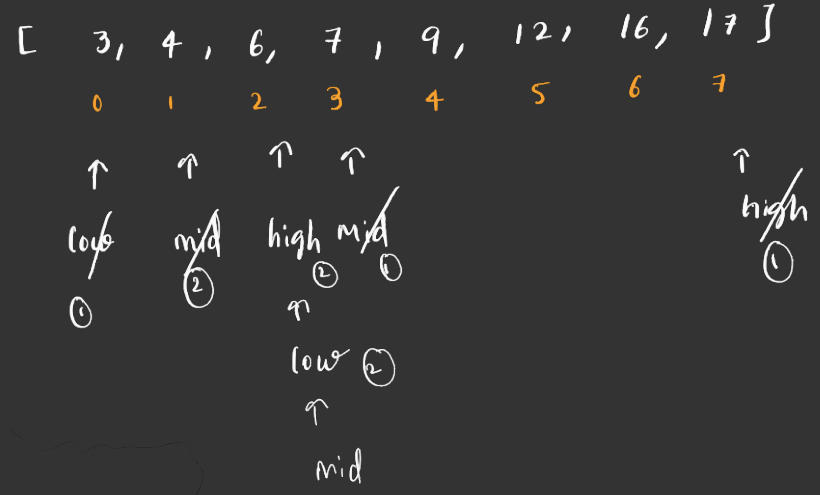
Binary Search.

- Real life Application : Dictionary
- Works by reducing the search space.
- Works on sorted work space
- uses divide & conquer Approach.
-



n = 8
target = 6.

low = 0
high = n - 1.
mid = (low + high) / 2
high = mid - 1
low = mid + 1

Compute mid

[3, 4, 6, 7, 9, 12, 16, 17]

0 1 2 3 4 5 6 7

n = 8

target = 13.

low / mid / low / mid / low / high

high / mid

$0+7/2 = 3.5 = 3$

$4+7 = 11/2 = 5$

$5+7 = 12/2 = 6$

as high < start

we need to stop.

or

start <= high

we need to compute mid and perform checks.

① Iterative.

```
f( arr, n, target ) :
    low = 0
    high = n
    while ( low <= high ) :
        mid = (low + high) / 2
        if arr[mid] == target:
            return mid
        elif arr[mid] > target :
            high = mid - 1
        elif arr[mid] < target :
            low = mid + 1.
```

② recursion

```
f( arr, low, high, target ) :
    if low > high :
        return -1.
    mid = low + high/2
    if arr[mid] == target
        return mid
    elif arr[mid] > target :
        f( arr, start, mid - 1, target)
    elif arr[mid] < target :
        f( arr, mid + 1, high, target)
```

32

① ↳ 16

② ↳ 8

③ ↳ 4

④ ↳ 2

⑤ ↳ 1

∴ 5 or 6 steps.

[32 elements.]

$\Rightarrow 32 = 2^5$

$\log_2(32) = 5$

$O(\log n)$

↳ Complexity.

Overflow Case.

[0 ..... int max]

↑ low, ↑ high

$$mid = \frac{low + high}{2}$$

imagine if we are at last position or at middle any where after 0.

So, we store it as

$$mid = low + \frac{high - low}{2}$$

$$\frac{intmax + intmax}{2} = \underline{\underline{2(intmax)}}$$

↳ which can't be stored in normal int we may need long / float to store.

intmax + or ∈ 0

mid. = intmax ✓

2. Lower Bound

↳ Smallest index such that arr[index] >= x

arr = [3, 5, 8, 15, 19, 19] n = 6

index: 0 1 2 3 4 5

x = 8 ⇒ Smallest index where arr[index] >= 8.

