**DSA – ASSIGNMENT 11**

💡 **Question 1** 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.

**Example 1:**

Input: x = 4

Output: 2

Explanation: The square root of 4 is 2, so we return 2.

**Example 2:**

Input: x = 8

Output: 2

Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is returned.

**Solution. :-**

* If x is 0 or 1, return x as the square root.
* Initialize two variables left and right as 1 and x respectively. These will represent the range in which the square root lies.
* While left is less than or equal to right:
  + Calculate the middle value as (left + right) // 2.
  + If the square of the middle value is less than or equal to x and the square of the next integer is greater than x, return the middle value as the square root.
  + If the square of the middle value is greater than x, update right to mid - 1.
  + Otherwise, update left to mid + 1.
* If the loop exits without returning, return right as the square root.

**def mySqrt(x):**

**if x == 0 or x == 1:**

**return x**

**left = 1**

**right = x**

**while left <= right:**

**mid = (left + right) // 2**

**square = mid \* mid**

**if square <= x < (mid + 1) \* (mid + 1):**

**return mid**

**elif square > x:**

**right = mid - 1**

**else:**

**left = mid + 1**

**return right**

**# Test the function**

**x = 4**

**print(mySqrt(x))**

💡 **Question 2** 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.

**Example 1:**

Input: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

**Example 2:**

Input: nums = [1,2,1,3,5,6,4]

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

**Solution. :-**

* Initialize two pointers, left and right, as 0 and the length of nums minus 1, respectively.
* While left is less than right:
  + Calculate the middle index as (left + right) // 2.
  + Compare the middle element nums[mid] with its neighboring elements nums[mid-1] and nums[mid+1].
  + If nums[mid] is greater than both of its neighbors, return mid as the peak index.
  + If nums[mid] is less than nums[mid+1], update left to mid + 1.
  + Otherwise, update right to mid - 1.
* Return left as the peak index.

**def findPeakElement(nums):**

**left = 0**

**right = len(nums) - 1**

**while left < right:**

**mid = (left + right) // 2**

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

**return mid**

**elif nums[mid] < nums[mid+1]:**

**left = mid + 1**

**else:**

**right = mid - 1**

**return left**

**# Test the function**

**nums = [1, 2, 3, 1]**

**print(findPeakElement(nums))**

💡 **Question 3** 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.*

**Example 1:**

Input: nums = [3,0,1]

Output: 2

Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

**Example 2:**

Input: nums = [0,1]

Output: 2

Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

**Example 3:**

Input: nums = [9,6,4,2,3,5,7,0,1]

Output: 8

Explanation: n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

**Solution. :-**

* Initialize a variable missing to n, which represents the missing number in the range.
* Iterate over each element num in the array nums:
  + Update missing by taking the XOR of missing and num.
  + Update missing by taking the XOR of missing and its index i.
* Return the value of missing, which represents the missing number in the range.

**def missingNumber(nums):**

**n = len(nums)**

**missing = n**

**for i, num in enumerate(nums):**

**missing ^= num**

**missing ^= i**

**return missing**

**# Test the function**

**nums = [3, 0, 1]**

**print(missingNumber(nums))**

💡 **Question 4** 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.

**Example 1:**

Input: nums = [1,3,4,2,2]

Output: 2

**Example 2:**

Input: nums = [3,1,3,4,2]

Output: 3

**Solution. :-**

* Initialize two pointers, slow and fast, as the first element of the array.
* Move slow one step at a time and fast two steps at a time until they meet at the same element.
  + This is done to find the meeting point, which indicates the presence of a cycle in the array.
* Reset slow to the first element.
* Move slow and fast one step at a time until they meet again.
  + This is done to find the entrance of the cycle, which corresponds to the repeated number in the array.
* Return the value of slow, which represents the repeated number.

**def findDuplicate(nums):**

**slow = nums[0]**

**fast = nums[0]**

**# Find the meeting point**

**while True:**

**slow = nums[slow]**

**fast = nums[nums[fast]]**

**if slow == fast:**

**break**

**# Find the entrance of the cycle**

**slow = nums[0]**

**while slow != fast:**

**slow = nums[slow]**

**fast = nums[fast]**

**return slow**

**# Test the function**

**nums = [1, 3, 4, 2, 2]**

**print(findDuplicate(nums))**

💡 **Question 5** 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**.

**Example 1:**

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2]

**Example 2:**

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [9,4]

Explanation: [4,9] is also accepted.

**Solution. :-**

* Initialize two sets, set1 and set2, and populate them with the elements from nums1 and nums2 respectively.
* Initialize an empty result list.
* Iterate over each element num in set1:
  + If num is present in set2, add it to the result list.
* Return the result list.

