] + k - [arr[e] - [e+1]]

= arr[e] + k - arr[e] + e + 1

= k + e + 1

start after loop termination as e > s

= start + k

PartXIII.java

```java
public static int findKthPositive(int[] arr, int k) {
    int start = 0;
    int end = arr.length - 1;

    while (start <= end){
        int mid = start + (end - start) / 2;
        int missing = arr[mid] - (mid + 1);

        if (missing < k) start = mid + 1;
        else end = mid - 1;
    }
    return start + k;
}
```

→ Aggressive cows (Hard)

↳ Assign stalls to k cows such that the minimum distance between any two of them is the maximum possible.

| 0 | 3 | 4 | 7 | 9 | 10 |

K = 4

cows → $C_1 \xleftrightarrow{3} C_2 \xleftrightarrow{1} C_3 \xleftrightarrow{3} C_4$ → min(3,1,3) = 1

$C_1 \xleftrightarrow{4} C_2 \xleftrightarrow{5} C_3 \xleftrightarrow{1} C_4$ → min(4,5,1) = 1

$C_1 \xleftrightarrow{4} C_2 \xleftrightarrow{3} C_3 \xleftrightarrow{3} C_4$ → min(4,3,3) = 3 ← answer.

| 0 | 3 | 4 | 7 | 9 | 10 |

K = 2

$C_1 \xleftrightarrow{10} C_2$

for two cows the max distance such that 2 cows have minimum distance is (max - min)

→ so the range would be from

$\{1, max - min\}$

→ How do we place cows such that the minimum distance between 2 cows is maximum;

Range: $\{1, 10\}$ ↙ let's keep minimum distance of

| 0 | 3 | 4 | 7 | 9 | 10 |

K = 4

① $C_1 \xleftrightarrow{3} C_2 \xleftrightarrow{1} C_3 \xleftrightarrow{3} C_4$ min(3,1,3) = 1

② $C_1 \xleftrightarrow{3} C_2 \xleftrightarrow{4} C_3 \xleftrightarrow{2} C_4$ min(3,4,2) = 2

③ $C_1 \xleftrightarrow{4} C_2 \xleftrightarrow{3} C_3 \xleftrightarrow{3} C_4$ min(4,3,3) = 3 ✓ maximum

④ $C_1 \xleftrightarrow{4} C_2 \xleftrightarrow{3} C_3 \xleftrightarrow{2} C_4$ min(4,3,2) = 2 ✗ (is NOT maximum)

```
boolean canPossible(int[] arr, int dist, int cows) {
    int noOfCows = 1 // 1st cow will always be at index 0
    int lastCows = arr[0] // last cows coordinate
    for(int e : arr) {
        if( e - lastCows >= dist) {
            noOfCows++;
            lastCow = e;
        }
    }
    return noOfCows >= cows;
}
```

→ we can apply binary search, we are trying to maximize the result → s = m + 1

- end will be our answer

whenever we can place k cows. ↓ else e = m - 1

PartXIV.java

```java
public static int aggressiveCows(int[] stalls, int k) {  no usages  new *
    Arrays.sort(stalls);                                  // nlogn
    int start = 1;
    int end = stalls[stalls.length - 1] - stalls[0];

    while (start <= end){                                 // logn
        int mid = start + (end - start) / 2;

        if (canWePlace(stalls,mid,k)) start = mid + 1;
        else end = mid - 1;
    }
    return end;
}

private static boolean canWePlace(int[] stalls , int distance , int cows){
    int noOfCows = 1;
    int lastCow = stalls[0];

    for (int e : stalls){                                 // O(n)
        if (e - lastCow >= distance) {
            noOfCows++;
            lastCow = e;
        }
    }
    return noOfCows >= cows;
}
```

Tc: $n \log n + \log n \times n = O(2n \log n)$

Sc: O(1)

→ Allocate Books / Books Allocation (Hard)

↳ Given: • books[] → books[i] representing number of pages each book have

• There are m student, task is to allocate all the books to the student

↳ We have to allocate the book to m students such that the maximum number of pages assigned to a student is minimum.

if no. of students > no of books
↳ Not possible
↳ return -1;

25	46	28	49	24

students = 4

25 | 46 | 28 | 49 | 24
S_1 S_2 S_3 S_4
maximum no. of page = (49+24) = 73

25 | 46 28 | 49 | 24
S_1 S_2 S_3 S_4
max. no of page = (46+28) = 74

25 46 | 28 | 49 | 24
S_1 S_2 S_3 S_4
max no of page = (25+46) = 71

↳ answer.