↳ 2.

x = 9 ⇒ 15 ⇒ lb = 3

x = 16 ⇒ lb = 4

x = 20 ⇒ lb = 6 or n.

x = 19

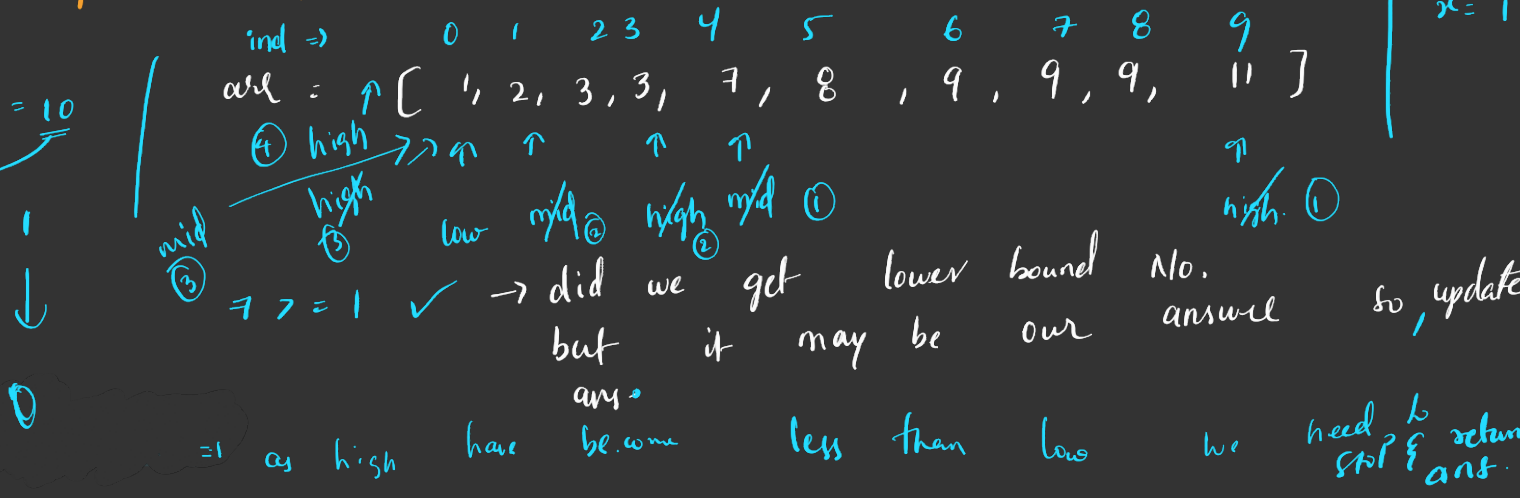
↳ We need to find the first index.

