**Binary Search**

Table of Contents

[LEVEL 1: **Amateur** 2](#_Toc207514946)

[1. Binary Search 2](#_Toc207514947)

[2. Lower Bound, upper bound 2](#_Toc207514948)

[3. First and last occurrence 2](#_Toc207514949)

[4. Number of occurrences in sorted array 2](#_Toc207514950)

[5. Find Kth Rotation 2](#_Toc207514951)

[6. Search in Rotated Sorted Array 2](#_Toc207514952)

[LEVEL 2: **Pro** 3](#_Toc207514953)

[LEVEL 3: **Legend** 4](#_Toc207514954)

[**SOLUTIONS:** 5](#_Toc207514955)

[**LEVEL 1:** 5](#_Toc207514956)

[**LEVEL 2:** 11](#_Toc207514957)

# LEVEL 1: **Amateur**

### Binary Search

Link: <https://leetcode.com/problems/binary-search/description/>

### Lower Bound, upper bound

Link: <https://www.geeksforgeeks.org/problems/implement-lower-bound/1>

Link: <https://www.geeksforgeeks.org/problems/implement-upper-bound/1>

### First and last occurrence

Link: <https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/>

### Number of occurrences in sorted array

Link: <https://www.geeksforgeeks.org/problems/number-of-occurrence2259/1>

### Find Kth Rotation

Link: <https://www.geeksforgeeks.org/problems/rotation4723/1>

### Search in Rotated Sorted Array

Link: <https://leetcode.com/problems/search-in-rotated-sorted-array/>

# LEVEL 2: **Pro**

### Find Peak Element

Link: <https://leetcode.com/problems/find-peak-element/description/>

### Find Peak Element

Link: <https://leetcode.com/problems/search-a-2d-matrix/description/>

### Find Peak Element

Link: <https://www.geeksforgeeks.org/problems/allocate-minimum-number-of-pages0937/1>

# LEVEL 3: **Legend**

# **SOLUTIONS:**

## **LEVEL 1:**

**1. Binary Search**

Iterative approach

class Solution:

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

        low, high = 0, len(nums)-1

        while low<=high:   #break condition

            mid = (low+high)//2        #find mid every time

            if nums[mid]==target:

                return mid

            if target>nums[mid]:

                low = mid+1

            else:

                high=mid-1

        return -1

Recursive approach

class Solution:

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

        def helper(low,high): #break condition

            if low>high:

                return -1

            mid = (low+high)//2

            if nums[mid]==target:

                return mid

            if target>nums[mid]:

                return helper(mid+1,high)

            return helper(low,mid-1)

        return helper(0,len(nums)-1)

Not a issue in python, but in other languages ( L + R)//2 can give integer overflow issue. So, in place of this use

L + ( L – R)//2.

**2. Lower Bound**

**Lower Bound**

The lower bound refers to the index of the first element in a sorted list that is **greater than or equal** to a given target value.

If the target value is **present** in the list, the **lower bound will be the index of its first occurrence**.

If the target value is **not present**, the lower bound will be the index where the target value should be inserted to maintain the sorted order.

# just iterate till you can, and low will reach to lower bound

class Solution:

    def lowerBound(self, arr, target):

        low, high = 0, len(arr)-1

        while low<=high:

            mid = (low+high)//2

            if target>arr[mid]:

                low = mid+1

            else:

                high = mid-1

        return low

#Can use python bisect library

import bisect

class Solution:

    def lowerBound(self, arr, target):

        #code here

        return bisect.bisect\_left(arr,target)

**Upper Bound**

The upper bound refers to the index of the first element in a sorted list that is **strictly greater** than a given target value.

If the target value is **present** in list, the **upper bound will be the index after the last occurrence of the target value**.

If the target value is **not present**, the **upper bound will be the same as the lower bound**, indicating the insertion point.

