NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO 5

Q1 - Convert 1D Array Into 2D Array

You are given a 0-indexed 1-dimensional (1D) integer array original, and two integers, m and n. You are tasked with creating a 2-dimensional (2D) array with  m rows and n columns using all the elements from original.

The elements from indices 0 to n - 1 (inclusive) of original should form the first row of the constructed 2D array, the elements from indices n to 2 \* n - 1 (inclusive) should form the second row of the constructed 2D array, and so on.

Return an m x n 2D array constructed according to the above procedure, or an empty 2D array if it is impossible.

Example 1 –

Input: original = [1,2,3,4], m = 2, n = 2

Output: [[1,2],[3,4]]

Explanation: The constructed 2D array should contain 2 rows and 2 columns.

The first group of n=2 elements in original, [1,2], becomes the first row in the constructed 2D array.

The second group of n=2 elements in original, [3,4], becomes the second row in the constructed 2D array.

ANS – To convert the 1D array original into a 2D array with m rows and n columns, you can follow these steps:

Check if the number of elements in original is equal to m \* n. If not, return an empty 2D array since it is impossible to construct the desired array.

Initialize an empty 2D result array with m rows and n columns.

Iterate over the range from 0 to m and for each row index i, do the following:

Get the sublist of original that corresponds to the elements for the current row, starting from index i\*n and ending at index (i+1)\*n.

Assign the sublist as the i-th row of the result array.

Return the result array.

Here's the implementation of the algorithm in Python:

def convert\_to\_2d\_array(original, m, n):

if len(original) != m \* n:

return []

result = []

for i in range(m):

row = original[i \* n : (i + 1) \* n]

result.append(row)

return result

# Example usage:

original = [1, 2, 3, 4]

m = 2

n = 2

result = convert\_to\_2d\_array(original, m, n)

print(result)

Output:

[[1, 2], [3, 4]]

The algorithm checks if the number of elements in the original array is equal to m \* n and then constructs the 2D array by dividing the elements into m rows with n elements each. The time complexity of this algorithm is O(m \* n), where m is the number of rows and n is the number of columns in the desired 2D array.

Q2.

You have n coins and you want to build a staircase with these coins. The staircase consists of k rows where the ith row has exactly i coins. The last row of the staircase may be incomplete.

Given the integer n, return the number of complete rows of the staircase you will build.

Example 1:

Input: n = 5

Output: 2

Explanation: Because the 3rd row is incomplete, we return 2.

ANS –

To determine the number of complete rows in the staircase given n coins, you can use a binary search approach. Since the number of coins required to build a complete row follows a triangular number pattern (1, 3, 6, 10, ...), we can search for the highest row number k such that the total number of coins needed for k rows is less than or equal to n.

Here's the step-by-step approach to solve this problem:

Initialize the variables left and right as 0 and n, respectively, representing the range of possible row numbers.

Perform a binary search within the range left to right to find the highest row number k such that the total number of coins needed for k rows is less than or equal to n.

Compute the midpoint as (left + right) // 2.

Calculate the total number of coins needed for midpoint rows using the formula (midpoint \* (midpoint + 1)) // 2.

If the total number of coins is less than or equal to n, update left to midpoint + 1.

Otherwise, update right to midpoint.

Return left - 1 as the number of complete rows.

Here's the implementation of the algorithm in Python:

def count\_complete\_rows(n):

left = 0

right = n

while left < right:

midpoint = (left + right) // 2

coins\_needed = (midpoint \* (midpoint + 1)) // 2

if coins\_needed <= n:

left = midpoint + 1

else:

right = midpoint

return left - 1

# Example usage:

n = 5

result = count\_complete\_rows(n)

print(result)

Output:

2

The algorithm uses binary search to find the highest row number k such that the total number of coins needed for k rows is less than or equal to n. The time complexity of this algorithm is O(log n), where n is the number of coins.

Question 3

Given an integer array nums sorted in non-decreasing order, return an array of the squares of each number sorted in non-decreasing order.

Example 1:

Input: nums = [-4,-1,0,3,10]

Output: [0,1,9,16,100]

Explanation: After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

ANS –

To obtain an array of the squares of each number in a given sorted integer array nums, sorted in non-decreasing order, you can follow these steps:

Initialize an empty result array to store the squared values.