Range: {max, sum}

① what is the minimum number of pages we can take initially such that any student can be assigned
1 book → max(arr)

② what is the maximum number of pages we can take initially such that all student can be assigned
1 book → sum(arr)

For above array → start = 49, end = 172

49	50	.....	70	71	.....	110	.....	150	...	171

s, m, e

is 110 pages possible ✗ → e = m-1 , s = 49, e = 109, m = 79
(can we allocate 110)

49	50	...	79	.....	109

is 79 page possible ✗ → e = m-1
s = 49, e = 78, m = 63

49	....	63	...	78

s → ... ← e ✗
↳ s = m+1

64	....	70	.....	78

s → ... ← e
↳ s = m+1

	...	71

e > s → ... ↳ return start

```java
public static int allocateBooks(int[] books , int noOfStudent){ 1 us
    int start = getMax(books);
    int end = Arrays.stream(books).sum();

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

        if (canWeAllocate(books,mid) > noOfStudent) start = mid + 1;
        else end = mid - 1 ;
    }
    return start;
}

private static int canWeAllocate(int[] books , int pages){ 1 usage n
    int countStudent = 1;
    int noOfPages = 0;

    for (int e : books){
        if (noOfPages + e <= pages) noOfPages += e;
        else {
            countStudent++;
            noOfPages = e;
        }
    }
    return countStudent;
}
```

→ Painters Partition / Split array - largest sum

↳ given an arr[] representing board length of K painters, each painting one contigious board, find the minimum time needed to paint all boards such that the maximum time any painter spends is minimized.

| 10 | 20 | 30 | 40 | K=2

[10] [20, 30, 40] → 10 | 90 → Take 90 units of time to finish it

[10, 20] [30, 40] → 30 | 70 → Take 70 units of time to finish it

[10, 20, 30] [40] → 40 | 60 → Take 60 units of time to finish it

min(90, 70, 60) → 60 → answer.

★ Split array Largest-sum → Split the array into K subarrays such that the maximum subarray sum is minimum.

| 7 | 2 | 5 | 10 | 8 | K=2

[7], [2, 5, 10, 8] → 7 | 25 → max(7, 5) = 5

[7, 2], [5, 10, 8] → 9 | 23 → max(7, 23) = 23

[7, 2, 5] [10, 8] → 14 | 18 → max(14, 18) = 18 → largest sum of subarray which is minimized

[7, 2, 5, 10] [8] → 8 | 24 → max(8, 24) = 24

min(25, 23, 18, 24) => 18

Same code will work on both the problems

TC - N log(max - sum)

SC - O(1)

```java
PartXV.java
public static int allocateBooks(int[] books , int noOfStudent){  1 us
    int start = getMax(books);
    int end = Arrays.stream(books).sum();

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

        if (canWeAllocate(books,mid) > noOfStudent) start = mid + 1;
        else end = mid - 1 ;
    }
    return start;
}

private static int canWeAllocate(int[] books , int pages){  1 usage  n
    int countStudent = 1;
    int noOfPages = 0;

    for (int e : books){
        if (noOfPages + e <= pages) noOfPages += e;
        else {
            countStudent++;
            noOfPages = e;
        }
    }
    return countStudent;
}
```

→ Minimise Maximum Distance between two station

↳ place K new gas station, such that we minimise the maximum distance between them.

[Hardest Question]

Ex-1 | 1 | 2 | 3 | 4 | 5 | K = 4

① [1, 2, 3, 4, 5, 6, 7, 8, 9] → max(1,...,1) = 1

② [1, 1.25, 1.5, 1.75, 2, 3, 4, 4.5, 5] → max(0.5,...,1) = 1

③ [1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5] → max(0.5,...,0.5) = 0.5

↳ answer, we have minimized the maximum.

Ex-2 | 1 | 7 | K = 2

① [1, 7, 8, 9] max(6,1,1) = 6

② [1, 2, 4, 7] max(1,2,3) = 3

↳ so we have reduced the maximum by not adding the gas station at right & neither at left

↓

This clears that, to find minimized maximum we have to insert the gas station somewhere in between.

☆ | 1 | 7 | K = 6

↳ so if we insert K gas station in between.

| 1 | G1 | G2 | 7 |

1 → G1, G1 → G2, G2 → 7 } K+1 sector's

↓

we would have to minimise K+1 sector's like from

Example

③ [1, 7] K = 2

6

↳ so to divide it optimally such that two consecutive distance is minimised

↓

$$dist = \frac{(upper - lower)}{K+1}$$

$$2 = \left(\frac{7-1}{2+1}\right)$$

↳ optimal sector distance.

[1, 3, 5, 7] K = 2

6

So, the minimum between two consecutive is minimum.

Example - 3

| 1 | 13 | 17 | 23 | K = 5

12, 4, 6

max(12, 4, 6) → 12 is maximum

↓

so we should minimise this first.