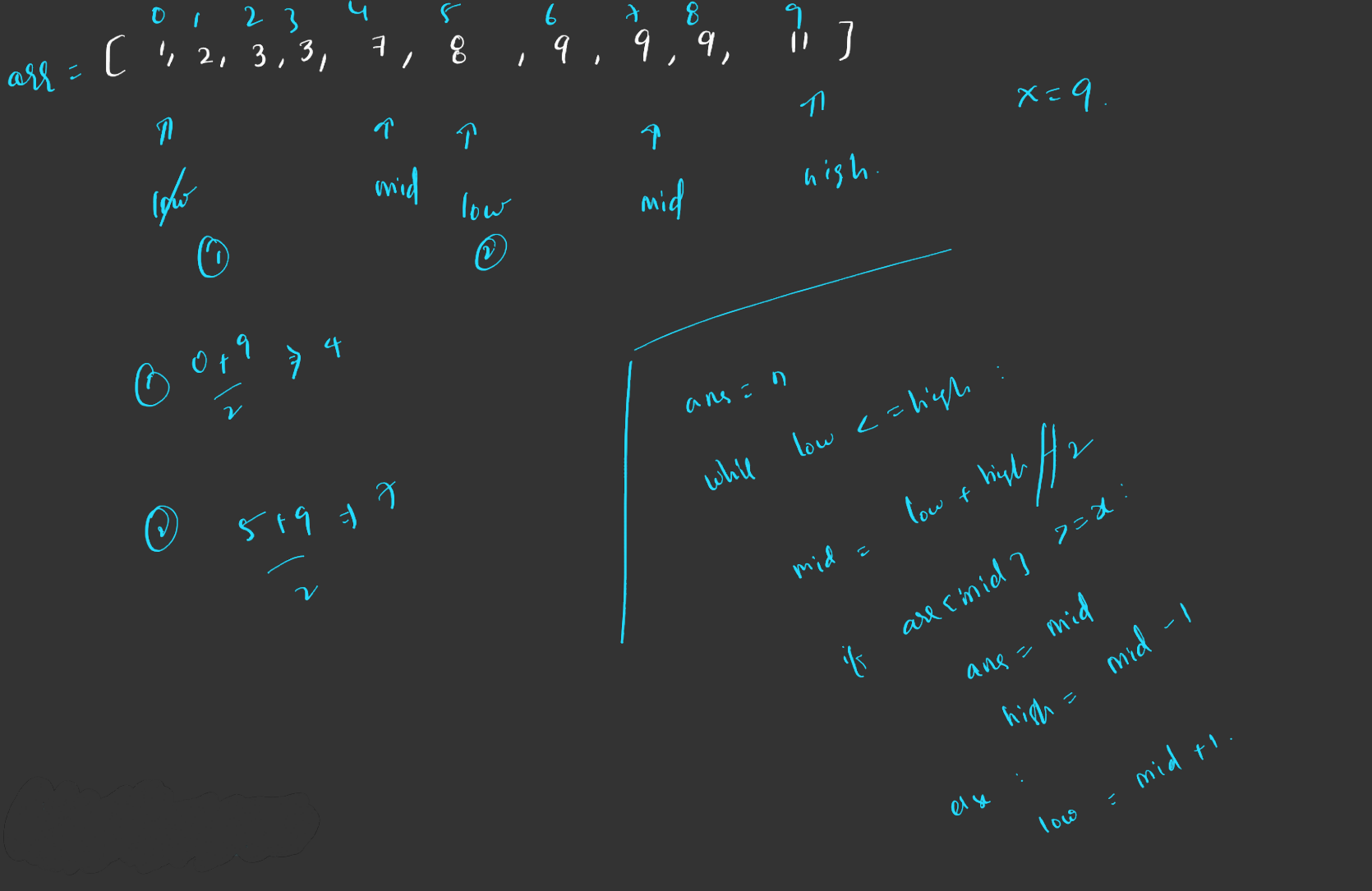
① linear search.

② Binary search.

As we know that in the earlier case ie, at x = 20, the answer is 6 which is the size of array which can't be found..

So by handling this corner case we need to move further.




0 1 2 3 4 5 6 7 8 9
arr = [1, 2, 3, 3, 7, 8, 9, 9, 9, 11]
x = 9.
low
mid
low
mid
high.
① 0+9/2 ⇒ 4
② 5+9/2 ⇒ 7
ans = n
while low <= high :
mid = low + high // 2
if arr[mid] >= x :
ans = mid
high = mid - 1
else :
low = mid + 1.

3. Upper bound

↳ smallest index such that

arr[index] > x.

Similar to LB

we just need to change

Condition.

arr[mid] > x. ✓

4. Search insert position

Problem Statement

Suggest Edit

no dups.

You are given a sorted array *'arr'* of distinct values and a target value *'m'*.

You need to search for the index of the target value in the array.

Note:

```
If the value is present in the array, return its index.
If the value is absent, determine the index where it would be
inserted in the array while maintaining the sorted order.
```

Example:

```
Input:  arr = [1, 2, 4, 7],  m = 6

Output: 3

Explanation: If the given array 'arr' is: [1, 2, 4, 7] and m =
6. We insert m = 6 in the array and get 'arr' as: [1, 2, 4, 6,
7]. The position of 6 is 3 (according to 0-based indexing)
```

0 1 2 3 n=4, m=6

[1, 2, 4, 7]

[1, 2, 4, 6, 7]

0 1 2 3 4

output.

if x=2

arr[mid] > x

[1, 2, 4, 7]

=>> it's same as finding lower bound..