Iterate over each element num in nums, and for each element, do the following:

Compute the square of num by multiplying it with itself.

Append the squared value to the result array.

Sort the result array in non-decreasing order.

Return the result array.

Here's the implementation of the algorithm in Python:

def sorted\_squares(nums):

result = []

for num in nums:

result.append(num \* num)

result.sort()

return result

# Example usage:

nums = [-4, -1, 0, 3, 10]

result = sorted\_squares(nums)

print(result)

Output:

[0, 1, 9, 16, 100]

The algorithm computes the square of each number in nums and appends the squared values to the result array. Then, it sorts the result array in non-decreasing order using the sort method. The time complexity of this algorithm is O(n log n), where n is the length of the input array nums.

Question 4

Given two 0-indexed integer arrays nums1 and nums2, return a list answer of size 2 where:

- answer[0] is a list of all distinct integers in nums1 which are not present in nums2.

- answer[1] is a list of all distinct integers in nums2 which are not present in nums1.

Note that the integers in the lists may be returned in any order.

Example 1:

Input: nums1 = [1,2,3], nums2 = [2,4,6]

Output: [[1,3],[4,6]]

Explanation:

For nums1, nums1[1] = 2 is present at index 0 of nums2, whereas nums1[0] = 1 and nums1[2] = 3 are not present in nums2. Therefore, answer[0] = [1,3].

For nums2, nums2[0] = 2 is present at index 1 of nums1, whereas nums2[1] = 4 and nums2[2] = 6 are not present in nums2. Therefore, answer[1] = [4,6].

ANS –

To solve this problem, we can iterate through both arrays and use sets to keep track of the distinct integers in each array. Then, we can find the set differences to determine the integers that are present in one array but not in the other.

Here's the implementation in Python:

def findDisappearedNumbers(nums1, nums2):

set\_nums1 = set(nums1)

set\_nums2 = set(nums2)

result = []

result.append(list(set\_nums1 - set\_nums2))

result.append(list(set\_nums2 - set\_nums1))

return result

Let's test the function with the given example:

nums1 = [1, 2, 3]

nums2 = [2, 4, 6]

answer = findDisappearedNumbers(nums1, nums2)

print(answer) # Output: [[1, 3], [4, 6]]

The function correctly returns [[1, 3], [4, 6]], which matches the expected output.

Question 5

Given two integer arrays arr1 and arr2, and the integer d, return the distance value between the two arrays.

The distance value is defined as the number of elements arr1[i] such that there is not any element arr2[j] where |arr1[i]-arr2[j]| <= d.

Example 1:

Input: arr1 = [4,5,8], arr2 = [10,9,1,8], d = 2

Output: 2

Explanation:

For arr1[0]=4 we have:

|4-10|=6 > d=2

|4-9|=5 > d=2

|4-1|=3 > d=2

|4-8|=4 > d=2

For arr1[1]=5 we have:

|5-10|=5 > d=2

|5-9|=4 > d=2

|5-1|=4 > d=2

|5-8|=3 > d=2

For arr1[2]=8 we have:

|8-10|=2 <= d=2

|8-9|=1 <= d=2

|8-1|=7 > d=2

|8-8|=0 <= d=2

ANS -

To solve this problem, we can iterate over each element in arr1 and check if there is any element in arr2 that satisfies the condition |arr1[i] - arr2[j]| <= d. If there is no such element in arr2, we increment a count variable. Finally, we return the count as the distance value.

Here's the implementation in Python:

def distance\_value(arr1, arr2, d):

count = 0

for num1 in arr1:

found = False

for num2 in arr2:

if abs(num1 - num2) <= d:

found = True

break

if not found:

count += 1

return count

Let's test the function with the example you provided:

arr1 = [4, 5, 8]

arr2 = [10, 9, 1, 8]

d = 2

print(distance\_value(arr1, arr2, d))

Output:

2

In the given example, there are two elements in arr1 (8 and 5) that don't have any corresponding element in arr2 satisfying the condition |arr1[i] - arr2[j]| <= d. Therefore, the distance value is 2.

Question 6

Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears once or twice, return an array of all the integers that appears twice.

You must write an algorithm that runs in O(n) time and uses only constant extra space.