**def intersection(nums1, nums2):**

**set1 = set(nums1)**

**set2 = set(nums2)**

**result = []**

**for num in set1:**

**if num in set2:**

**result.append(num)**

**return result**

**# Test the function**

**nums1 = [1, 2, 2, 1]**

**nums2 = [2, 2]**

**print(intersection(nums1, nums2))**

💡 **Question 6** 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.

**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.

**Solution. :-**

* Initialize two pointers, left and right, as the first and last indices of the array nums.
* While left is less than right:
  + Calculate the middle index as (left + right) // 2.
  + Check if the middle element nums[mid] is greater than the last element nums[right].
    - If true, it means the minimum element is located in the right half of the array. Update left to mid + 1.
    - If false, it means the minimum element is located in the left half of the array or is equal to nums[mid]. Update right to mid.
* Return the value of nums[left], which represents the minimum element.

**def findMin(nums):**

**left = 0**

**right = len(nums) - 1**

**while left < right:**

**mid = (left + right) // 2**

**if nums[mid] > nums[right]:**

**left = mid + 1**

**else:**

**right = mid**

**return nums[left]**

**# Test the function**

**nums = [3, 4, 5, 1, 2]**

**print(findMin(nums))**

💡 **Question 7** 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.

**Example 1:**

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

**Example 2:**

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

**Example 3:**

Input: nums = [], target = 0

Output: [-1,-1]

**Solution. :-**

* Initialize two variables, start and end, as -1. These will store the starting and ending positions of the target value.
* Perform a binary search to find the leftmost occurrence of the target value:
  + Initialize two pointers, left and right, as the first and last indices of the array nums.
  + While left is less than or equal to right:
    - Calculate the middle index as (left + right) // 2.
    - If the middle element is equal to the target, update start to the middle index and move right to the left of the middle index to search for a leftmost occurrence.
    - If the middle element is greater than the target, update right to mid - 1.
    - If the middle element is less than the target, update left to mid + 1.
* Perform a binary search to find the rightmost occurrence of the target value:
  + Initialize left as start and right as the last index of the array nums.
  + While left is less than or equal to right:
    - Calculate the middle index as (left + right) // 2.
    - If the middle element is equal to the target, update end to the middle index and move left to the right of the middle index to search for a rightmost occurrence.
    - If the middle element is greater than the target, update right to mid - 1.
    - If the middle element is less than the target, update left to mid + 1.
* Return the list [start, end], which represents the starting and ending positions of the target value.

**def searchRange(nums, target):**

**start = -1**

**end = -1**

**# Find the leftmost occurrence**

**left = 0**

**right = len(nums) - 1**

**while left <= right:**

**mid = (left + right) // 2**

**if nums[mid] == target:**

**start = mid**

**right = mid - 1**

**elif nums[mid] < target:**

**left = mid + 1**

**else:**

**right = mid - 1**

**# Find the rightmost occurrence**

**left = start**

**right = len(nums) - 1**

**while left <= right:**

**mid = (left + right) // 2**

**if nums[mid] == target:**

**end = mid**

**left = mid + 1**

**elif nums[mid] < target:**

**left = mid + 1**

**else:**

**right = mid - 1**

**return [start, end]**

**# Test the function**

**nums = [5, 7, 7, 8, 8, 10]**

**target = 8**

**print(searchRange(nums, target))**

💡 **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 return the result in **any order**.

**Example 1:**

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2,2]

**Example 2:**

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [4,9]

Explanation: [9,4] is also accepted.

**Solution. :-**

* Initialize an empty hash map freqMap to store the frequencies of elements in nums1.
* Iterate through each element num in nums1:
  + If num exists in freqMap, increment its frequency by 1.
  + If num does not exist in freqMap, add it to the map with a frequency of 1.
* Initialize an empty list intersection to store the common elements.
* Iterate through each element num in nums2:
  + If num exists in freqMap and its frequency is greater than 0, append num to intersection and decrement its frequency by 1 in freqMap.
* Return the intersection list.

**from collections import defaultdict**

**def intersect(nums1, nums2):**

**freqMap = defaultdict(int)**

**# Count frequencies in nums1**

**for num in nums1:**

**freqMap[num] += 1**

**# Find intersection**

**intersection = []**

**for num in nums2:**

**if freqMap[num] > 0:**

**intersection.append(num)**

**freqMap[num] -= 1**

**return intersection**

**# Test the function**

**nums1 = [1, 2, 2, 1]**

**nums2 = [2, 2]**

**print(intersect(nums1, nums2))**