For, K = 1 →

1st gas station

$\frac{(13-1)}{1+1} = 6$

| 1 | 7 | 13 | 17 | 23 |

6, 6, 4, 6

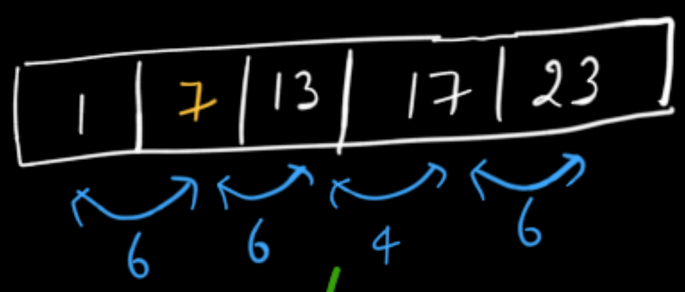
↳ multiple same max so we can place gas station anywhere

0	1	2
1		

↳ in org. array we have three sector's

- so we have placed 1 gas station in {1,13} so we will move sector 0 as 1

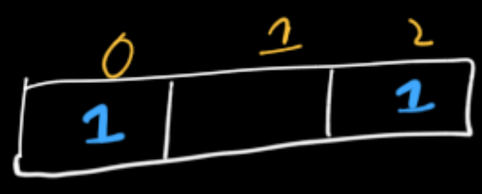
For, K=2



2nd gas station

$\frac{(23-17)}{1+1} : 3$

we are placing 2nd gas station not K gas station.



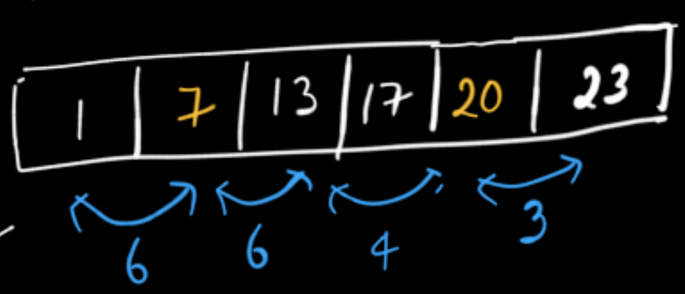
↳ we placed 1 in sector 2.

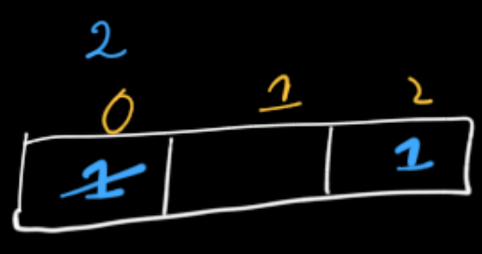
For K=3,

↳ so instead of placing 1 new gas station at rear, we already know that, sector 0, may already be divided

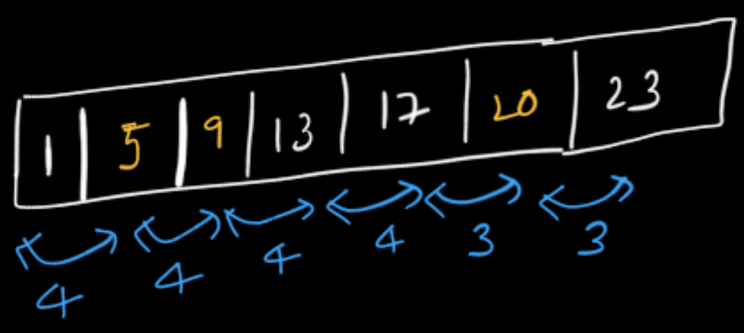
↳ for this K=3, we will discard the divided part

Instead of going over 6 we will ignore initial sector division. ← start from {1,13}.





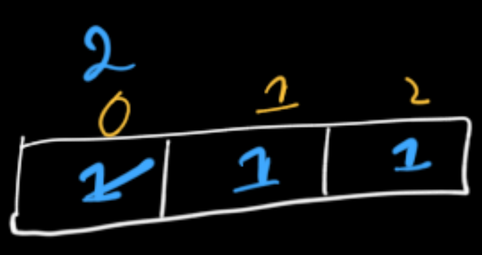
↳ we placed 1 in sector 2.