- Floor & Ceil in Sorted array.

!

largest num in array <= x

↳ Just Conditions Changes

Smallest num in array >= x.

arr[mid] >= x

or index

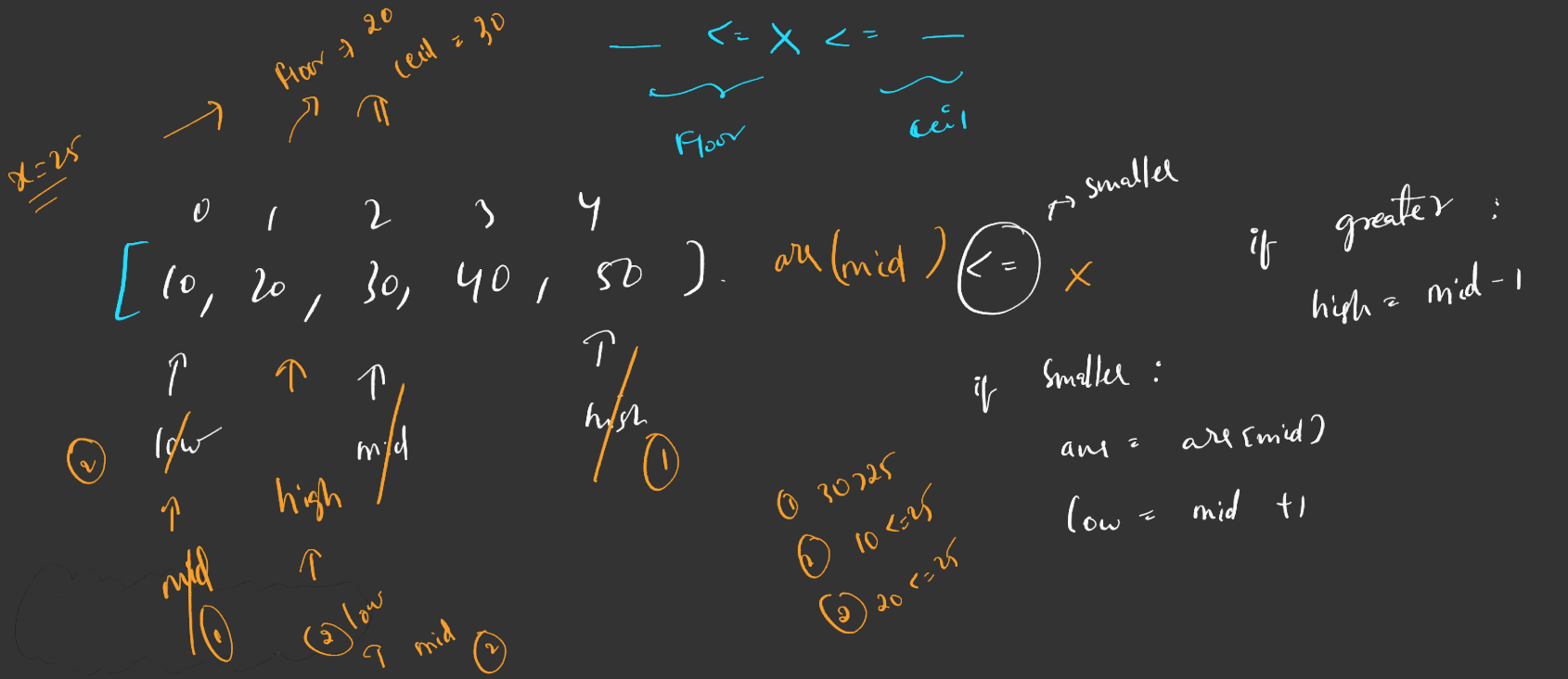
↳ lower bound implementation

We need to return -1 if not found.

that's it.

