NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO 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.

ANS –To find the square root of a non-negative integer x without using any built-in exponent function or operator, we can utilize a binary search algorithm.

Here's the step-by-step process:

Set the variables left and right to 0 and x respectively. These variables will represent the search range for the square root.

While left is less than or equal to right, do the following steps:

Calculate the midpoint of the range as mid = (left + right) / 2.

If mid \* mid is greater than x, set right to mid - 1 since the square root is in the lower half of the range.

Otherwise, if mid \* mid is less than x, set left to mid + 1 since the square root is in the upper half of the range.

If neither of the above conditions is met, it means mid \* mid is equal to x, so we can return mid as the square root of x.

After the loop ends, the value of right will be the largest integer whose square is less than or equal to x. Thus, we return right as the square root of x.

Here's the implementation of the above algorithm in Python:

def mySqrt(x):

if x < 2:

return x

left, right = 0, x

while left <= right:

mid = (left + right) // 2

if mid \* mid > x:

right = mid - 1

elif mid \* mid < x:

left = mid + 1

else:

return mid

return right

Let's test the function with the provided examples:

print(mySqrt(4)) # Output: 2

print(mySqrt(8)) # Output: 2

The function produces the expected results.

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.

ANS –

To find a peak element in an integer array nums using an algorithm that runs in O(log n) time, we can utilize a binary search approach.

Here's the step-by-step process:

Set the variables left and right to 0 and len(nums) - 1 respectively. These variables represent the search range.

While left is less than right, do the following steps:

Calculate the midpoint of the range as mid = (left + right) // 2.

If nums[mid] < nums[mid + 1], it means the peak element is in the right half of the range. Set left to mid + 1.

Otherwise, if nums[mid] > nums[mid + 1], it means the peak element is in the left half of the range or mid itself is the peak. Set right to mid.

After the loop ends, left and right will converge to the peak element. Return left or right as the index of the peak element.

Here's the implementation of the above algorithm in Python:

def findPeakElement(nums):

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

while left < right:

mid = (left + right) // 2

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

left = mid + 1

else:

right = mid

return left

Let's test the function with the provided examples:

print(findPeakElement([1, 2, 3, 1])) # Output: 2

print(findPeakElement([1, 2, 1, 3, 5, 6, 4])) # Output: 5

The function produces the expected results.

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.

ANS –

To find the missing number in an array nums containing n distinct numbers in the range [0, n], we can utilize the concept of XOR.

Here's the step-by-step process:

Initialize a variable missing with the value n. This will represent the missing number.

Iterate over each element num in nums and update missing by performing the XOR operation between missing and num and the index of num.

After the iteration, the value of missing will be the missing number.

Here's the implementation of the above algorithm in Python:

def missingNumber(nums):

n = len(nums)

missing = n

for i, num in enumerate(nums):

missing ^= i ^ num

return missing

Let's test the function with the provided examples:

print(missingNumber([3, 0, 1])) # Output: 2

print(missingNumber([0, 1])) # Output: 2

print(missingNumber([9, 6, 4, 2, 3, 5, 7, 0, 1])) # Output: 8

The function produces the expected results.

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

ANS –

To find the repeated number in an array nums containing n + 1 integers where each integer is in the range [1, n] inclusive, we can utilize the concept of cycle detection in a linked list.

Here's the step-by-step process:

Initialize two pointers, slow and fast, to the first element of the array.

Move slow one step forward and fast two steps forward in each iteration until they meet.

Once slow and fast meet, reset slow to the first element of the array.

Move slow and fast one step forward in each iteration until they meet again.

The meeting point of slow and fast is the repeated number.

Here's the implementation of the above algorithm in Python:

def findDuplicate(nums):

slow = nums[0]

fast = nums[0]

# Move slow one step and fast two steps

slow = nums[slow]

fast = nums[nums[fast]]

# Find the meeting point of slow and fast

while slow != fast:

slow = nums[slow]

fast = nums[nums[fast]]

# Reset slow to the first element

slow = nums[0]

# Move slow and fast one step until they meet again

while slow != fast:

slow = nums[slow]

fast = nums[fast]

return slow

Let's test the function with the provided examples:

print(findDuplicate([1, 3, 4, 2, 2])) # Output: 2

print(findDuplicate([3, 1, 3, 4, 2])) # Output: 3

The function produces the expected results.

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.

ANS –

To find the intersection of two integer arrays nums1 and nums2, we can utilize Python's built-in set data structure.

Here's the step-by-step process:

Convert nums1 and nums2 into sets, set1 and set2, respectively.

Use the intersection method on set1 with set2 to find the common elements between the two sets.

Convert the resulting set into a list and return it as the intersection.

