Q1. Given a non-negative integer x, return *the square root of* x *rounded down to the nearest integer*. The returned integer should be **non-negative** as well.

You **must not use** any built-in exponent function or operator.

* For example, do not use pow(x, 0.5) in c++ or x \*\* 0.5 in python.

def floorSqrt(x):

# Base cases

if (x == 0 or x == 1):

return x

# Do Binary Search for floor(sqrt(x))

start = 1

end = x//2

while (start <= end):

mid = (start + end) // 2

# If x is a perfect square

if (mid\*mid == x):

return mid

# Since we need floor, we update

# answer when mid\*mid is smaller

# than x, and move closer to sqrt(x)

if (mid \* mid < x):

start = mid + 1

ans = mid

else:

# If mid\*mid is greater than x

end = mid-1

return ans

Q2. 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.

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

s,e,n = 0,len(nums)-1 ,len(nums);

# test case of array of len 1 and 2

if n==1:return 0 ;

if n==2:

if nums[1]>nums[0]:return 1;

if nums[0]>nums[1]:return 0;

while s<=e:

m = int(s+(e-s)/2);

if m>0 and m<n-1:

if(nums[m]>nums[m-1] and nums[m]>nums[m+1]): return m;

elif nums[m]<nums[m-1]:e=m-1;

elif nums[m]<nums[m+1]:s=m+1;

elif m==0:

if nums[m]>nums[m+1]:return 0;

else : s=m+1;

elif m==n-1:

if nums[m]>nums[m-1]:return n-1;

else :e=m-1;

return -1;

Q3. Given an array nums containing n distinct numbers in the range [0, n], return *the only number in the range that is missing from the array.*

class Solution:

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

nums.sort()

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

while left <= right:

middle = (left + right) // 2

if nums[middle] == middle:

left = middle + 1

else:

right = middle - 1

return left

Q4. Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return this repeated number.

You must solve the problem **without** modifying the array nums and uses only constant extra space.

class Solution:

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

start = 1

end = len(nums)

ans = float('inf')

def bs(start, end):

nonlocal ans

if start > end:

return -1

else:

mid = (start + end) // 2

cnt = 0

for i in range(len(nums)):

if nums[i] <= mid:

cnt += 1

if cnt > mid:

ans = mid

return bs(start, mid-1)

else:

return bs(mid+1, end)

bs(start, end)

return ans

Q5. Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:

        if len(nums1) < len(nums2):

            nums1, nums2 = nums2, nums1

        nums1.sort()

        nums2 = set(nums2)

        res = []

        for i in nums2:

            l, r = 0, len(nums1)-1

            while l <= r:

                mid = (l+r)//2

                val = nums1[mid]

                if val == i:

                    res.append(i)

                    break

                elif val < i:

                    l = mid + 1

                else:

                    r = mid - 1

        return res

Q6. 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*.

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

class Solution(object):

def findMin(self, nums):

start = 0

end = len(nums)-1

while (start < end):

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

if (nums[mid] < nums[end]):

end = mid

else:

start = mid + 1

return nums[start]

Q7. Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

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

class Solution:

# for this solution we need to modify our binary search since we can't return

# mid here.

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

# searching from left and then from right

left = self.binSearch(nums, target, True)

right = self.binSearch(nums, target, False)

return [left, right]

# if leftBias=True, we search for left side

# if False, we seach for right side

def binSearch(self, nums, target, leftBias):

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

i = -1

while l <= r:

m = (l+r) // 2

if nums[m] < target:

l = m + 1

elif nums[m] > target:

r = m - 1

else: # if nums[m] == target

i = m

if leftBias: # if I'm searching from left-side

r = m - 1

else: # if I'm searching from right-side

l = m + 1

return i

**Question 8**

Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may order

class Solution:

def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:

res = []

for e in nums1:

ans, pos = self.search(e, nums2)

if ans:

del nums2[pos]#will make BS more quick

res.append(e)

return res

def search(self, value, a):

a.sort()

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

while l<=r:

m = l+(r-l)//2

if a[m]==value:

return True, m

elif a[m]<value:

l=m+1

else:

r=m-1

return None, -1