We discarded initial K=1 sector division

$\frac{(13-1)}{2+1} = 4$

So, K=2

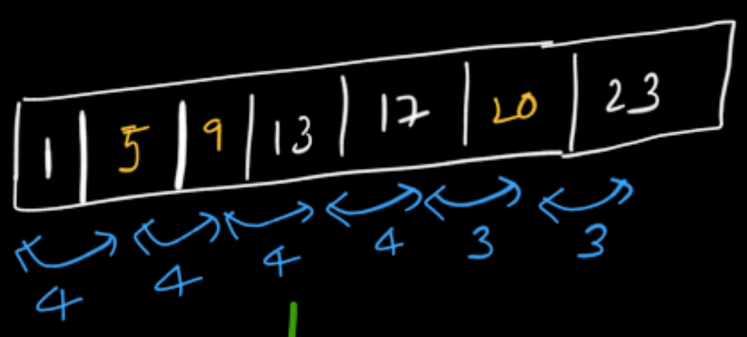


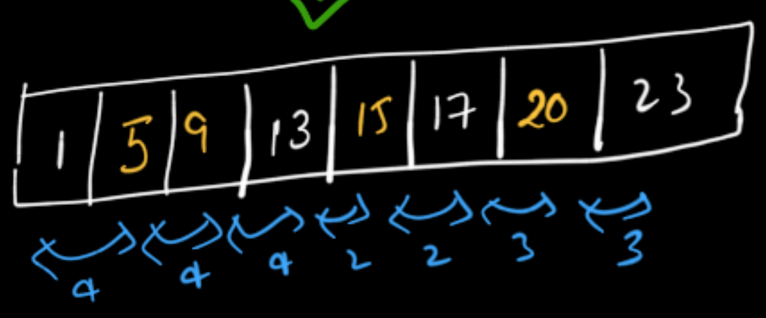
↳ we placed 1 in sector 1

↓

as we have minimised the first sector of original array

For K=4

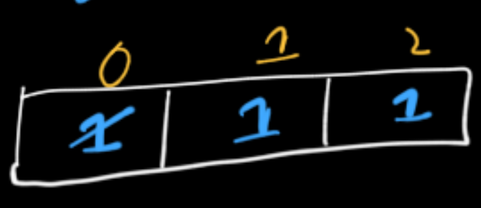




So, $\frac{(17-13)}{1+1} = 2$

For K=5

↳ again we will do same, as we have done for K=3,

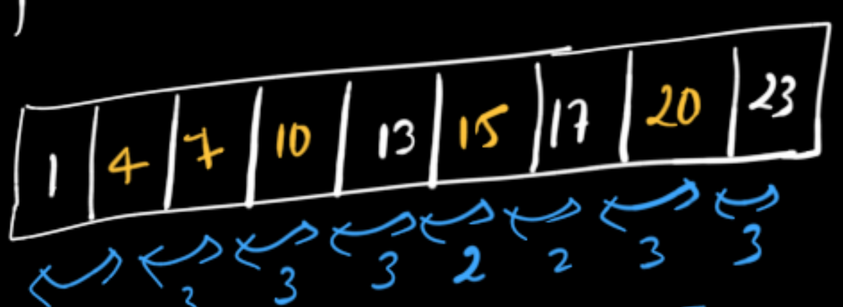


↳ we placed 1 in sector 1

↓

as we have minimised the first sector of original array

$= \left(\frac{13-1}{3+1}\right) = 3$



for K=1
K=3
K=5

max(3, 2, 3) = 3

answer

→ Linear search Approach

```java
PartXVII.java
public static double MinimiseMaxDistance(int[] arr, int K){  no usages
    int[] howMany = new int[arr.length - 1];
    Arrays.fill(howMany, val: 0);

    for (int i = 1; i <= K; i++) {
        double maxSection = -1;
        int maxIndex = -1;

        for (int j = 0; j < arr.length - 1; j++) {
            double diff = arr[j + 1] - arr[j];
            double sectionLength = diff / (double) (howMany[j] + 1);

            if (sectionLength > maxSection){
                maxSection = sectionLength;
                maxIndex = j;
            }

        }
        howMany[maxIndex]++;
    }

    double maxAns = -1;
    for (int i = 0; i < arr.length - 1; i++) {
        double diff = arr[i + 1] - arr[i];
        double sectionLength = diff / (double) (howMany[i] + 1);
        maxAns = Math.max(maxAns,sectionLength);
    }
    return maxAns;
}
```

Tc: $O(n \times K)$

Sc: $O(n-1)$

→ Priority Queue

```java
PartXVII.java
public static double MinimiseMaxDistanceOpt(int[] arr, int K) {  no usages  new *
    int[] howMany = new int[arr.length - 1];

    PriorityQueue<double[]> pq = new PriorityQueue<>(
            ( double[] a,  double[] b) -> Double.compare(b[0], a[0])
    );

    for (int i = 0; i < arr.length - 1; i++) {
        double diff = arr[i + 1] - arr[i];
        pq.add(new double[]{diff, i});
    }

    for (int i = 1; i <= K; i++) {
        double[] top = pq.poll();
        int idx = (int) top[1];
        howMany[idx]++;

        double segmentLength = (double) (arr[idx + 1] - arr[idx]) / (howMany[idx] + 1);
        pq.add(new double[]{segmentLength, idx});
    }

    return pq.poll()[0];
}
```

Tc: $O(n \log n + K \log n)$

Sc: $O(n-1)$

→ Median of two sorted Arrays

| 1 | 3 | 4 | 7 | 10 | 12 |

| 2 | 3 | 6 | 15 |

→ mergesort - solution

0	1	2	3	4	5	6	7	8	9
1	2	3	3	4	6	7	10	12	15

→ sorted array.

For - even length → There must be two median

① At $\frac{n}{2}$ ② At $\frac{n}{2} + 1$

So, median is $\frac{10}{2}$, $\frac{10}{2}+1$ => {5, 6} → position

↓

{4, 5} → index

So, answer is $\left(\frac{4+5}{2}\right) = 5$

For odd length → only 1 media

↳ median = $\left(\frac{n}{2} + 1\right)^{th}$

0	1	2	3	4	5	6
1	2	4	6	10	12	18

→ $\left(\frac{7}{2}+1\right)$ => 4^{th} → position

(4 - 1) → index.