Floor → — X
↳ This could be if present

ceil → X —
↳ This will be ans if x isn't present
Just greater than x.



- Find the First and last Occurance of a given number in a Sorted array.

	0	1	2	3	4	5	6	7
arr = [	2,	4,	6,	8,	8,	8,	11,	13]

LB ↑ (index 3)

UB ↑ (index 6)

lower bound ⇒ a[index] >= x

upper bound ⇒ a[index] > x

x = 8
o/p = 3,5

x = 10
o/p = -1,-1

x = 11
o/p = 6,6

So,

First = LB

last = UB - 1

But what if element was not present ??

① as w.r.t LB returns n if element x is greater than all elements.

else:

② it will return the next Element which is greater than that.

So we need to check for both of these conditions.

if lb(arr, x) == n or arr[lb] != x

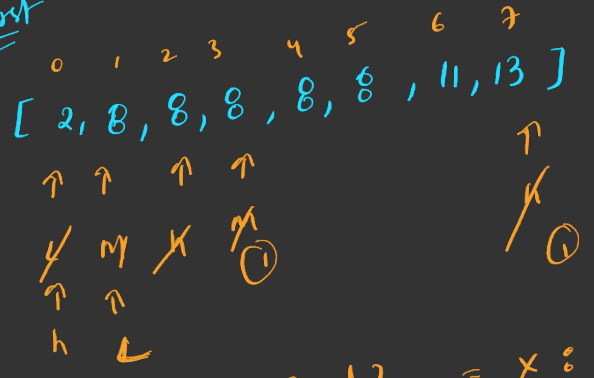
(1) (2)

Proceed further..

Standard Binary Search:

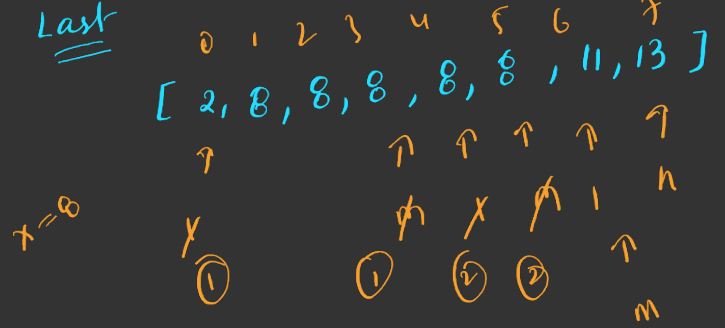
ans = -1, 3, 1

First




```
if arr[mid] == x:
    ans = mid
    high = mid - 1
elif arr[mid] < x
    low = mid + 1
else:
    high = mid - 1
```

Last



ans = -1, 3, 5,

```
if arr[m] == x:
    ans = m
    l = m+1
elif arr[m] > x:
    h = m-1
else:
    l = m+1
```

Search in Rotated Sorted Array [unique Elements Array].

↳ We need to identify the Sorted Half
↳ Left / Right

" and eliminate one half

	0	1	2	3	4	5	6	7	8
	[7	8	9	1	2	3	4	5	6]
	↑	↑	↑	↑	↑				↑
	✗	✗	✗	h	✗				✗
		✗	L						

target = 1

7 <= 1 ≤ 8

⇑

1 is blw 7 & 8

↳ False

↳ Eliminate yourself

low = mid + 1

if it was True :

high = mid - 1

① check if left half is sorted.

mid → 7 <= 8

True

check if 1 is blw 7 & 2

False.

