**Assignment 2 Questions - Arrays | DSA**

**Question 1** Given an integer array nums of 2n integers, group these integers into n pairs (a1, b1), (a2, b2),..., (an, bn) such that the sum of min(ai, bi) for all i is maximized. Return the maximized sum.

**Example 1:** Input: nums = [1,4,3,2] Output: 4

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

**Solution:** class Solution:

**-----------** def arrayPairSum(self, nums):

nums.sort()

result = 0

numsLen = len(nums)

for i in range(0, numsLen - 1, 2):

result += nums[i]

return result

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

**Complexity Analysis:**

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

* Time complexity: **O(NlogN), N is length of the input array**
* Space complexity: O(1)

**===================================x=================================**

**Question 2** Alice has n candies, where the ith candy is of type candyType[i]. Alice noticed that she started to gain weight, so she visited a doctor. The doctor advised Alice to only eat n / 2 of the candies she has (n is always even). Alice likes her candies very much, and she wants to eat the maximum number of different types of candies while still following the doctor's advice. Given the integer array candyType of length n, return the maximum number of different types of candies she can eat if she only eats n / 2 of them.

**Example 1:** Input: candyType = [1,1,2,2,3,3] Output: 3

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

**Solution:** class Solution:

**-----------** def distributeCandies(self, candyType: List[int]) -> int:

unique\_candies = 0

for i in range(len(candyType)):

for j in range(0, i):

if candyType[i] == candyType[j]:

break

else:

unique\_candies += 1

return min(unique\_candies, len(candyType) // 2)

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

**Complexity Analysis:**

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

* Time complexity : O(N2). We traverse over each of the N elements of candyType, and for each, we check all of the elements before it. Checking each item for each item is the classic O(N2) time complexity pattern.
* Space complexity : O(1). We don't allocate any additional data structures, instead only using constant space variables.

**===================================x=================================**

**Question 3** We define a harmonious array as an array where the difference between its maximum value and its minimum value is exactly 1. Given an integer array nums, return the length of its longest harmonious subsequence among all its possible subsequences. A subsequence of an array is a sequence that can be derived from the array by deleting some or no elements without changing the order of the remaining elements.

**Example 1:** Input: nums = [1,3,2,2,5,2,3,7] Output: 5

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

**Solution:** class Solution:

**-----------** def findLHS(self, nums):

numsLen, result = len(nums), 0

counts = {}

for val in nums:

if val in counts:

counts[val] += 1

else:

counts[val] = 1

inc = val + 1

dec = val - 1

if dec in counts:

result = max(result, counts[val] + counts[dec])

if inc in counts:

result = max(result, counts[val] + counts[inc])

return result

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

**Complexity Analysis:**

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

* Time complexity : O(n log n). Sorting takes O(n log n) time.
* Space complexity : O(log n). log n space is required by sorting in average case.

**===================================x=================================**

**Question 4** You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots. Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return true if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule and false otherwise.

**Example 1:** Input: flowerbed = [1,0,0,0,1], n = 1 Output: true

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

**Solution:** class Solution:

**-----------** def canPlaceFlowers(self, flowerbed: List[int], n: int) -> bool:

count = 0

for i in range(len(flowerbed)):

if flowerbed[i] == 0:

empty\_left\_plot = (i == 0) or (flowerbed[i - 1] == 0)

empty\_right\_lot = (i == len(flowerbed) - 1) or (flowerbed[i + 1] == 0)

if empty\_left\_plot and empty\_right\_lot:

flowerbed[i] = 1

count += 1

return count >= n

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

**Complexity Analysis:**

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

* Time complexity : O(n). A single scan of the flowerbed array of size n is done.
* Space complexity : O(n). Constant extra space is used.

**===================================x=================================**

**Question 5** Given an integer array nums, find three numbers whose product is maximum and return the maximum product.

**Example 1:** Input: nums = [1,2,3] Output: 6

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

**Solution:** class Solution:

**-----------** def maxProduct(arr, n):

if n < 3:

return -1

      arr.sort()

return max(arr[0] \* arr[1] \* arr[n - 1],

                arr[n - 1] \* arr[n - 2] \* arr[n - 3])

if \_\_name\_\_ == "\_\_main\_\_":

     arr = [1, 2, 3]

     n = len(arr)

     \_max = maxProduct(arr, n)

     if \_max == -1:

         print("No Triplet Exists")

     else:

         print("Maximum product is", \_max)

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

**Complexity Analysis:**

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

* Time complexity : O(n log n). Sorting the nums array takes n log n time.
* Space complexity : O(log n). Sorting takes O(log n) space.

**===================================x=================================**

**Question 6** Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

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

**Example 1:** Input: nums = [-1,0,3,5,9,12], target = 9 Output: 4

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

**Algorithm:**

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

1. Initialize the boundaries of the search space as left = 0 and right = nums.size - 1.

2. If there are elements in the range [left, right], we find the middle index mid = (left + right) / 2 and compare the middle value nums[mid] with target:

* If nums[mid] = target, return mid.
* If nums[mid] < target, let left = mid + 1 and repeat step 2.
* If nums[mid] > target, let right = mid - 1 and repeat step 2.

3. We finish the loop without finding target, return -1.

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

**Solution:** class Solution:

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

left = 0

right = len(nums) - 1

while left <= right:

mid = (left + right) // 2

if nums[mid] == target:

return mid

elif nums[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

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

**Complexity Analysis:**

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

Let n be the size of the input array nums.

* Time complexity : O(logn). nums is divided into half each time. In the worst-case scenario, we need to cut nums until the range has no element, and it takes logarithmic time to reach this break condition.
* Space complexity : O(1). During the loop, we only need to record three indexes, left, right, and mid, they take constant space.

**===================================x=================================**

**Question 7** An array is monotonic if it is either monotone increasing or monotone decreasing. An array nums is monotone increasing if for all i <= j, nums[i] <= nums[j]. An array nums is monotone decreasing if for all i <= j, nums[i] >= nums[j].

Given an integer array nums, return true if the given array is monotonic, or false otherwise.

**Example 1:**  Input: nums = [1,2,2,3] Output: true

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

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

**-----------** def isMonotonic(self, A):

store = 0

for i in xrange(len(A) - 1):

c = cmp(A[i], A[i+1])

if c:

if c != store != 0:

return False

store = c

return True

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

**Complexity Analysis:**

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

* Time Complexity : O(N), where N is the length of A.
* Space Complexity: O(1).

**===================================x=================================**