Tc: O(n+m)

Sc: O(first + second)

is there any way we can reduce the space

PartXVIII.java

```java
public static double median(int[] first, int[] second) {
    int[] sorted = new int[first.length + second.length];

    int i = 0;
    int j = 0;
    int k = 0;

    while (i < first.length && j < second.length) {
        if (first[i] <= second[j]) {
            sorted[k] = first[i];
            i++;
        } else {
            sorted[k] = second[j];
            j++;
        }
        k++;
    }

    while (i < first.length) {
        sorted[k] = first[i];
        i++;
        k++;
    }

    while (j < second.length) {
        sorted[k] = second[j];
        j++;
        k++;
    }
    int len = sorted.length;
    return len % 2 == 0
            ? (sorted[len / 2 - 1] + sorted[len / 2]) / 2.0
            : sorted[len / 2];
}
```

→ optimizing the space-complexity

f = | 1 | 3 | 4 | 7 | 10 | 12 | (0 1 2 3 4 5)

s = | 2 | 3 | 6 | 15 | (0 1 2 3)

- we know the length of the array i.e (f.length + s.length)
  - For even length → $\{\frac{n}{2}, \frac{n}{2}+1\}$
    - we just need to find two element at these two above index.
  - For odd length → $\{\frac{n}{2}\}$

we can do this without using extra space

f = | 1 | 3 | 4 | 7 | 10 | 12 | (0 1 2 3 4 5) with i, i, i, i

s = | 2 | 3 | 6 | 15 | (0 1 2 3) with j, j, j, j

$i_1 = 4$
$i_2 = 5$ ↓ goal

K=0	· f[i] < s[j] → i++, K++; (1)
K=1	· f[i] > s[j] → j++, K++; (2)
K=2	· f[i] ≥ s[j] → i++; K++; (3)
K=3	· f[i] < s[j] → j++; K++; (4)
K=4	· f[i] < s[j] → i++; K++ (5)
K=5	· f[i] > s[j] → j++; K++; (6)

There maybe a case when either of the two length maybe greater than each other.

Index1 = $\frac{n}{2} - 1 = (5-1) = 4$

Index2 = Index1 + 1

e1 = -1

e2 = -1

Here K == Index1 → $e_1 = f[i]$;

Here K == Index2 → $e_1 = s[i]$

↓ Found our two element,

median = $\left(\frac{e_1 + e_2}{2.0}\right)$

Tc: O(n+m)

Sc: O(1)

PartXVIII.java

```java
public static double medianSpaceOpt(int[] first, int[] second) {
    int len = first.length + second.length;

    int index1 = len / 2 - 1;
    int index2 = index1 + 1;

    int e1 = -1;
    int e2 = -1;

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

    while (i < first.length && j < second.length) {
        int val;
        if (first[i] <= second[j]) {
            val = first[i++];
        } else {
            val = second[j++];
        }
        if (k == index1) e1 = val;
        if (k == index2) e2 = val;
        k++;
    }

    while (i < first.length) {
        int val = first[i++];
        if (k == index1) e1 = val;
        if (k == index2) e2 = val;
        k++;
    }

    while (j < second.length) {
        int val = second[j++];
        if (k == index1) e1 = val;
        if (k == index2) e2 = val;
        k++;
    }

    return len % 2 == 0 ? (e1 + e2) / 2.0 : e2;
}
```

→ Binary search Approach

t = | 1 | 3 | 4 | 7 | 10 | 12 |

s = | 2 | 3 | 6 | 15 |

1	2	3	3	4	6	7	10	12	15

5 element | 5 element

we start picking element from both array's to form the above sorted Arrays.

	t	s
No of element	2	4
take x element from (t) & y element from (s)	2	3
	3	2

1 2 3 6 15 | 3 4 7 10 12 — Sorted? ✗

1 3 | 2 3 6 15 | 4 7 10 12 — ✗

1 3 4 | 2 3 6 | 7 10 12 15 — ✓

(1, 2, 3, 3, 4, 6, 7, 10, 11, 15)
↳ sorted.

How do we know if it is valid symmetry?

1 3 4 l_1 | r_1 7 10 12
2 3 l_2 | r_2 6 15

↳ left half must be less than the right half.

↳ so, if $arr[l_1] <= arr[r_2]$ as $arr[l_1] \leq arr[r_1]$
↳ valid symmetry.

How do we find median then?

1	2	3	3	4	6	7	10	12	15

max(left) = 4 ↓ min(right) = 6

1 3 4 l_1 | r_1 7 10 12
2 3 l_2 | r_2 6 15

$max(l_1, l_2)$ $min(r_1, r_2)$ → median = $\frac{max(l_1, l_2) + min(r_1, r_2)}{2.0}$

→ Binary search condition

t = | 1 | 3 | 4 | 7 | 10 | 12 |

s = | 2 | 3 | 6 | 15 |

$l_1 > r_1$ → cnd = m − 1

1 3 4 7 l_1 | r_1 3 6 5
2 l_2 | r_2 12

$l_2 > r_1$ → start = m + 1

1 3 l_1 | r_1 4 7 10 12
2 3 6 l_2 | r_2 15

	0	1	2	3
f =	7	12	14	15

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

↳ Apply BS on smallest array, will reduce the time comp.