So right half is Sorted → check if 1 blw 2 & 6.

if No eliminate right half

else eliminate left half.

```
f(arr, n, target)
    low = 0
    high = n-1
    while low <= high :
        mid = (low + high) / 2
        if arr[mid] == target  return mid.
        // Left Sorted
        if arr[low] <= arr[mid] :
            if arr[low] <= target && target <= arr[mid]
                high = m - 1
            else :
                low = mid + 1
```

```
// Right Sorted.
else
    if arr[mid] <= target && target <= arr[high]:
        low = mid + 1
    else
        high = mid - 1

return -1;
```

- Search in Rotated Sorted Array - II [Duplicates] → return Boolean...

arr = [7 8 1 2 3 3 3 4 5 6] target = 3.

arr = [6 7 1 2 3 4 4 5] ✓

l m h

✗

} we can determine the sorted half's & go ahead

arr = [4 5 1 2 3 3 4 4] ✓

l m h

arr = [3 1 2 3 3 3 3]

↑ ↑ h

l m

} here we cannot determine the sorted part...

The program wont be able to determine, which part is sorted if low, high, mid values are equal.

So the idea is to shrink the array.

ie, if arr[low] == arr[mid] == arr[high].

low += 1

high -= 1 → add a continue to go for next iteration.

	0	1	2	3	4	5	6
[	3	1	2	3	3	3	3]
	l			M		h	

So to the earlier code

we need to add that condition

& shrink:

The shrinking goes on like that

if The elements are

And find the sorted part

of array.

Time Complexity

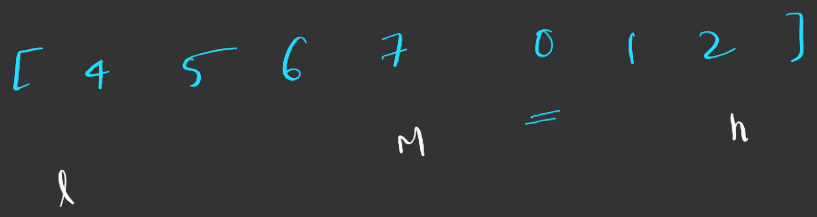
arr = [3 3 1 3 3 3 3]

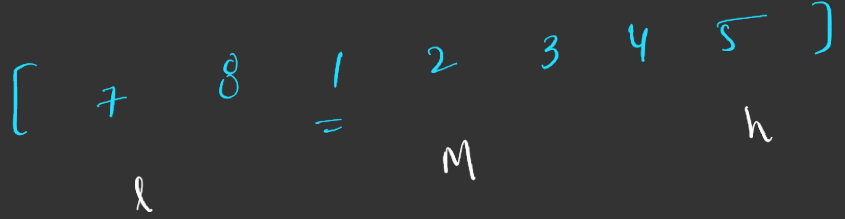
So if we have all duplicates we need to traverse for atmost n/2 times.

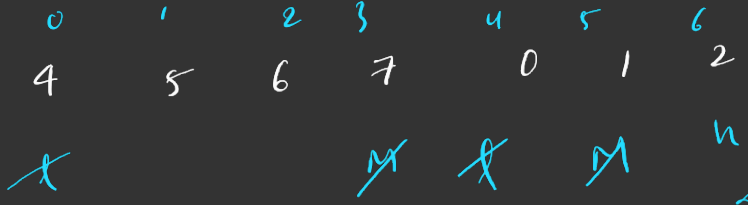
max

So it's $O(n/2)$

- Minimum in Rotated Sorted Array.







ans = ∞ , 4 , 0

① Check if left half is Sorted.

4 <= 7 ✓ ⇒ ans = min(4, ∞)

l = mid + 1

② 0 <= 1 ✓ ⇒

l = mid + 1

③ 1 <= 1 ✓

④ 2 <= 2 ✓

if we have got the Sorted part we can take the minimum element and Eliminate it.

[4 5 1 2 3]

l M h

→ left 4 <= 1 X

ans = min(ans, arr(mid)) .

high = mid - 1

- Find how many times array has been rotated by 1.

```
  0  1  2  3  4
[ 4  5  1  2  3 ]
```

=

We need to find Min Element it's index.

2 times rotated and return