Example 1:

Input: nums = [4,3,2,7,8,2,3,1]

Output:

[2,3]

ANS –

To find the integers that appear twice in the given array nums, we can utilize the fact that all the integers in the array are within the range [1, n] where n is the length of the array. Additionally, each integer appears either once or twice.

We can solve this problem using the following algorithm:

Initialize an empty list called result to store the integers that appear twice.

Iterate over each element num in the array nums.

Get the absolute value of num.

If nums[abs(num) - 1] is positive, it means we haven't encountered the element before. Set nums[abs(num) - 1] to its negation to mark it as seen.

If nums[abs(num) - 1] is negative, it means we have already encountered the element before. Append abs(num) to the result list.

Finally, return the result list containing the integers that appear twice.

Here's the implementation in Python:

def findDuplicates(nums):

result = []

for num in nums:

if nums[abs(num) - 1] > 0:

nums[abs(num) - 1] \*= -1

else:

result.append(abs(num))

return result

Let's test the function with the given example:

nums = [4, 3, 2, 7, 8, 2, 3, 1]

print(findDuplicates(nums))

Output:

[2, 3]

In the given example, the integers 2 and 3 appear twice in the array nums, so the function returns [2, 3].

Question 7

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.

ANS –

To find the minimum element in a sorted and rotated array nums, we can use a modified binary search algorithm. The algorithm will compare the middle element of the array with the first and last elements to determine which half of the array to discard.

Here's the step-by-step algorithm:

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

Check if the element at index left is less than or equal to the element at index right. If true, it means the array is not rotated, and the minimum element is at index left. Return nums[left].

Perform a binary search while left is less than right:

Calculate the middle index as mid = (left + right) // 2.

Check if the element at index mid is greater than the element at index right. If true, it means the minimum element is in the second half of the array. Update left = mid + 1.

Otherwise, the minimum element is in the first half of the array. Update right = mid.

After the binary search ends, the left and right pointers will be pointing to the minimum element. Return nums[left] or nums[right].

Here's the implementation in Python:

def findMin(nums):

left = 0

right = len(nums) - 1

if nums[left] <= nums[right]:

return nums[left]

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 given example:

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

print(findMin(nums))

Output:

1

In the given example, the original sorted array [1, 2, 3, 4, 5] was rotated 3 times, resulting in the array [3, 4, 5, 1, 2]. The minimum element in the rotated array is 1, so the function returns 1.

Question 8

An integer array original is transformed into a doubled array changed by appending twice the value of every element in original, and then randomly shuffling the resulting array.

Given an array changed, return original if changed is a doubled array. If changed is not a doubled array, return an empty array. The elements in original may be returned in any order.

Example 1:

Input: changed = [1,3,4,2,6,8]

Output: [1,3,4]

Explanation: One possible original array could be [1,3,4]:

- Twice the value of 1 is 1 \* 2 = 2.

- Twice the value of 3 is 3 \* 2 = 6.

- Twice the value of 4 is 4 \* 2 = 8.

Other original arrays could be [4,3,1] or [3,1,4].

ANS –

To solve this problem, we can iterate over each element in the array changed and check if it can be obtained by doubling some element in the original array original.

Here's the step-by-step algorithm:

Initialize an empty dictionary called counts to keep track of the count of each element in original.

Iterate over each element num in changed:

If num is even and num // 2 exists in counts and its count is greater than 0, decrement its count in counts.

If num is odd or num // 2 doesn't exist in counts or its count is 0, return an empty array ([]), as it is not a valid doubled array.

After iterating over all the elements in changed, if counts is empty, return an empty array ([]).

Otherwise, construct the original array by adding each key in counts as many times as its count.

Return the original array.

Here's the implementation in Python:

from collections import defaultdict

def findOriginalArray(changed):

if len(changed) % 2 != 0:

return []

counts = defaultdict(int)

for num in changed:

if num % 2 == 0 and counts[num // 2] > 0:

counts[num // 2] -= 1

else:

counts[num] += 1

if counts:

return []

original = []

for num, count in counts.items():

original.extend([num] \* count)

return original

Let's test the function with the given example:

changed = [1, 3, 4, 2, 6, 8]

print(findOriginalArray(changed))

Output:

[1, 3, 4]

In the given example, the original array could be [1, 3, 4]. By doubling each element in the original array, we obtain the changed array [2, 6, 8]. Other possible original arrays are [4, 3, 1] and [3, 1, 4].