start = 0 → minimum number of element we can take

end = 4 → max number of element we can take

mid1 = (low + high) >> 1 → 2

mid2 = no of element we want − mid1 → 3 (5)

① $l_1 > r_2$ → end = mid − 1

↳ s = 0, e = 1, m_1 = 0, m_2 = 5

↳

	7	12	l_1	r_1	4	9	11
	1	2	3	l_2	r_2	14	15

mid1 → ↗ ; mid2 → ↘

INT-MIN				l_1	r_1 (m_1)		
	7	12	14	15			
1	2	3	4	9	11 (m_2)		
				↓ l_2	↑ r_2		

② $l_2 > r_1$ → s = m + 1

↳ s = 1, e = 1, m_1 = 1

m_2 = 5 − 1 = 4

↳

				7	l_1	r_1 9	11
1	2	3	4	l_2	r_2 12	14	15

③ Both, $l_1 < r_2$ && $l_2 < r_1$ → $\frac{\max(l_1, l_2) + \min(r_1, r_2)}{2.0} = \frac{16}{2}^{8.0}$

④ when length is odd,

↳ median = max(l_1, l_2)

PartXX.java

```java
public static double findMedian(int[] first , int[] second){  3 usages  new *
    int first_len = first.length;
    int sec_len = second.length;
    if (first_len > sec_len) return findMedian(second,first);

    int start = 0;
    int end = first_len;
    int left = (first_len + sec_len + 1) / 2;

    while (start <= end){
        int m1 = start + (end - start)/2;
        int m2 = left - m1;

        int l1 = (m1 > 0) ? first[m1 - 1] : Integer.MIN_VALUE;
        int l2 = (m2 > 0) ? second[m2 - 1] : Integer.MIN_VALUE;
        int r1 = (m1 < first_len) ? first[m1] : Integer.MAX_VALUE;
        int r2 = (m2 < sec_len) ? second[m2] : Integer.MAX_VALUE;

        if (l1 <= r2 && l2 <= r1){
            if ((first_len + sec_len) % 2 == 1) return Math.max(l1,l2);
            else return (double) (Math.max(l1,l2)  + Math.min(r1,r2)) / 2.0;
        } else if (l1 > r2) end = m1 - 1;
        else start = m1 + 1;
    }
    return -1.0;
}
```

TC : O(log(min(n1,n2)))

SC : O(1)

→ Find Kth Element in two sorted Array

start = Math.max(0, k - sec_len);

↳ we cannot pick more than m elements from second array. So the minimum no. of elements we must take from first array is

↳ Math.max(0, k - m)

end = Math.min(k, n)

↳ We cannot pick more than n elements from first array, so the maximum number we can take from it is

↳ Math.min(k, n);

(n) f = [2, 3, 6, 7] (m) s = [1, 4, 8, 10] k = 7

k - m = 7 - 4 = 3 → must take 3 element from f

min(k, n) = min(7, 4) = 4

↳ can take at most 4 from f

PartXX.java

```java
public static double kthElement(int[] first, int[] second, int k) {  1 usa
    int first_len = first.length;
    int sec_len = second.length;
    if (first_len > sec_len) return kthElement(second, first, k);

    if (k == 1) {
        return Math.min(first.length > 0 ? first[0] : Integer.MAX_VALUE,
                second.length > 0 ? second[0] : Integer.MAX_VALUE);
    }

    int start = Math.max(0, k - sec_len);
    int end = Math.min(k, first_len);

    int left = (first_len + sec_len + 1) / 2;

    while (start <= end) {
        int m1 = start + (end - start) / 2;
        int m2 = left - m1;

        int l1 = (m1 > 0) ? first[m1 - 1] : Integer.MIN_VALUE;
        int l2 = (m2 > 0) ? second[m2 - 1] : Integer.MIN_VALUE;
        int r1 = (m1 < first_len) ? first[m1] : Integer.MAX_VALUE;
        int r2 = (m2 < sec_len) ? second[m2] : Integer.MAX_VALUE;

        if (l1 <= r2 && l2 <= r1) return Math.max(l1, l2);
        else if (l1 > r2) end = m1 - 1;
        else start = m1 + 1;
    }
    return -1.0;
}
```

→ Find Row with maximum 1's

↳ Return the row which have maximum number of 1's, if there are multiple return the smallest index.

	0	1	2	3
0	0	0	1	1
1	0	0	0	0
2	0	1	1	1
3	0	1	1	1

```
fn(mat[][]) {
    int count = 0;
    int index = -1;
    for(int i=0; i<mat.length; i++){
        int curr_count = 0;
        for(int j=0; j<mat[0].length; j++){
            if(mat[i][j] == 1){
                curr_count++;
            }
        }
        if(curr_count > count){
            count = curr_count;
            index = i;
        }
    }
    return index;
```

Tc: $O(n \times m)$

Sc: $O(1)$

How do we reduce it?

↳ Each row is sorted, it we can find no. of 1's in each row in less than O(n) we can reduce the time complexity.

