**Assignment 11 Questions - Binary Search | DSA**

**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

**------------**

**Solution :** class Solution(object):

**------------** def mySqrt(self, x):

# right is set to x + 1 to deal with special cases like x = 0, x = 1

left, right = 0, x + 1

while left < right:

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

# Square root means x\*x = y

if mid \* mid > x:

right = mid

else:

left = mid + 1

# because of our loop we break when we encounter a mid which when squared is more than our target

return left - 1

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : O(log(X)).
* Space Complexity : O(1).

====================================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**

**--------------**

**Algorithm :**

**--------------**

**Case 1.** In this case, we firstly find 3 as the middle element. Since it lies on a falling slope, we reduce the search space to [1, 2, 3]. For this subarray, 2 happens to be the middle element, which again lies on a falling slope, reducing the search space to [1, 2]. Now, 1 acts as the middle element and it lies on a falling slope, reducing the search space to [1] only. Thus, 1 is returned as the peak correctly.

**Case 2.** In this case, we firstly find 3 as the middle element. Since it lies on a rising slope, we reduce the search space to [4, 5]. Now, 4 acts as the middle element for this subarray and it lies on a rising slope, reducing the search space to [5] only. Thus, 5 is returned as the peak correctly.

**Case 3.** In this case, the peak lies somewhere in the middle. The first middle element is 4. It lies on a rising slope, indicating that the peak lies towards its right. Thus, the search space is reduced to [5, 1]. Now, 5 happens to be the on a falling slope(relative to its right neighbour), reducing the search space to [5] only. Thus, 5 is identified as the peak element correctly.

**------------**

**Solution :** class Solution():

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

return self.search(nums, 0, len(nums) - 1)

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

if l == r:

return l

mid = (l + r) // 2

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

return self.search(nums, l, mid)

return self.search(nums, mid + 1, r)

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : O(log2(n)). We reduce the search space in half at every step. Thus, the total search space will be consumed in log2(n) steps. Here, n refers to the size of nums array.
* Space Complexity : O(log2(n)). We reduce the search space in half at every step. Thus, the total search space will be consumed in log2(n) steps. Thus, the depth of recursion tree will go upto log2(n).

====================================x==================================

**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

**------------**

**Solution :** class Solution():

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

numSet = set(nums)

expectedNumCount = len(nums) + 1

for number in range(expectedNumCount):

if number not in numSet:

return number

return -1

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : O(n). Because the set allows for O(1) containment queries, the main loop runs in O(n) time.
* Space Complexity : O(n). nums contains n-1 distinct elements, so it costs O(n) space to store a set containing all of them.

====================================x==================================

**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

**------------**

**Solution :** class Solution():

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

nums.sort()

for i in range(1, len(nums)):

if nums[i] == nums[i-1]:

return nums[i]

return -1

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : O(nlogn). Sorting takes O(nlog⁡n) time. This is followed by a linear scan, resulting in a total of O(nlog⁡n) + O(n) = O(nlog⁡n) time.
* Space Complexity : O(log⁡n) or O(n). In Python, the sort() function is implemented using the Timsort algorithm, which has a worst-case space complexity of O(n).

**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]**

**------------**

**Solution :** class Solution():

**------------** def set\_intersection(self, set1, set2):

return [x for x in set1 if x in set2]

def intersection(self, nums1, nums2):

set1 = set(nums1)

set2 = set(nums2)

if len(set1) < len(set2):

return self.set\_intersection(set1, set2)

else:

return self.set\_intersection(set2, set1)

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : O(n+m), where n and m are arrays' lengths. O(n) time is used to convert nums1 into set, O(m) time is used to convert nums2, and contains/in operations are O(1) in the average case.
* Space complexity : O(m+n) in the worst case when all elements in the arrays are different.

====================================x==================================

**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

**--------------**

**Algorithm :**

**--------------**

1. Find the mid element of the array.

2. If mid element > first element of array this means that we need to look for the inflection point on the right of mid.

3. If mid element < first element of array this that we need to look for the inflection point on the left of mid.

In the above example mid element 6 is greater than first element 4. Hence we continue our search for the inflection point to the right of mid.

4 . We stop our search when we find the inflection point, when either of the two conditions is satisfied:

nums[mid] > nums[mid + 1] Hence, mid+1 is the smallest.

nums[mid - 1] > nums[mid] Hence, mid is the smallest.

**------------**

**Solution :** class Solution():

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

if len(nums) == 1:

return nums[0]

left = 0

right = len(nums) - 1

if nums[right] > nums[0]:

return nums[0]

# Binary search way

while right >= left:

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

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

return nums[mid + 1]

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

return nums[mid]

if nums[mid] > nums[0]:

left = mid + 1

else:

right = mid - 1

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : Same as Binary Search O(logN).
* Space Complexity : O(1)

====================================x==================================

**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]

**------------**

**Solution :** from collections import defaultdict

**------------** class Solution():

@staticmethod

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

if len(nums1) > len(nums2):

return Solution.intersect(nums2, nums1)

m = defaultdict(int)

for val in nums1:

m[val] += 1

k = 0

for val in nums2:

if m[val] > 0:

nums1[k] = val

k += 1

m[val] -= 1

return nums1[:k]

if \_\_name\_\_ == '\_\_main\_\_':

nums1 = [1, 2, 2, 1]

nums2 = [2, 2]

ans = Solution.intersect(nums1, nums2)

print(ans)

**-------------------------**

**Complexity Analysis:**

**-------------------------**

* Time complexity : **O(n+m).** Where n and m are lengths of the array. We iterate linearly through both the arrays and insert and fetch operation in hash map takes constant time.
* [Space Complexity](https://en.wikipedia.org/wiki/Space_complexity) : O(min(n,m)). We use hash map to store the elements of the smaller array.

====================================x==================================