Here's the implementation of the above algorithm in Python:

def intersection(nums1, nums2):

set1 = set(nums1)

set2 = set(nums2)

return list(set1.intersection(set2))

Let's test the function with the provided examples:

print(intersection([1, 2, 2, 1], [2, 2])) # Output: [2]

print(intersection([4, 9, 5], [9, 4, 9, 8, 4])) # Output: [9, 4]

The function produces the expected results.

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.

ANS –

To find the minimum element in a sorted rotated array nums of unique elements, we can utilize a modified binary search algorithm.

Here's the step-by-step process:

Set two pointers, left and right, to the first and last indices of the array respectively.

While left is less than right, do the following steps:

Calculate the midpoint of the range as mid = (left + right) // 2.

If nums[mid] > nums[right], it means the minimum element is in the right half of the range. Set left to mid + 1.

Otherwise, if nums[mid] <= nums[right], it means the minimum element is in the left half of the range or mid itself is the minimum. Set right to mid.

After the loop ends, left will be pointing to the minimum element.

Return nums[left] as the minimum element.

Here's the implementation of the above algorithm in Python:

def findMin(nums):

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

while left < right:

mid = (left + right) // 2

if nums[mid] > nums[right]:

left = mid + 1

else:

right = mid

return nums[left]

Let's test the function with the provided examples:

print(findMin([3, 4, 5, 1, 2])) # Output: 1

print(findMin([4, 5, 6, 7, 0, 1, 2])) # Output: 0

print(findMin([11, 13, 15, 17])) # Output: 11

The function produces the expected results.

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]

ANS –

To find the starting and ending position of a given target value in a sorted non-decreasing array nums, we can utilize a modified binary search algorithm.

Here's the step-by-step process:

Set two variables, left and right, to store the leftmost and rightmost indices of the target range. Initialize left to -1 and right to -1.

Use binary search to find the leftmost index of the target value:

Set low to 0 and high to the length of the array minus 1.

While low is less than or equal to high, do the following:

Calculate the midpoint as mid = (low + high) // 2.

If nums[mid] is greater than or equal to the target value, update right to mid and set high to mid - 1.

Otherwise, if nums[mid] is less than the target value, set low to mid + 1.

Use binary search to find the rightmost index of the target value:

Set low to 0 and high to the length of the array minus 1.

While low is less than or equal to high, do the following:

Calculate the midpoint as mid = (low + high) // 2.

If nums[mid] is less than or equal to the target value, update left to mid and set low to mid + 1.

Otherwise, if nums[mid] is greater than the target value, set high to mid - 1.

Finally, return [left, right] as the starting and ending positions of the target value.

Here's the implementation of the above algorithm in Python:

def searchRange(nums, target):

left = -1

right = -1

# Find the leftmost index of the target value

low, high = 0, len(nums) - 1

while low <= high:

mid = (low + high) // 2

if nums[mid] >= target:

right = mid

high = mid - 1

else:

low = mid + 1

# Find the rightmost index of the target value

low, high = 0, len(nums) - 1

while low <= high:

mid = (low + high) // 2

if nums[mid] <= target:

left = mid

low = mid + 1

else:

high = mid - 1

return [left, right]

Let's test the function with the provided examples:

print(searchRange([5, 7, 7, 8, 8, 10], 8)) # Output: [3, 4]

print(searchRange([5, 7, 7, 8, 8, 10], 6)) # Output: [-1, -1]

print(searchRange([], 0)) # Output: [-1, -1]

The function produces the expected results.

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.

ANS –

To find the intersection of two integer arrays nums1 and nums2, we can use a hash map to store the frequency of each number in nums1. Then, while iterating through nums2, we can check if the current number exists in the hash map and its frequency is greater than zero. If it does, we add it to the result array and decrement its frequency in the hash map.

Here's the step-by-step process:

Create a hash map freqMap to store the frequency of numbers in nums1.

Iterate through nums1 and update the frequency of each number in freqMap.

Initialize an empty array intersection to store the intersection of nums1 and nums2.

Iterate through nums2:

If the current number exists in freqMap and its frequency is greater than zero, add it to intersection and decrement its frequency in freqMap.

Return the intersection array as the result.

Here's the implementation of the above algorithm in Python:

from collections import defaultdict

def intersect(nums1, nums2):

freqMap = defaultdict(int)

for num in nums1:

freqMap[num] += 1

intersection = []

for num in nums2:

if freqMap[num] > 0:

intersection.append(num)

freqMap[num] -= 1

return intersection

Let's test the function with the provided examples:

print(intersect([1, 2, 2, 1], [2, 2])) # Output: [2, 2]

print(intersect([4, 9, 5], [9, 4, 9, 8, 4])) # Output: [4, 9]

The function produces the expected results.