Binary Search:

3, 4, 6, 7, 9, 13, 16, 17 $n = 8$

$t = 6$

$i = 0$

low = 3 high = 17

$mid = \frac{0+7}{2} = 3.5 = 3$

3, 4, 6, 7, 9, 13, 16, 17
↑ ↑ ↑
l m h

$t < a[m]$

$h = m - 1$

$i = 1$. 3, 4, 6
↑ ↑

$l = 3$ $h = 6$

$m = \frac{0+2}{2} = 1$ if $a[m] == t$ return

$t > a[m]$ $l = m + 1$

$i = 2$

~~No need.~~

~~$l = m = 1$~~ $l = 2$, $h = 2$

$m = 2$

return.

Now for the target that is not in arr

[3, 4, 6, 7, 9, 12, 16, 17] n = 8

target = 13

i = 0

m = 3.5 = 3

l = 3+1 = 4 [9]

h = 7 [17]

i = 1

m = $\frac{4+7}{2} = \frac{11}{2}$ = 5.5 = 5

l = [9] = 4

h = [17] = 17

m = [12] = 12 no

target > a[m]

So

l = 5 , h = 7

i = 2

l = 6 , h = 7

m = $\frac{6+7}{2}$ = 6.5 = 6

again target < m

So

l = 6 , h = 5

i = 3 ~~m =~~ return -1 as l >= h

Recursive code

```
f (arr, l, h)
    if l > h :
        return -1

    mid = (l+h)//2
if (mid == t)
    return m
elif    t > a[m]:
    return f (arr, m+1, h)
else
    return f (arr, l, m-1)
```

Lower bound and upper bound.

Smallest index such that arr[idx] >= n

arr = [3, 5, 8, 15, 19]
      0 1 2 3 4

if

n = 5 n = 15 n = 19 n = 20

then

lb_idx = 2 Lb = 3 lb = 4 len(arr) = 5

means the lowest index that can have that element.

-> as the array is sorted so we can apply bs

Example:

arr = [1, 2, 3, 3, 7, 8, 9, 9, 9, 11]

we can always return len(arr) if not found

i = 0

[1, 2, 3, 3, 7, 8, 9, 9, 9, 11] t = 10

m = (0+9)/2 = 4

ans = 7 >= 1 = 7

-> but we will not return it

-> we are going to store it then move onward

as target = 10 so low = mid + 1

So

i = 1 [—, 8, 9, 9, 9, 11]

m = (5+9)/2 = 7

ans = 7

still t > arr[m]

So

i=3

[9, 11]

l=8, h=9

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

ans = 8

target >

So

l = 9

Now

~~Break~~

~~i=3~~

i=3

target < 11

So mid = 9

still ans = 11, ~~9~~ idx

but

l = 9, h = 8 Now

i=4

break, no iteration.

The main point is if
arr[idx] >= target
then update ans.
if element not in arr then
l <= h
condition will click
then and ans will still be
len(arr)
other wise ans will be updated
again and again

UPPER Bound
Smallest index such that arr[idx] > x

arr = [2, 3, 6, 7, 8, 8, 11, 11, 11, 12]
0 1 2 3 4 5 6 7 8 9

target: x = 6	x = 11	x = 12	x = 13	x = 0
return idx = 5	idx = 9	idx = 10	idx = 10	idx = 0

every thing same except arr[idx] > x

floor and ceil

floor → largest no in arr <= x

ceil → Smallest no in arr >= x

arr = [10, 20, 30, 40, 50]

floor

x = 25	x = 30	x = 50
20 (1)	20 (1)	40 (3)

ceil

x = 25	x = 40	x = 50
30 (2)	50 (4)	5

first and last occurance of element in arr

arr = [2, 4, 6, 8, 8, 8, 11, 13]

x = 8	x = 10	x = 11
{3, 5}	{-1, -1}	{6, 6}

linear search code in file.

---

we can do this with lower bound and upper bound approach

arr = [2, 4, 6, 8, 8, 8, 11, 13]

lb = arr[idx] >= x it will give 3

up_b = arr[idx] > x it will give 6-1

after upperbound we have to subtract one it will be pointing to 11

another case:

x = 10

lb of 10 is 11 idx 6 which is not correct.

Also if x = 14
it will give 8 len of arr.

So a condition if

=> (arr[lb] != x) || ~~arr[lb]~~ (lb == n)
make -1

So we just return (-1, -1) this will not be true in case of 8.

Now only using plain binary search.

arr = [2, 3, 8, 8, 8, 8, 11, 13]

if n = 8

first = bs(arr, target, isfirst)

in bs code
like at at iteration we are at i = 3 and want first idx then we will move pointer to left side. and update answer accordingly but is it first false then move towards right. → ✓
code in files.

Search in Rotated array.

which is sorted.