0	0	1	1	1	1

if we can find first occurrence of 1's then we can find, Total no of 1's in that row

So,

$$\text{no of 1's} = \text{len(row)} - \text{first occurrence(arr, 1)}$$

Binary search

$$Tc: O(n \times \log n)$$

```
PartXXI.java
public static int[] rowAndMaximumOnes(int[][] mat) {  2 usa
    int count = 0;
    int index = -1;

    for (int i = 0; i < mat.length; i++) {
        int[] e = mat[i];
        int pos = firstOccurrence(e, target: 1);
        int count_max = (pos == -1) ? 0 : e.length - pos;

        if (count_max > count) {
            count = count_max;
            index = i;
        }
    }
    return new int[]{index , count};
}

public static int firstOccurrence(int[] arr, int target) {
    int start = 0;
    int end = arr.length - 1;
    int first = -1;

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

        if (arr[mid] == target) {
            first = mid;
            end = mid - 1;
        } else if (arr[mid] < target) start = mid + 1;
        else end = mid - 1;
    }
    return first;
}
```

→ Search in 2D Matrix - 1

↳ Apply binary search on each row of the matrix

★ Little - optimization

↳ Apply binary search only if target lies in that range

i.e arr[first] ≤ target ≤ arr[last]

↳ Tc: O(n log m)

1	3	5	7
10	11	16	20
23	30	34	60

→ Flattend Binary search concept

↳ Treating a 2D matrix as 1D matrix, so we can apply binary search directly.

Flattend →

0	1	2	3	4	5	6	7	8	9	10	11
1	3	5	7	10	11	16	20	23	30	34	60

How do we convert 1D array index to 2D coordinate ?

↳ For matrix with n rows & m column

ROW = index / m

col = index % m

index = row * col + col

↳ from 2D → 1D index

1D → 2D index

- Work only on row-wise sorted matrix.
- No variable column size i.e

1	2	3	4
5	6	7	
8	9		

↳ will not work on this

```
PartXXII.java
public static boolean flattenedBS(int[][] mat , int target){

    int start = 0;
    int end = mat.length * mat[0].length - 1;


    while (start <= end){
        int mid = start + (end - start) / 2;
        int row = mid / mat[0].length;
        int col = mid % mat[0].length;

        if (mat[row][col] == target) return true;
        else if (mat[row][col] < target) start = mid + 1;
        else end = mid - 1;
    }

    return false;
}
```

Row-wise Binary search

① Use when each row is individually sorted

↳

1	3	5
2	4	7
6	8	9

② Tc: O(n log m)

Flattened Binary search

① Whole matrix is globally sorted

↳

1	3	5
6	7	9
10	12	15

② Tc: O(log(m * n))

→ Search in 2D Matrix II

↳ matrix is sorted both row & column wise

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

down

left

→ Observation

↳ • Going left gives smaller values
• Going down gives larger values

→ At every step

- if curr-val == target → true
- if curr-val > target → move left
- if curr-val < target → move right

→ Start from top-right element i.e mat[0][m-1]. But why ?

↳ • It gives a binary decision path at every step i.e
↳ • Left if target is smaller
• Down if target is larger.

↳ Both direction move us closer to or eliminate parts of the matrix safely.

Only corner where we can make both these decision

Tc: O(n+m)

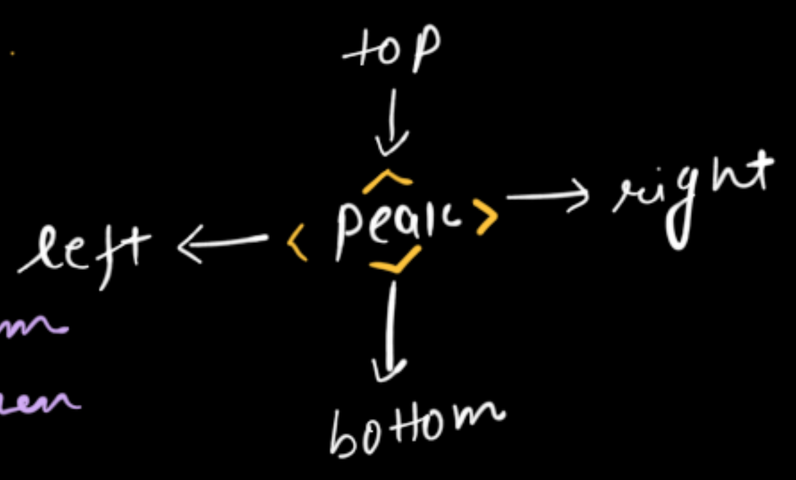
Sc: O(1)

PartXXIII.java

```java
public static int[] search2D(int[][] mat, int target) {  1 usage
    // for row and col wise sorted matrix

    int row = 0;
    int col = mat[0].length - 1;

    while (row < mat.length && col >= 0) {

        if (mat[row][col] == target) return new int[]{row, col};
        else if (mat[row][col] > target) col--;
        else row++;
    }

    return new int[]{-1, -1};
}
```

→ Find Peak Element II

↳ given a mxn matrix, where no two adjacent cells are equal, find any peak element mat[i][j].