#using bisect library

import  bisect

class Solution:

    def upperBound(self, arr, target):

        return bisect.bisect\_right(arr,target)

class Solution:

    def upperBound(self, arr, target):

        n = len(arr)

        ans=n               #if it never goes inside else, len(arr) is ans

        low, high = 0 , n-1

        while low<=high:

            mid = (low+high)//2

            if target>=arr[mid]: #greater or equal, so that at low num is greater than target

                low = mid+1

            else:

                ans = mid

                high = mid-1

        return ans

**3. First and Last occurrence of element**

class Solution:  #using basic binary search concept, and tracking target index

    def leftOccurrence(self,nums,target):

        low, high = 0, len(nums)-1

        ans = -1

        while low<=high:

            #another way to do (low +high) may go out of range so use this

            mid = low + (high-low)//2

            #even if we get value, at left we can get lower index

            if target==nums[mid]:

                ans = mid

                high = mid-1

            elif target<nums[mid]:

                high = mid-1

            else:

                low = mid+1

        return ans

    def rightOccurrence(self,nums,target):

        low, high = 0, len(nums)-1

        ans = -1

        while low<=high:

            mid = low + (high-low)//2

            #even if we get value, at righ we can get higher index value

            if target==nums[mid]:

                ans = mid

                low = mid+1

            elif target<nums[mid]:

                high = mid-1

            else:

                low = mid+1

        return ans

    def searchRange(self, nums: List[int], target: int) -> List[int]:

        lower = self.leftOccurrence(nums,target)

        upper = self.rightOccurrence(nums,target)

        return [lower,upper]

Using upper bound and lower bound concept

import bisect

class Solution:

    def searchRange(self, nums: List[int], target: int) -> List[int]:

        #lower bound give first occurance

        low = bisect.bisect\_left(nums,target)

        #upper bound give num greater than target (so -1 for last occurance)

        high = bisect.bisect\_right(nums,target) - 1

        #if num is not present in array

        #can also have low > nums[len] if target>max(arr)

        if(low==len(nums) or nums[low]!=target):

            return [-1,-1]

        return [low,high]

**4. Number of occurrences in sorted array**

Just find upper and lower bound of target. All number in this range [low, high) have value target.

In case target is not present, for that lower and upper bound will be same, hence will return 0

import bisect

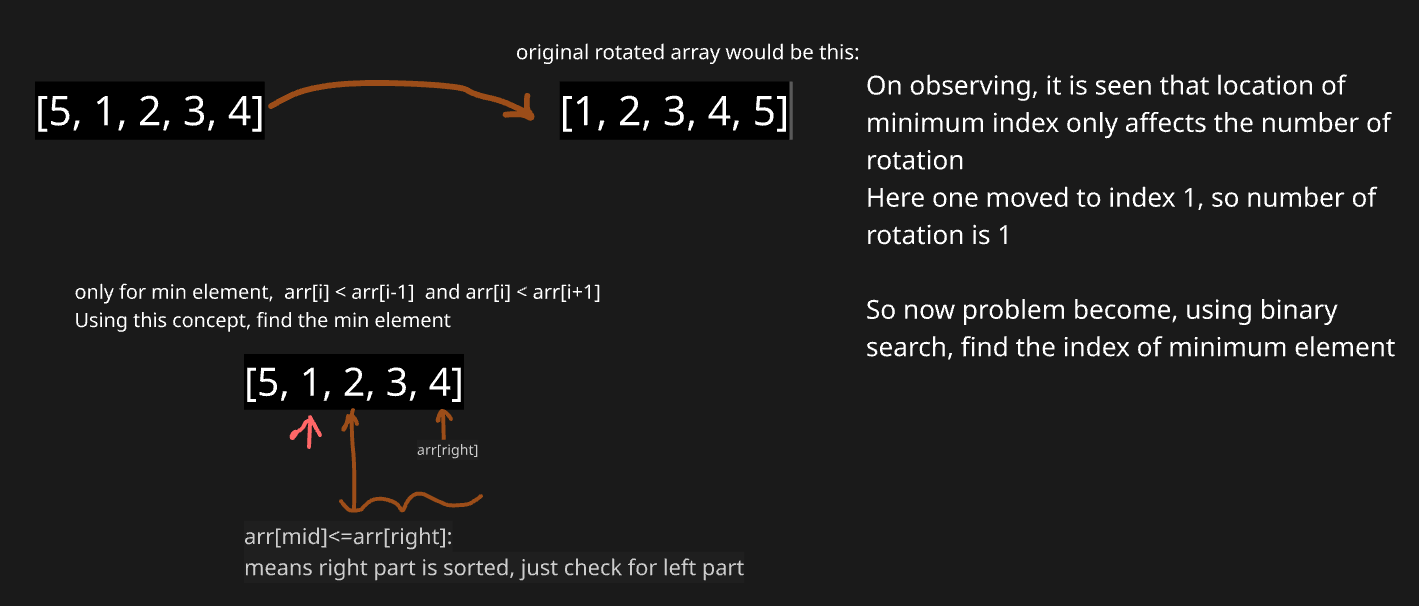
class Solution:

    def countFreq(self, arr, target):

        # code here

        return bisect.bisect\_right(arr,target)-bisect.bisect\_left(arr,target)

**5. Find Kth Rotation**

****

**Below shows, how min will value will be present in left, by depicting 2 possible scenarios**

****

#reason why not mid-1

#for arr: 8,10,2,5,6: 2---6 is sorted array, 2 is min, if we do mid-1, we miss 2

class Solution:

    def findKRotation(self, arr):

        left,right = 0, len(arr)-1

        while(left<right):

            mid = (left+right)//2

            if arr[mid]<=arr[right]:

                right = mid   #remember it is not mid-1

            else:

                left = mid+1

        return left

**6. Search in Rotated Sorted Array**

This code is solving **search in a rotated sorted array** using binary search.

* It keeps two pointers (left and right) and repeatedly finds the middle index (mid).
* If nums[mid] == target, return the index.
* Otherwise, it checks which half (left or right) is **sorted**.
  + If the **left half** is sorted, it checks if the target lies in that half → adjust right.
  + Else, the **right half** is sorted, and it checks if the target lies there → adjust left.
* If the element is not found, return -1.

Basically, it narrows down the search by always picking the sorted half where the target could exist.

class Solution:

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

        left, right = 0, len(nums) - 1

        while left <= right:

            mid = (left + right) // 2

            if nums[mid] == target:

                return mid

            #left half is sorted

            if nums[left] <= nums[mid]:

                if nums[left] <= target < nums[mid]:

                    right = mid - 1

                else:

                    left = mid + 1

            #right half is sorted

            else:

                if nums[mid] < target <= nums[right]:

                    left = mid + 1

                else:

                    right = mid - 1

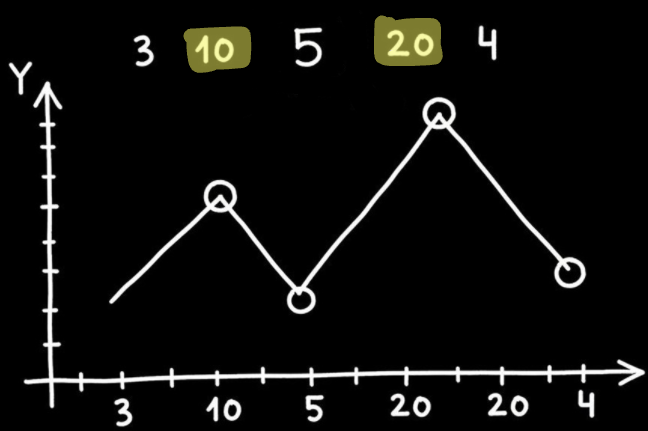
        return -1

## **LEVEL 2:**

**1. Find Peak Element**

**Possible Cases of Input:**

1. **Single element**: [1] → return 0 (only element is peak).
2. **Strictly increasing**: [1,2,3,4] → peak at last index.
3. **Strictly decreasing**: [4,3,2,1] → peak at first index.
4. **Peak in middle**: [1,3,2] → returns 1.
5. **Multiple peaks**: [1,2,1,3,5,6,4] → could return index 1 or 5.
6. **Duplicates**: Problem states all elements are **distinctly comparable**, so not allowed.