[1, 2, 3, 4, 5]

↓↓

rotated

[4, 5, 1, 2, 3]

arr = [7, 8, 9, 1, 2, 3, 4, 5, 6] t = 1

0 1 2 3 4 5 6 7 8

BS

Standing at 2

we first check if we are in sorted and also in which half.

first identify the sorted half.

if target

~~arr[mid] <~~

first lower value should be < 2

so right half is sorted

like in eg

7, 8, 9, 1, [2, 3, 4, 5, 6] (sorted)

7 < 2 ✗

then see if

if arr[mid] <= target <= arr[h] ✗

then : h-1

So

7, 8, 9, 1
l m h

Now

7 <= 1 ✗

Now see target

the low = m + 1

(i) first check that which half is sorted.
and then
(ii) check if target in that half.
(iii) if not then h-1
(iii) else

low = mid + 1

Search in Rotated array 2
this have dubliate
elements.

the same code will fail for a case like this

1, 0, 1, 1, 1 t = 0
↑ ↑ ↑

left half is sorted in that way
and target not in this
cuz 0 < 1 so it will
set l = m+1

So we will trim down these conditions.

So the problem is
arr[m] == arr[l] == arr[h] => l = l+1, h = h-1
we will shrink our search space
like

1, 0, 1, 1, 1
↑ ↑ ↑
l m h

Find Minimum in Rotated arr

arr = [7, 8, 1, 2, 3, 4, 5, 6]
l m h

ans = Max int

[7, 8, 1, 2, 3, 4, 5, 6]
arr[l]

ans = 2

Now pick left side

arr = [7, 8, 1, 2, 3 ——]
l m h

ans = 2

arr [7, 8, 1]
l
h
m

ans = 1

→ pick up sorted part
→ then eliminate that part and move to next part.
→ Also check if ans is becoming lesser.

how many times array have been rotated.

arr = [3, 4, 5, 1, 2] and = 3

eg:

arr = [1, 2, 4, 5, 7]

arr is sorted 0 times.

last logic like

if we have the idx of minimum of sorted array then we can analyze that times it is rotated

like

[3, 4, 5, 1, 2]
      3

[1, 2, 3, 4, 5] => [3, 4, 5, 1, 2]

Single element in the sorted array

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

O(N) solution

either first element or last element or in between

```
for (i=0 → n)
    if (i=0)
        if(arr[i] != arr[i+1]):
            return arr[i]
    elif (i == n-1)
        if(arr[i] != arr[i-1])
            return arr[i]
    else:
        if [arr[i] != arr[i-1] &&
                arr[i] != arr[i+1])
            return arr[i];
```

O(logN) solution:

arr = [1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6]
e o e o e o o e o e

if we see the pattern

So the element left have indexing like (even, odd)
and element right hav indexing like (odd, even)

But for the comparison we have to eliminate the first and last idx
cuz then if we use this condition in code it will give index out of bounds error.

if (arr[i] != arr[i-1] &&
arr[i] != arr[i+1])

So for that the 3 base conditions are

1. if (len arr == 1)
return arr[0]

② if arr[0] != arr[1]:
    return arr[0]

③ if arr[n-1] != arr[n-2]
    return arr[n-2]

Now in code:

```
while l <= h:

    if (arr[i] != arr[i-1] and arr[i] != arr[i+1])
        return arr[i];
#to eliminate the half
    if (mid %2 == 1 && arr[mid-1] == arr[mid])
        # left half
        low = mid + 1
        -> Also we could be standing at even
    OR (mid %2 == 0 and arr[mid] = arr[m+1])
    else:
        h = mid - 1
```

Find Peak element:

arr[i-1] < arr[i] > arr[i+1]

arr = [1, 2, 3, 4, 5, 6, 7, 8, 5, 1] → 1 peak

arr = [1, 2, 1, 3, 5, 6, 4] → 2 peak

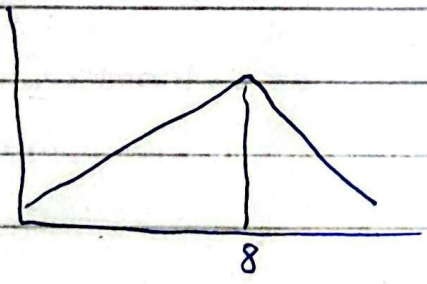
arr = -∞ [1, 2, 3, 4, 5] -∞

arr = -∞ [5, 4, 2, 3, 1] -∞

can be done in log(N)

if arr have 1 peak

arr = [1, 2, 3, 4, 5, 6, 7, 8, 5, 1] ans = 8
l h



either peak will be on right of mid
or left or mid is peak.

i=0

1, 2, 3, 4, 5, 6, 2, 8, 5, 1

l m h

if mid is not the peak then peak will always be or right

So

l = mid + 1

i = 1

6, 7, 8, 5, 1

l ↑ h

peak return

~~another case:~~

if element is in increasing curve then l = m+1

if element is in decreasing curve h = m-1

So if

arr[mid] < arr[mid+1]

then l = m+1

if

arr[mid] > arr[mid-1]

h = m-1

as we are assuming

$-\infty$ [arr] $-\infty$

So if arr[0] > arr[1] return

Same for last.