	0	1	2	3	4
0	-1	-1	-1	-1	-1
1	-1	10	20	15	-1
2	-1	21	30	14	-1
3	-1	7	16	32	-1
4	-1	-1	-1	-1	-1

- if we can find maximum in a particular column & then check its left & right, we can find our peak.

Ex: Col-2 → Find max → 30

↳ . check it left < 30 > right i.e 21 < 30 > 14 ✓

↳ return {2,2}

Steps

- Find the max element in the column, return its index.
- Calculate left & right, if the boundary condition doesn't satisfy take -1.
- Check it,

left < element at maxIndex > right

↳ ✓, peak element found.

- if left is bigger

↳ end = mid - 1

- if right is bigger

↳ start = mid + 1

Tc: O(n log m)

Sc: O(1)

PartXXIV.java

```java
public static int[] findPeak2D(int[][] matrix) {  no usages  new *

    int row = matrix[0].length;
    int start = 0;
    int end = row - 1;

    while (start <= end) {
        int mid = start + (end - start) / 2;
        int maxIndex = findMaxInCol(matrix, mid);
        int left = mid - 1 >= 0 ? matrix[maxIndex][mid - 1] : -1;
        int right = mid + 1 < row ? matrix[maxIndex][mid + 1] : -1;

        int value = matrix[maxIndex][mid];
        if (value > left && value > right){
            return new int[]{maxIndex,mid};
        }
        else if (value < left) end = mid - 1;
        else start = mid + 1;
    }
    return new int[]{-1,-1};
}

private static int findMaxInCol(int[][] mat, int col) {  1 usage  new

    int index = -1;
    int max = Integer.MIN_VALUE;

    for (int i = 0; i < mat.length; i++) {
        if (mat[i][col] > max) {
            max = mat[i][col];
            index = i;
        }
    }
    return index;
}
```

→ Median of Row wise sorted matrix

1	5	7	9	11
2	3	4	5	10
9	10	12	14	16

→ Flatten out & sort it

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
[1, 2, 3, 4, 5, 5, 7, 9, 9, 10, 10, 11, 11, 12, 14, 16]

$median = \frac{m \times n}{2} = \frac{5 \times 3}{2} = 7 \approx 9$

TC: $n \times m + n \times m \log(m \times n)$

SC: $O(n \times m)$

→ Optimization

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

1 2 3 4 5 5 7 9 9 10 10 11 12 14 16

1 2 3 4 6 6 7 9 9 11 11 12 13 14 15

$Req = \frac{m \times n}{2} = \frac{5 \times 3}{2} = \frac{15}{2} = 7$

No. of element less than or equal to each arr[i]

① Find min & max → search range
↳ median will lie in between

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ← max

min → 1 2 3 4 6 6 7 7 9 9 11 11 12 13 14 15

s, m, e

↳ Now, mid = 8 & total count of element less than or equal to 8 is 7

↳ i.e 7 < 8 → too few ← eliminate left half

s = m + 1

8 9 10 11 12 13 14 15 16
7 9 9 11 11 12 13 14 15

→ mid = 12 > 8 → enough → eliminate right half

e = m - 1

8 9 10 11 12 13 14 15 16
7 9 9 11 11 12 13 14 15

→ mid = 9 > 8 → enough → eliminate right half

e = m - 1

8 9 10 11 12 13 14 15 16
7 9 9 11 11 12 13 14 15

→ mid = 9 > 8 → enough → eliminate right half

e = m - 1

8 9 10 11 12 13 14 15 16
7 9 9 11 11 12 13 14 15

→ s > e

↳ Start will be our answer

How do we count no of element which is less than or Equal to mid?

↳ Apply upperbound on each row & with target as mid.

Tc: $\log(max - min) * n \log_2 m + n$

PartXXV.java

```java
public static int findMedian(int[][] mat) {  1 usage  new *

    int start = getMaxMin(mat)[0];
    int end = getMaxMin(mat)[1];

    int req = (mat.length * mat[0].length) / 2;

    while (start <= end) {
        int mid = start + (end - start) / 2;
        int c = countLessOrEqual(mat, mid);

        if (c <= req) start = mid + 1;
        else end = mid - 1;
    }
    return start;
}

private static int countLessOrEqual(int[][] mat, int value) {
    int count = 0;
    for (int[] e : mat) {
        count += upperBound(e, value);
    }
    return count;
}

private static int[] getMaxMin(int[][] mat) {  2 usages  new *
    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;

    for (int[] e : mat) {
        min = Math.min(min, e[0]);
        max = Math.max(max, e[mat[0].length - 1]);
    }
    return new int[]{min, max};
}

public static int upperBound(int[] arr, int target) {  1 usage
    int start = 0;
    int end = arr.length - 1;
    int ans = arr.length;

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

        if (arr[mid] > target) {
            ans = mid;
            end = mid - 1;
        } else start = mid + 1;
    }

    return ans;
}
```

↳ $O(n \log_2 m)$

$O(n)$

$O(\log_2 m)$