class Solution:

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

        n = len(nums)

        # Edge cases ( values at corners)

        if n == 1:

            return 0

        if nums[0] > nums[1]:

            return 0

        if nums[n-1] > nums[n-2]:

            return n-1

        left, right = 1, n-2 #after ignoring corner values

        while left <= right:

            mid = (left + right) // 2

            if nums[mid] > nums[mid-1] and nums[mid] > nums[mid+1]:

                return mid

            #larger values lie in left, so iterate that half

            elif nums[mid] < nums[mid-1]:

                right = mid - 1

            else:

                left = mid + 1

        return -1  # shouldn't happen

Explanation:

1. Here at start we check for elements at first and last index, if they are greater than their single respective neighbours, return them as answers
2. Else check in left over elements, Remember array is not sorted, here we are just eliminating halves thus applying Binary Search. Also there will always be atleast 1 peak in any array, if array is sorted than also leftmost or rightmost element is peak.
3. If mid is greater than its respective neighbours, it is answer
4. Else if it is smaller than left neighbour, that means larger values are in left side array, just check there. Remember though it is also possible to still have peak on right side, but that case is one with multiple peaks, so just check one side.

**Easier code:**

class Solution:

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

        left, right = 0, len(nums) - 1

        # Binary search approach

        while left < right:

            mid = (left + right) // 2

            # If mid is greater than next, peak is on the left side (including mid)

            if nums[mid] > nums[mid + 1]:

                right = mid

            else:

                # Otherwise peak must be on the right side

                left = mid + 1

        # left == right → peak index

        return left

In place of checking every time for max value from both neighbour, here we just shrink the observable array and at end make left point to peak element.

**2. Search in 2D matrix**

class Solution:

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

        n = len(matrix)

        m = len(matrix[0])

        l, r = 0, m-1

        while r>=0 and l<n:

            if matrix[l][r]==target:

                return True

            if matrix[l][r]>target:

                r-=1

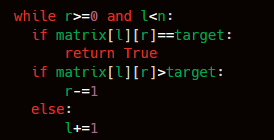
            else:

                l+=1

        return False

**Explanation**

1. Start search at **top-right** element (matrix[0][m-1]).
2. From this position, you can **move left** (smaller numbers) or **down** (larger numbers).



1. If current element is too **big**, move **left**.
2. If current element is too **small**, move **down**.
3. This works because rows & columns are sorted.

If we exit the loop, it means we scanned all possible candidates → target not found.

**3. Allocate Minimum Pages**

class Solution:

    def check(self,arr,mid,k):

        student\_cnt = 1

        max\_p = 0  #pages assigned to current student

        for page in arr:

            if max\_p + page <= mid:

                max\_p += page

            else:

                student\_cnt += 1

                max\_p = page

        return student\_cnt <= k

    def findPages(self, arr, k):

        n = len(arr)

        # if number of books < number of students: return -1

        if n<k:

            return -1

        l = max(arr)

        r = sum(arr)

        ans=-1

        while l <= r:

            mid = (l+r)//2

            if self.check(arr,mid,k):

                ans = mid  #update answer to mid

                r = mid - 1 #try for smaller value

            else:

                l = mid + 1  #try for larger value

        return ans

* We have n books in an array arr[], where arr[i] = pages in book i.
* We must allocate books **in order** to k students.
* Each student gets **contiguous books**.
* Goal: **minimize the maximum pages assigned to any student**.

**Approach**:

**Note**: given arr is not required to be sorted:

The approach uses **binary search on the answer**: we guess the maximum pages (mid) a student can get, then check feasibility by allocating books sequentially. If feasible, try smaller; else increase. This minimizes the maximum pages per student.

