Find the minimum in a rotated sorted array

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

Example 1:

Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.

Example 2:

Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

Example 3:

Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

Logic:
check if sorted based on mid and last element.
if its sorted then the smallest will be in the left.
Bu tif unsorted will be only in the right

```
class Solution:
    def findMin(self, nums: List[int]) -> int:

        l,r=0,len(nums)-1

        while l<r:
            m= (l+r)//2

            if nums[m]>nums[r]:
                l=m+1
            else:
                r=m

        return  nums[l]
```

Now apply the logic to this question 👍

<u>Search inside a Rotated sorted Array I</u>

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

You must write an algorithm with O(log n) runtime complexity.

Logic:

1. if a sorted array is rotated, it has two separate sorted arrays.
2. If nums[mid]>nums[left]===> left to mid is in order, otherwise not in order.
3. So compare target with nums[left] and nums[mid] to reduce search space.

Code:

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:

        l=0
        r=len(nums)-1

        while l<=r:

            m=(l+r)//2

            if target==nums[m]:
                return m

            if nums[l]<=nums[m]:  #checking if portion to the left is in order
                if target<nums[m] and target>=nums[l]:   #constraint to reduce search space to left portion
                    r=m-1
                else:
                    l=m+1
            else:
                if target<nums[m] or target>=nums[l]: #constraint to reduce search space to left portion
                    r=m-1
                else:
                    l=m+1

        return -1
```

Complexity:
O(log(n)) since search space is reduced to half each time until search space is =1
Brute force: O(n) linear search.

Find the peak element

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Logic:
The side with the greater number will always have a peak element. why?
Think for example - if there was a greater element on m+1, then if there was a smaller element of m+2, element at m+1 would be peak, else if there was a greater element on m+2 then the same logic can be applied and we will be able to find the peak while the numbers are still increasing. (if its an array of increasing elements that follows, the last element will be the peak element)
Remember: we are only reducing the search space to a side that will guarantee a peak element.

```
class Solution:
def findPeakElement(self, nums: List[int]) -> int:

l,r=0,len(nums)-1
while l < r:
      mid = (l+r)//2
      if nums[mid]< nums[mid+1]:
            l=mid+1
      else:
            r=mid
      return r
```

Bonus : when would this fail?

Search space reduction problems:

Search a 2D Matrix II

Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

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

Example 1:

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

Input: matrix = [[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]], target = 5
Output: true

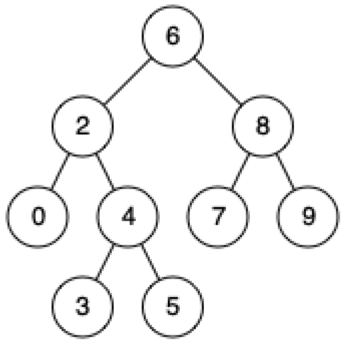
Logic : we know the multidimensional array is sorted horizontally and vertically, start search from an index such that the search conditions will lead you to the possible index of the element.
Think- if greater, then? Otherwise if lesser, then?

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:

        r,c=len(matrix)-1,0


        while(r>=0 and c<=len(matrix[0])-1):
            if(target < matrix[r][c]):
                r-=1
            elif(target > matrix[r][c]):
                c+=1
            elif(target==matrix[r][c]):
                return True
        return False
```

Lowest common ancestor of a binary search tree

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:



Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.

Logic: how can we reduce the search space knowing that in a BST smaller elements are in the left, greater elements are in the right of a node?
Hence the ancestor would be the element that is greater than the min of the two nodes and lesser than the max of the two nodes, because it would then be the first point of divide between the nodes (before they are part of two different subtrees emanating from the reference ancestor.)

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':



        def lca(root,p,q):

            if root:

                if root.val >=min(p.val,q.val) and root.val<= max(p.val,q.val):
                    return root

                elif root.val>max(p.val,q.val):
                    return lca(root.left,p,q)
                elif root.val<min(p.val,q.val):
                    return lca(root.right,p,q)

        return lca(root,p,q)
```

HW:
Go home and try out:

1. Search inside a Sorted Array Whose length is unknown
2. Find median of 2 sorted array (scary but dw)
3. Aggressive cows (very scary but dw)